$_m$Java 到 Micro-Dalvik 虚拟机的编译验证

江 南[1,3],何炎祥[1,2],张晓瞳[1,2]
(1. 武汉大学计算机学院,湖北武汉 430072;2. 武汉大学软件工程国家重点实验室,湖北武汉 430072;
3. 湖北工业大学计算机学院,湖北武汉 430070)

摘 要: 针对类 Java 的面向对象语言$_m$Java 到类 Dalvik 的寄存器架构虚拟机 Micro-Dalvik 的编译验证,给出了$_m$Java 语言和 Micro-Dalvik 的操作语义.从$_m$Java 语言程序到 Micro-Dalvik 虚拟机指令的编译分为两步,首先将$_m$Java 语言程序中的本地变量名转换为相应的序号,得到一个中间语言程序,再将该中间语言程序翻译成 Micro-Dalvik 虚拟机指令程序.在给出中间语言的操作语义后,构造了$_m$Java 语言程序与编译后的中间语言程序的语义保持定理并证明,以及构造了中间语言程序的语义与编译后的 Micro-Dalvik 虚拟机程序的语义保持定理并证明.整个形式化编译验证在定理证明助手 Isabelle/HOL 中进行了机器检测.$_m$Java 语言和 Micro-Dalvik 虚拟机分别对 Java 语言和 Dalvik 虚拟机进行了抽象,是我们兼顾语言的真实性和形式化的清晰性的结果.但是,所有形式化的语义严格遵从语言规范中的定义,并与 Dalvik VM 的实现保持一致,从这种意义上讲,该编译器并不是一个实验性质的假想编译器,而是有其实用意义的.

关键词: 编译验证;定理证明;操作语义;机器检测;寄存器架构;面向对象语言
中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112 (2016)07-1619-11
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2016.07.015

Formal Verification of $_m$Java Compiler Targeting Micro-Dalvik Virtual Machine

JIANG Nan[1,3], HE Yan-xiang[1,2], ZHANG Xiao-tong[1,2]
(1. *Computer School, Wuhan University, Wuhan, Hubei* 430072, *China*;
2. *State Key Laboratory of Software Engineering, Wuhan University, Wuhan, Hubei* 430072, *China*;
3. *Computer School, Hubei University of Technology, Wuhan, Hubei* 430070, *China*)

Abstract: This paper introduces a formal verification of $_m$Java compiler targeting Micro-Dalvik virtual machine (VM) where $_m$Java is an object-oriented language similar to Java, and Micro-Dalvik is a Dalvik-like VM of the register-based architecture. The operational semantics of $_m$Java and Micro-Dalvik VM are defined. The compiler operates in two stages. First it replaces the names of local variables by their corresponding indices and hence translates $_m$Java into an intermediate language. Then it generates the Micro-Dalvik VM instructions. After defining the operational semantics of the intermediate language, the correctness of the two stages are formulated in terms of the preservation of the semantics and is proved respectively. The whole formalization is machine-checked in the theorem proof assistant Isabelle/HOL. The $_m$Java language and Micro-Dalvik VM are more abstract than the comparable Java and the Dalvik VM, respectively, which is a result of a compromise between the the realism of the language and the clarity of the formalization. However, $_m$Java language and Micro-Dalvik VM exhibit core features of an object-oriented programming language and a register-based architecture, respectively, and thus in this sense, this verified compiler is non-trivial.

Key words: compiler verification; theorem proving; formal semantics; machine-check; register-based architecture; object-oriented

收稿日期:2015-12-01;修回日期:2016-03-10;责任编辑:李勇锋
基金项目:国家自然科学基金(No. 61373039)

1 引言

编译器是将高级语言编写的程序转换到能在硬件平台上运行的指令集的重要系统软件.商用编译器虽然经过大量测试,但仍然存在许多问题,这些问题不断公布在其缺陷列表的网站上[1~3].编译器错误产生的原因可以归纳为两点,其一是高级语言的语义规范通常复杂,且以自然语言书写,导致编译器难以理解一些模糊的语义定义,不知道它应该将之翻译为怎样的机器指令,只能采取试着运行再观察的态度(run it and see)[4];其二是编译器本身作为一个复杂的软件,也可能有错.这些错误对于安全攸关的软件系统来说,即使出现概率很小,也可能造成重大的人员伤亡与经济损失.由于编译器的不可信,实践中,一些控制系统的关键部分不得不采用汇编语言编写并在汇编级进行验证[1].但是,随着安全攸关的软件大小和复杂度与日俱增,使用汇编语言来开发这些软件已不可行[4].因此,迫切需要构建一个经过形式化验证的编译器,这个证明正确的编译器能够保证最后运行的机器程序是无错的.

为了得到一个验证的编译器,McCarthy 和 Painter 进行了首次尝试[5].其源语言是由代表数值的常量或者变量以及加运算所组成的算术表达式,目标机器是具有一个累加寄存器的简单冯诺依曼机器[1,5].他们使用抽象语法来定义语义,这种方法成为定义编程语言语义的典范[6],但是他们的方法被认为过于泛化[7,8].相对普通程序的验证而言,编译器可以被更好地结构化,因此,Morris 提出使用严格定义的数学结构来表达编译器正确性问题的本质[1].所表达的验证思想是:编译器被定义为由源程序到目标程序的数学函数;语义定义为程序到数学值的映射函数,该值可以是非常复杂的类型;目标语义函数表示的语义代表源语义函数所表示的语义[7].这种验证思想流行于后来的大多数编译器验证研究中[8].

尽管 Morris 给出了指导性方法,编程语言的语义以及编译前后两种语言语义的保持性定义和证明并不容易.研究者们采取不同的语义,对应于不同的证明方式,在命令式语言的编译器验证领域进行了深入研究,这些方法可以归纳为两大类别.

(1) 先定义源语言和目标语言的语义,再定义编译规则,最后构造语义保持定理,证明采用归纳[9~11].代表性成果包括:Stepney 基于指称语义,完成了从一个假定的编程语言到汇编语言的编译验证[4];Polak 同样基于指称语义,验证了一个 Pascal 语言子集的编译器[12];Leroy 等人基于操作语义,开发了 CompCert 编译器,其源语言是 C 语言子集,目标语言是 PowerPC 汇编代码[13~15];Klein 和 Nipkow 等人验证了一个称为 Jinja 的 Java 子集到 Java 虚拟机(Virtual Machine,VM)字节码的编译器[16,17].

(2) 定义程序转换规则,源语言程序按照转换规则被逐步精化为可执行的目标语言程序.转换规则即是编译器行为,规则的可靠性保证编译的正确性.这种方法通常采用代数语义.Hoare、He、以及 Sampaio 等人使用这种方法将源程序精化为范式,范式和相关的精化定理可用于针对不同目标机器的编译器设计[18,19].Duran 等人将该方法扩展到了一种类似 Java 的面向对象语言[20].

McCarthy 于 1961 年也指出,我们应该证明程序符合其规范,而不是测试,并且证明需要经过某个计算机程序的检查.编译器作为一个特殊的程序,其规范与源、目标语言的语义直接相关,真实语言的语义往往庞大而复杂,譬如 Java 官方语言规范有近 700 页的描述,JVM 规范有约 600 页,这些加大了形义定义的困难性,以及语义保持及证明的复杂性.定理证明系统,如 Coq、Isabelle/HOL、ACL2、PVS 等的出现在编译器验证的实用性方面起到了关键作用.在定理证明系统的支持下进行语义表达和证明更为高效,经过机器检测的编译器具有更高的可靠性和可维护性[16,21~23].CompCert 的整个开发工作在定理证明助手 Coq 中完成,其源语言是一个 C 语言子集,包括了能够运用于编程安全攸关的工业嵌入式软件所需要的语法成份,所产生代码的执行性能接近于 O1 优化级 GCC[15].

在近半个世纪的探索研究之后,过程式语言的编译验证得到了较好的理解与实施,出现了象 CompCert 这个被广泛认可的 C 语言的验证编译器.Eiffel 语言设计中缺陷的发现促使编程语言,特别是面向对象语言的设计们致力于证明语言的类型可靠性,导致了 Java 语言、Java 虚拟机(JVM)、JVM 字节码验证的形式化得到了广泛研究[24~28];近几年来由于 Android 成为一个广泛使用的移动设备平台,虽然该平台上的应用程序用 Javia 编写,但是被编译成了 Dalvik 字节码,而不是 Java 字节码,使得已有的 Java 程序分析工具不能直接使用或者难以使用,而运行在这个平台上的应用程序的安全性和可靠性问题变得越来越重要,因此,Android Dalvik VM 程序的静态分析以及形式化研究也陆续展开[29~31].虽然象 Java 语言、JVM、以及 Android Dalvik VM 的形式化研究以及 JVM 字节码验证都取得了一定进展,然而在支持 OO 语言所增加的封装、继承、方法覆盖、动态绑定和多态等语言特性的编译验证领域,目前成果并不多见,还有待深入研究.文献[20]使用代数的方法构造了一个类 Java 语言到 JVM 的编译器,文献[24]使用抽象状态机定义了 Java 语言,并对编译到 Ja-

va VM 的正确性进行了证明,但他们都未经过机器检测. 目前尚未见有针对 Dalvik VM 的编译验证.

本文使用定理证明助手 Isabelle/HOL[32] 形式化验证一个编译器,其源语言$_m$Java 是一个较为完善的符合 OO 特点的顺序语言子集,目标机器以典型寄存器架构的虚拟机 Android Dalvik VM 为参考.

2　$_m$Java 的形式化

本文讨论的源语言称为$_m$Java,$_m$Java 包括了封装、继承、多态、异常处理,目前暂未考虑线程和数组. 式(1)定义了$_m$Java 语言的语法成分,其中,$'a$ 是类型变量,因此可以用来同时表达$_m$Java 程序的语法和编译时生成的中间语言程序的语法. 本地变量的声明出现在块语句中,它是一个变量名和其类型组成的二元组的列表.

$$\begin{aligned}
&'a\ expr = \mathrm{Val}\ val\\
&\mid \mathrm{Var}\ 'a\\
&\mid \mathrm{Binop}\ ('a\ expr)\ bop\ ('a\ expr)\ _ << _ >> _\\
&\mid \mathrm{New}\ cname\\
&\mid \mathrm{LAss}\ 'a('a\ expr)\ _ := _\\
&\mid \mathrm{Cond}\ ('a\ expr)\ ('a\ expr)\ ('a\ expr)\ \mathrm{if}(_)\ _\ \mathrm{else}\ _\\
&\mid \mathrm{Seq}\ ('a\ expr)\ ('a\ expr)\ _;;_\\
&\mid \mathrm{Block}(('a \times ty)\ list)('a\ expr)\{_;_\}\\
&\mid \mathrm{ThrowEx}('a\ expr)\\
&\mid \mathrm{Cast}\ cname\ ('a\ expr)\\
&\mid \mathrm{TryCatch}('a\ expr)cname\ 'a('a\ expr)try_catch(__)_\\
&\mid \mathrm{FAcc}('a\ expr)vname\ cname\ _\cdot_\{_\}\\
&\mid \mathrm{FAss}\ ('a\ expr)\ vname\ cname('a\ expr)_\cdot_\{_\} := _
\end{aligned}\tag{1}$$

式(2)中的$'m$ 是类型变量,可以用来代表$_m$Java 和 Micro-Dalvik 程序中的方法体;ty 代表类型,可以是空类型 *Void*,布尔类型 *Boolean*,整型 *Integer*,空引用类型 *NT*,类类型 *Class cname*. 式(2)表示,程序(*prog*)是类声明(*cdecl*)的列表;类声明是类名和类(*class*)的二元组;类是由其超类名、变量声明(*vdecl*)列表以及成员方法声明(*mdecl*)列表组成的三元组;变量声明是变量名和其类型的二元组,方法声明是由其方法名、形参类型列表、返值类型和方法体组成的四元组.

$$\begin{aligned}
&'m\ prog = 'm\ cdecl\ list\\
&'m\ cdecl = cname \times 'm\ class\\
&'m\ class = cname \times vdecl\ list \times 'm\ mdecl\ list\\
&vdecl = vname \times ty\\
&'m\ mdecl = mname \times ty\ list \times ty \times 'm
\end{aligned}\tag{2}$$

$_m$Java 语言程序如式(3)所示,其中,J_expr 使用变量名 *vname* 取代了类型参数$'a$,表示$_m$Java 程序的合法语句是式(1)中的 15 个表达式语句;J_mb 表示$_m$Java 程序的方法体,它是形参变量名列表和 J_expr 组成的二元组. 因此得到$_m$Java 程序的定义.

$$\begin{aligned}
&J_prog = J_mb\ prog\\
&J_mb = vname\ list \times J_expr\\
&J_expr = vname\ expr
\end{aligned}\tag{3}$$

$_m$Java 程序状态(*state*)的完整定义如式(4)所示,它是堆(*heap*)和本地变量(*locals*)的二元组;堆是地址(*addr*)到对象(*obj*)的映射;地址是自然数类型 *nat*;对象是类名和其成员变量(*fields*)的二元组;成员变量和本地变量的区别是:成员变量的定义中包括了定义该成员变量的类名,因此是其名称和该类名所组成的的二元组到其值的映射,而本地变量则仅是其名称到值的映射. 值由 *val* 定义,可以是空引用值(*Null*)、布尔值(*Bool bool*)、整数值(*Intg int*)、以及对象的地址(*Addr addr*).

$$\begin{aligned}
&state = heap \times locals\\
&heap = addr \rightharpoonup obj\\
&addr = nat\\
&obj = cname \times fields\\
&fields = vname \times cname \rightharpoonup val\\
&locals = vname \rightharpoonup val
\end{aligned}\tag{4}$$

程序的语义定义采取大步操作语义(Big-Step Operational Semantics)[32,33],以归纳谓词(inductively defined predicate)的方式给出,其语义规则的形式如式(5)所示,表示:执行程序 P 中组成方法体的表达式 e,从初始状态(h,l)到达新的状态(h',l'),e 计算到 e'.

$$P \vdash [e,(h,l)] \Rightarrow [e',(h',l')]\tag{5}$$

方法调用时的参数是表达式列表,为了对表达式列表进行计算,扩展了式(5)的定义为:如果表达式列表为空,则状态不发生改变;如果表达式形如 *e#es*,其中,#是单个元素 e 加在列表 *es* 的首部,则先计算单个表示式 e 的值,剩余列表再按形如 *e#es* 进行递归计算,每次计算的值组成新的值列表.

语义规则参照 Java 语言规范[34]. 在此给出较为复杂的方法调用 $e.M(ps)$ 在当前状态 s_0 下的语义计算规则,如式(6)所示.

$$\frac{\begin{array}{l}
P \vdash [e,s_0] \Rightarrow [addr\ a,s_1];\\
P \vdash [ps,s_1][\Rightarrow][map\ Val\ vs,(h_2,l_2)];\\
h_2 a = \lfloor (C,fs) \rfloor \ Ts = map(the \circ typeof_h)\ vs;\\
seesMethod\ P\ C\ M\ Ts\ T\ (pns,body)\ D;\\
length\ vs = length\ pns;\\
l_2' = [this \mapsto Addr\ a, pns[\mapsto]vs];\\
P \vdash [body,(h_2,l_2')] \Rightarrow [e',(h_3,l_3)]
\end{array}}{P \vdash [e \cdot M(ps),s_0] \Rightarrow [e',(h_3,l_2)]}\tag{6}$$

式(6)表示:当对方法调用语句进行计算时,首先计算 e,得到其值 *addr a*,到达新的状态 s_1,*addr a* 代表调用方法的对象的地址;接着,在状态 s_1 下,计算形参表达式列表,得到形参值列表,并更新状态到(h_2,l_2);如果在堆 h_2 中,地址 a 代表着一个对象(C,fs),Ts 是形参计算得到的值列表所对应的类型列表,对于程序 P 中的类 C 而言,方法名为 M,形参类型列表为 Ts 的方法是可见的,这个方法的返值类型是 T,方法体是(pns,$body$),通过 *seesMethod* 完成;形参值列表长度与形参名列表长度相等,则为这个调用方法建立新的本地变量 l_2',在这个新的映射中,*this* 映射到地址 a,形参名列表映射到值列表,其中,加了中括号的映射符号代表两个列表的各个元素按照顺序对应映射;最后,在状态(h_2,l_2')下,执行调用方法的方法体,得到 e',和新的状态(h_3,l_3). 所有这些计算完成后,方法调用语句的最终结果为 e',堆状态为 h_3,但是本地变量恢复到调用方法前的本地变量 l_2.

函数 *seesMethod* 调用了归纳定义的谓词 *seesMethodDef*,它的归纳定义如式(7)所示. 式(7)获得一个映射,即映射方法名和方法参数列表的二元组到返值和方法体的二元组,因此,根据给定的(M,Ts),式(8)可以查找到其对应的方法 m. 式(7)的实现为,由函数 *getClassDef* 获得指定类名的类定义为(D,fs,ms),如果对象类不是类 *Object*,则按类层次关系向上查找类中定义的方法列表. 函数 *map_of* 中的 *lambda* 演算将表示方法的四元组变换成三元组,其中第一个元素为二元组(*mName*,*Ts*),然后 *map_of* 函数中的将元组列表转换成一个映射,这个映射与第一个 *lambda* 演算形成函数组合. 父类获得的映射在 ++ 运算符的左边,子类获得的映射在右边,如果 m_1 和 m_2 代表两个映射,$m_1 ++ m_2$ 运算规则是:如果在映射 m_2 中查找到对应在的方法,则结果为该方法,若没有找到对应的方法,即为 *None*,再在映射 m_1 中查找是否有对应的方法. 最后的效果为:如果父类和子类具有同名同参数列表的方法,由于子类中的查找在前,结果为子类中定义的方法,即子类覆盖了父类的方法. 由于方法查找时,方法名和形参类型同时作为参数,因此对于同名而不同形参的方法,视为不同的方法,即方法重载.

seesMethodDefInduct:

$$\left\|\begin{array}{l} getClassDef\ P\ C = \lfloor (D, fs, ms) \rfloor; \\ C \neq Object; seesMethodDef\ P\ D\ MmNameTs; \\ MmNameTs' = MmNameTs + + \\ (Option.map\ (\lambda TyMbody.\ (TyMbody, C)))^{\circ} \\ (map\_of\ (\lambda (mName, Ts, T, mbody). \\ ((mName, Ts), T, mbody))\ ms))) \end{array}\right\| \Rightarrow \quad (7)$$

seesMethodDef P C MmNameTs'

$$\begin{array}{l} seesMethod\ P\ C\ M\ Ts\ T\ m\ D \equiv \\ \exists Mm.\ \ seesMethodDef\ P\ C\ Mm \wedge \\ Mm(M, Ts) = \lfloor ((T, m), D) \rfloor \end{array} \quad (8)$$

以上方法覆盖要求方法名和参数类型相同,返值类型不大于被覆盖方法的返值类型,式(9)给出了这个定义,它是定义程序 *well-formedness* 的一个重要组成部分.

$$\begin{array}{l} \lambda(C, (D, fs, ms)).\ (\forall m \in set\ ms.\ distinct\_fst \\ (map(\lambda(M, Ts, T, mb).((M, Ts), T, mb))\ ms) \wedge \\ (C \neq Object \longrightarrow is\_class\ P\ D \wedge \\ (\forall (M, Ts, T, mb) \in ms.\ \ \forall D'\ T'\ m'. \\ seesMethod\ P\ D\ M\ Ts\ T'\ m'\ D' \longrightarrow T \leqslant T' \end{array} \quad (9)$$

3 Micro-Dalvik 的操作语义

目标机器是寄存器架构的虚拟机 Micro-Dalvik,参考 Android Dalvik[35~38]. Micro-Dalvik 虚拟机的抽象指令如式(10)所示.

$$\begin{array}{l} instr = \text{Const}\ nat\ nat \\ \quad | \text{Iget}\ nat\ nat\ vname\ vname \\ \quad | \text{Iput}\ nat\ nat\ vname\ vname \\ \quad | \text{AriBinop}\ ariop\ nat\ nat\ nat \\ \quad | \text{Cmpeq/}ne/gt/ge/lt/le\ nat\ nat\ nat \\ \quad | \text{IFfalse}\ idx\ int \\ \quad | \text{Goto}\ branchoffset \\ \quad | \text{Return}\ nat \\ \quad | \text{Invoke}\ nat\ nat\ mname \\ \quad | \text{NewIntance}\ nat\ cname \\ \quad | \text{Throw}\ nat \\ \quad | \text{CheckCast}\ nat\ cname \\ \quad | \text{Move}\ nat\ nat \\ \quad | \text{MoveResult}\ nat \\ \quad | \text{MoveException}\ nat \end{array} \quad (10)$$

Micro-Dalvik 虚拟机程序定义如式(11)所示,其中,Micro-Dalvik 程序(*dvm_prog*)的定义是将式(2)中第一行的类型变量 $'m$ 替换为其方法体(*dvm_mb*),它是三元组,第一个分量表示寄存器个数(不计 *this* 和实参的后 N 个寄存器),该值在编译时确定,第二个分量是指令列表,最后一个分量是编译时生成的异常表(*ex_table*). 异常表是异常表项(*ex_entry*)的列表,一个异常表项由 *try* 块起始指令程序计数 *pc*、结束指令程序计数 *pc*、捕获异常类型 *cname* 以及对应的 *catch* 块的起始指令程序计数 *pc* 组成. *pc* 是自然数类型的别名.

$$\begin{array}{l} dvm\_prog = dvm\_mb\ prog \\ dvm\_mb = nat \times instr\ list \times ex\_table \\ ex\_table = ex\_entry\ list \\ ex\_entry = pc \times pc \times cname \times pc \end{array} \quad (11)$$

式(12)给出了 Micro-Dalvik 虚拟机状态 dvm_state,它由是否产生异常、堆(heap)、栈桢(frame)列表和方法返回值组成,如果有异常产生,*val option* 的值为异常对象的地址(*Addr a*),否则其值为 None. 栈桢由捕获的异常、寄存器中存放的值的列表、方法所在的类名、方法名、形参列表以及程序计数组成,如果捕获到异常,*xcptval* 的值为异常对象的地址. 堆(heap)的定义与源语言$_m$Java 相同,如式(4)所示.

$$\begin{aligned} &dvm\_state = val\ option \times heap \times frame\ list \times rtval\\ &frame = xcptval \times val\ list \times cname \times mname \times ty\ list \times pc \end{aligned} \tag{12}$$

Micro-Dalvik 虚拟机程序的操作语义与源$_m$Java 程序大步操作语义的定义方式不同,它先定义函数 *run*,通过调用函数 *run_instr* 给出了单步执行的状态转换,如式(13)所示.

$$\begin{aligned} &run(P,xp,h,[\,],r) = \mathrm{None}\\ &run(P,xp,h,(x,reg,C,M,ts,pc)\#frs,r) =\\ &(let\ i = instrs\_of\ P\ C\ M\ ts!\ pc;(xp,h',frs',r') =\\ &run\_instr\ i\ P\ h\ x\ reg\ C\ M\ ts\ pc\ frs\ r\ in\\ &\left\lfloor \begin{array}{l} case\ xp\ of\ None \Rightarrow (None,h',frs',r') \lfloor a \rfloor \Rightarrow \\ lookupH\ P\ a\ h((x,reg,C,M,ts,pc)\#frs)r \end{array} \right\rfloor) \end{aligned} \tag{13}$$

使用上述函数式的定义,关系方式的语义定义 *runRel* 实现为状态转换的集合,因此,程序的执行 *runProg* 由任意有限步的单步执行组成,为了方便表示,将 Micro-Dalvik 虚拟机程序的语义定义 *runProg P* σ σ'写为如式(14)的形式,它表示在初始虚拟机状态 σ 下执行程序 *P*,到达新的状态 σ'.

$$P \vdash \sigma \xrightarrow{dvm} \sigma' \tag{14}$$

状态转换函数 *run_instr* 以及详细阐述可详见文献[39].

4　编译及证明

4.1　$_m$Java 到 Micro-Dalvik VM 的编译

4.1.1　$_m$Java 程序的编译方法

在不考虑优化的情况下,$_m$Java 源程序 *J_prog* 到 Micro-Dalvik VM 目标程序 *dvm_prog* 的翻译是将组成源程序方法体的抽象表达式 *J_epr* 中的每一个构造器所构造的语句翻译成一条或多条目标机器指令,即 *instr* 定义的指令. 由于源程序中的本地变量名不再出现在机器指令中,这些值按照其声明的顺序存储在一片连续的寄存器中,为避免一遍翻译及证明的复杂,定义一个中间语言 *Ji_prog*,如式(15)所示.

$$\begin{aligned} &Ji\_prog = Ji\_mb\ prog\\ &Ji\_mb = nat \times Ji\_epr\\ &Ji\_epr = nat\ epr \end{aligned} \tag{15}$$

其中,*Ji_epr* 用 *nat* 取代了式(1)中的类型变量$'a$,即中间语言程序中的语句中不再出现方法本量名,而是方法变量名对应的序号. *Ji_mb* 中的 *nat* 是方法形参个数(不包括 *this*),将用于指定代码生成时可用的寄存器序号. 因此,将式(2)中第一行的类型变量$'m$ 用 *Ji_mb* 替换后,得到中间语言程序的定义 *Ji_prog*.

编译分为两步,第一步是中间代码生成,用于消除本地变量名,将 *J_prog* 翻译成 *Ji_Prog*,由 $cEpr_1$ 完成;第二步是代码生成,将 *Ji_prog* 翻译成 *dvm_prog*,由 $cEpr_2$ 和 $cEprex_2$ 共同完成. 我们将在下一小节 4.1.2 以及 4.1.3 中分别予以讨论.

对方法进行编译的函数如式(16)所示,表示:给定的一个方法(M,Ts,T,m)和一个编译函数 *f*,编译后的结果为$(M,Ts,T,f\ m)$.

$$\begin{aligned} &cMethod\ f \equiv\\ &\lambda(M,Ts,T,m).(M,Ts,T,f\ m) \end{aligned} \tag{16}$$

对应于源程序方法体 *J_mb* 和中间语言程序方法体 *Ji_mb*,相应的 *f* 函数都是 *lambda* 运算,分别如式(17)和(18)所示.

$$\begin{aligned} &\lambda(pns,body).\\ &(length\ pns, cEpr_1(this\#pns)\ body) \end{aligned} \tag{17}$$

$$\begin{aligned} &\lambda(n,body).\ let\ locals = maxLocals\ body;\\ &idx = 1 + n + locals;\\ &ws = maxWS\ body; regs = idx + ws;\\ &is = cEpr_2\ idx\ body; xt = cEprex_2\ idx\ body\ 0;\\ &moveps = genMoveps\ (n+1)\ regs\\ &in\ (regs, moveps@is@[Return\ idx], xt) \end{aligned} \tag{18}$$

其中,函数 *maxLocals* 归纳计算出方法体中嵌套变量的最大深度,因此,*idx* 是起始可用的寄存器序号,式(19)给出了块语名和捕获异常语句的计算规则. 函数 *maxWS* 归纳得到所需要的最大计算空间,式(20)给出了创建对象、变量访问、条件以及方法调用的计算规则.

$$\begin{aligned} &maxLocals\{vts;e\} = maxLocals\ e + size\ vts\\ &maxLocals(try\ e\ catch(C\ V)\ e') =\\ &max\ (maxLocals\ e)(maxLocals\ e'+1) \end{aligned} \tag{19}$$

$$\begin{aligned} &maxWS\{NewInstance\ n\ C\} = 1\\ &maxWS\ (Var\ i) = 1\\ &maxWS(if\ (e)e_1 else\ e_2) =\\ &max\ (maxWS\ e)\ (\max\ (maxWS\ e_1)(maxWS\ e_2))\\ &maxWS(e \cdot M(es)) =\\ &\max\ (maxWS\ e)\ (maxWS\ es) + 1 \end{aligned} \tag{20}$$

函数 *genMoveps* 生成 Move 指令,将实参值顺序放到序号从 0 开始的寄存器中,如式(21)所示.

$$\begin{aligned} &genMoveps\ 0 regs = [\,]\\ &genMoveps\ (Suc\ m)\ regs =\\ &[Move\ m\ (regs+m)]\ @\ genMoveps\ m\ regs \end{aligned} \tag{21}$$

主体编译函数如式(22)所示,使用运算°组合两步编译函数,完成从$_m$Java 程序到 Micro-Dalvik 虚拟机程序的编译.

$$\begin{aligned} &J2DVM \equiv cPrg_2 \circ cPrg_1\\ &cPrg_1 \equiv cPrg \text{ 式(25)}\\ &cPrg_2 \equiv cPrg \text{ 式 (26)}\\ &cPrg\ f \equiv\ map\ (cClass\ f)\\ &cClass\ f \equiv \lambda(C,D,fs,ms).\\ &(C,D,fs,map(cMethod\ f)\ ms) \end{aligned} \tag{22}$$

4.1.2 中间代码程序 Ji_prog 生成

为了得到本地变量声明序号,需要知道当前本地变量环境,参数 *Vars*::*vname list* 代表这个环境,初始本地变量环境为 *this* 和方法形参名,在式(17)给出.

由$_m$Java 源程序到中间语言程序的转换由函数 $cEpr_1$ 完成,在给定当前变量环境下,计算表达式中出现的变量名的序号. 因此,除了本地变量名被转换成了其对应的序号外,其余与源程序相同. 式(23)给出了编译块表达式语句的方法,其他表达式的语句的编译较为直接,不再赘述.

$$\begin{aligned} &cEpr_1 Vars\ \{vts;e\} = \{\\ &enumerate(size\ Vars)(map(\lambda(vn,t).\ t)vts);\\ &cEpr_1 Vars@\ (map(\lambda(vn,t).\ vn)vts))e\} \end{aligned} \tag{23}$$

4.1.3 目标 Micro-Dalvik 虚拟机代码生成

目标代码的生成包括虚拟机指令的生成 $cEpr_2$ 和异常表项的生成 $cEprex_2$. 由于 Dalvik 这种寄存器架构的虚拟机没有操作数栈,它使用了额外的指令,即 *MoveResult* 指令存放方法的返回值,以及使用 *MoveException* 指令存放捕获到的异常对象. 因此,编译时,方法调用指令之后紧跟一条 *MoveResult* 指令. 在编译异常处理块时,首先添加一条 *MoveException* 指令.

针对每一个 *Ji* 表达式语句,指令生成方法是直接的. 式(24)给出了异常捕获语句的指令生成方法,其中参数 *idx* 表示编译时可用的第一个寄存器序号,初始序号为方法形参个数 +1(this)与方法本地变量个数的和.

$$\begin{aligned} &cEpr_2\ idx\ (try\ e\ catch(C\ i)\ e') =\\ &(let\ try = cEpr_2\ idx\ e;\\ &catch = [MoveException\ idx\ ]\ @\ cEpr_2(idx+1)\ e'\\ &in\ try@\ [Goto\ (int(size\ catch)+1)]@catch) \end{aligned} \tag{24}$$

由于异常表项中的程序指令计数是绝对程序指令计数,而不是如跳转指令中的相对偏移量,所以异常表项生成函数 $cEprex_2$ 包括了一个参数,代表当前的程序指令计数,针对异常捕获语句的异常表项生成如式(25) 所示.

$$\begin{aligned} &cEprex_2\ idx\ (try\ e\ catch(C\ V)\ e')pc = (let\\ &pc' = pc + |cEpr2\ idx\ e|\ in\\ &cEprex2\ idx\ e\ pc@\ cEprex2\ (idx+1)\ e'\ (pc'+2)\\ &@[(pc,pc',C,pc'+1)] \end{aligned} \tag{25}$$

4.2 编译正确性证明

编译正确性在于编译前后两种程序语义的保持. 对于 4.1 节中定义的中间语言程序 *Ji_prog*,我们还未定义其语义,因此,在本节先给出它的大步操作语义;再构造 *J_prog* 和和编译后的 *Ji_prog* 语义保持定理并证明,以及 *Ji_prog* 和编译后的 *dvm_prog* 语义保持定理并证明,最后由这两步的语义保持,证明 *J_prog* 到 *dvm_prog* 程序的编译的正确性.

4.2.1 Ji_prog 的大步操作语义

与源程序 *J_prog* 的状态比较,中间语言程序 *Ji_prog* 由于消除了变量名,状态表示中,本地变量不再是变量名到其值的映射函数,而是一系列值,如式(26)所示.

$$state_1 = heap \times (val\ list) \tag{26}$$

Ji_prog 的大步操作语义的计算规则与 *J_prog* 的大步操作语义规则基本相同. 式(27)给出它的方法调用的语义规则.

$$\frac{\begin{array}{l} P \vdash {}_1[e,s_0] \Rightarrow [addr\ a, s_1];\\ P \vdash {}_1[es,s_1][\Rightarrow][map\ Val\ vs,(h_2,ls_2)];\\ h_2 a = |(C,fs)|; ts = map(the \circ typeof_h)vs;\\ seesMethod\ P\ C\ M\ ts\ T\ (\_,body)\ D;\\ ls_2' = replicate\ (maxLocals\ body);\\ undefined@(Addr\ a)\ \#\ vs; |vs| = |ts|;\\ P \vdash {}_1[body,(h_2,ls_2')] \Rightarrow [e',(h_3,ls_3)] \end{array}}{P \vdash {}_1[e \cdot M(es), s_0] \Rightarrow [e',(h_3,ls_2)]} \tag{27}$$

4.2.2 J_prog 到 Ji_prog 的编译正确性

定理 1

$$\begin{aligned} &wf\_J\_prog\ P; fv\ e \subseteq set\ Vars;\\ &P \vdash [e,(h,l)] \Rightarrow [e',(h',l')];\\ &\forall a \in dom\ l.\ l\ a = ([Vars[\vdash\!\!\rightarrow]ls]\ a)\\ &|Vars| + maxLocals\ e \leqslant |ls|\\ &\exists ls'.\ cPrg_1 P \vdash {}_1[cEpr_1 Vars\ e,(h,ls)]\\ &\Rightarrow [fin_1 e',(h,ls')] \wedge l' \subseteq_m [Vars[\vdash\!\!\rightarrow]ls'] \end{aligned}$$

其中,符号$[\vdash\!\!\rightarrow]$将两个列表中的元素按序号从左到右一一映射,$l' \subseteq_m [Vars[\vdash\!\!\rightarrow]ls']$表示:

$$\forall a \in dom\ l'.\ l'\ a = ([Vars[\vdash\!\!\rightarrow]ls']\ a)$$

定理 1 表示:若$_m$Java 程序是良构的(well-formed),编译前 *J_Prog* 的状态与编译后的 *Ji_prog* 的状态保持.

程序 *P* 是良构的,除式(7)描述的以外,完整的良构性条件描述为:程序 *P* 中定义的所有类是良构的,且类名不重复. 一个类 *C* 是良构的,当且仅当 *C* 中定义的

成员变量名不重复;成员变量的类型是 P 中合法类型;没有同名且同参数的方法;所有方法体是良构的;若 C 不是根类 *Object*,如果 C 的父类是 D,则 D 不可能是 C 的子类,即不存在父子类的循环定义;若子类方法覆盖父类方法,子类方法的返值类型不大于父类型,如式(7)所示. 一个方法体是良构的,当且仅当形参名列表的长度与方法形参类型列表长度相同;形参名没有重复;*this* 不出现在形参名列表中;不存在嵌套块中变量重复声明;不会出现未声明变量的使用.

按照 J_Prog 到 Ji_prog 的翻译:J_prog 中的本地变量名转换为对应的序号,如果 J_prog 的状态由 (h,l) 转换到 (h',l'),Ji_prog 的状态中堆也转换到 h',设本地变量状态转换到 ls',则 l' 和 ls' 之间的保持关系为:对于所有在映射 l' 已赋值的变量($l'\ V$ 的值不等于 None,表示为 $\forall a \in dom\ l'$),其值与 ls' 中对应的值相同,当然,需要满足的前提是 J_prog 和 Ji_prog 程序执行前的状态 l 和 ls 之间也需要满足这个关系;同时,函数 fv 保证 J_prog 程序中出现的变量名都是方法体中声明的变量、*this* 或者方法的形参;ls' 的大小不小于计算 e 时所需的本地变量个数;没有嵌套块中变量的重复声明. 函数 fin_1 表示 $fin_1(Val\ v) = val\ v, fin_1(ThrowEx\ a) = ThrowEx\ a$.

证明　在语义规则 $P \vdash [e,(h,l)] \Rightarrow [e',(h',l')]$ 上进行归纳证明. 我们给出抛出和捕获异常语句的证明. 首先它是一个递归定义的表达式,证明使用了归纳假定;第二,异常处理是最为复杂的语义之一,它涉及在编译生成的异常表项中查找是否有对应的处理,以及向上层调用方法的异常抛出;第三,寄存器架构的虚拟机并不在操作数栈的栈顶存放抛出的异常,而是由虚拟机本身来传递,这是一个与编译到栈架构虚拟机显著不同的地方. 因此,我们给出这个具有典型性和复杂性的证明. 令 $e = try\ e_1\ catch(C\ V)\ e_2$,对于子表达式 e_1 和 e_2,对应的归纳假定(Induction Hypothesis,IH)成立,设为 IH_1 和 IH_2:

$$\mathrm{IH1}:\frac{\begin{array}{l}P \vdash [e_1,(h,l)] \Rightarrow [(ThrowEx\ a),(h_1,l_1)];\\ fv\ e_1 \subseteq set\ Vars; l \subseteq_m [Vars[\vdash\!\!\rightarrow]ls];\\ |Vs| + maxLocals\ e_1 \leqslant |ls|;\end{array}}{\begin{array}{l}\exists ls_1.\ cPrg_1\ P \vdash_1 [cEpr_1 Vars\ e_1,(h,ls)]\\ \Rightarrow [fin_1(ThrowEx\ a),(h_1,ls_1] \wedge l_1 \subseteq_m [Vars[\vdash\!\!\rightarrow]ls_1]\end{array}}$$

$$\mathrm{IH2}:\frac{\begin{array}{l}P \vdash [e_2,(h_1,l_1(V \vdash\!\!\rightarrow Addr\ a))] \Rightarrow [e',(h_2,l_2)];\\ Vars' = Vars@[V]; ls' = ls_1[|Vars| := Addr\ a];\\ fv\ e_2 \subseteq set\ Vars';\\ l_1(V \vdash\!\!\rightarrow Addr\ a) \subseteq_m [Vars'[\vdash\!\!\rightarrow]ls'];\\ |Vars'| + maxLocals\ e_2 \leqslant |ls'|;\end{array}}{\begin{array}{l}\exists ls_2.\ cPrg_1\ P \vdash_1 [cEpr_1 Vars' e_2,(h_1,ls']\\ \Rightarrow [fin_1\ e',(h_2,ls_2)] \wedge l_2 \subseteq_m [Vars'[\vdash\!\!\rightarrow]ls_2]\end{array}}$$

又 J_prog 的 try-catch 的语义定义为:

$$\frac{\begin{array}{l}P \vdash [e_1,(h,l]] \Rightarrow [(ThrowEx\ a,(h_1,l_1))]; h_1 a = \lfloor (D,fs) \rfloor;\\ P \vdash D \leqslant^* C; P \vdash [e_2,(h_1,l_1(V \vdash\!\!\rightarrow Addr\ a))] \Rightarrow [e',(h_2,l_2)];\end{array}}{P \vdash [e,(h,l]] \Rightarrow [e',(h_2,l_2(V := l_1\ V))]}$$

联合 IH1 和 IH2,满足 J_prog 的 try-catch 的语义规则的前提,于是有:

$$P \vdash [e,(h,l]] \Rightarrow [e',(h_2,l_2(V := l_1\ V))]$$

又 J_prog 程序 try-catch 的编译规则为:

$$cEpr_1 Vars\ (try\ e_1\ catch(C\ V)\ e_2) =$$
$$try\ (cEpr_1\ Vars\ e_1)\ catch(C\ size(Vars)),$$
$$(cEpr_1(Vars@[V])\ e_2)$$

而 Ji_prog 的 try-catch 语义规则前提为:

$$P \vdash_1 [e_1,(h,ls] \Rightarrow [(ThrowEx\ a,(h_1,ls_1))];$$
$$h_1 a = |(D,fs)|; P \vdash D \leqslant^* C;$$
$$P \vdash_1 [e_2,(h_1,ls_1(i := Addr\ a))] \Rightarrow [e',(h_2,ls_2)]$$

因此,结合 IH2,ls_2 即是待要证明存在的本地变量列表,即 $\exists ls_2$ 使得:

$$cPrg_1\ P \vdash_1 [cEpr_1\ Vars\ e,(h,ls)] \Rightarrow$$
$$[fin_2\ e',(h_2,ls_2)] \wedge l_2(V := l_1\ V) \subseteq_m [Vars[\vdash\!\!\rightarrow]ls_2]$$

其中,$l_2(V := l_1\ V)$ 是计算 e 之后的本地变量状态.

证毕.

4.2.3　Ji_prog 到 dvm_prog 的编译正确性

定理 2

$$\frac{\begin{array}{l}seesMethod\ P_1\ CM\ Ts\ T\ (n,body)\ C\\ P_1 \vdash_1 [body,(h,ls)] \Rightarrow [e',(h',ls')]\end{array}}{\begin{array}{l}cPrg_2\ P_1 \vdash_1 (None,h,[(Unit,ls,C,M,ts,0)],Unit)\\ \xrightarrow{dvm} (exception\ e',h',[],Unit)\end{array}}$$

定理 2 表示:给定程序 Ji_Prog,其类 C 具有可执行的方法 M,即 $(M,Ts,T,(n,body))$,在状态 (h,ls) 下执行其方法体 *body*,到达新的状态 (h',ls');等价于在机器状态 $(None,h,[(Unit,ls,C,M,ts,0)],Unit)$ 下执行编译后的程序 $cPrg_2P_1$,,则执行后的机器状态中,堆也应该是 h',程序执行完毕,桢栈为空. $exception\ e'$ 为 *None* 或某个产生的异常对象地址.

定理 2 直观地表达了编译前后的语义保持性,但是其表达的是方法体内所有语句全部执行完毕后的语义,不能进行归纳证明. 因此,代替方法体全部执行完毕来表示语义,而是给定正在执行的指令计数,并按照 Ji_Prog 执行其表达式的结果是正常值还是抛出异常两方面分别构造语义保持,从而构造定理 3,如下所示.

定理 3　给定 $P_1 :: Ji\_prog$, $e :: epr_1$, $P \equiv cPrg_2$

$$P_1\ \frac{P_1 \vdash_1 [e,(h,ls)] \Rightarrow [e',(h',ls')]}{\begin{array}{l}(\forall C\ M\ ts\ idx\ pc\ v\ xa\ frs.\ P,C,M,ts,pc \rhd cEpr_2\ idx\ e \Rightarrow\\ (e' = Val\ v \longrightarrow P \vdash (None,h,(xt,ls,\\ C,M,ts,pc)\#frs,retv) \xrightarrow{dvm} (None,h',(xt,ls',\end{array}}$$

$C,M,ts,pc+|cEpr_2\ idx\ e|)\#frs,retv))\wedge$
$(e'=ThrowEx\ (Val(Addr\ xa))\rightarrow$
$(\exists pc_1.\ pc\leqslant pc_1\wedge pc_1\leqslant pc+|cEpr_2\ idx\ e|\wedge$
$\neg\ caught\ P\ pc_1\ h'\ xa\ (cEprex_2\ idx\ e\ pc)\ \wedge$
$P\vdash\ (None,h,(xt,ls,C,M,ts,pc)\#frs,retv)$
$\xrightarrow{dvm}\ lookupH\ P\ xa\ h'((Addr\ xa),xt\ ,ls',$
$C,M,ts,pc_1)\#\ frs\ retv)))$

其中:$P,C,M,ts,pc\triangleright cEpr_2\ idx\ e$ 表示当前栈帧的指针不会超出被编译表达式所对应的指令, pc 指向表达式 e 编译后对应的第一条指令.

证明 在计算规则 $P\vdash_1[e,(h,ls)]\Rightarrow[e',(h',ls')]$ 上进行归纳证明,即对每一个表达式,在其子表达式归纳假定成立的条件下,证明该表达式计算前后的语义是相同的. 在此,我们仍然给出最为复杂的语句之一,异常处理语句,即捕获到抛出异常并作处理的归纳证明过程.

令 $e=try\ e_1\ catch(C\ i)e_2$,

定理的前提条件之一 e 的语义定义为:

$$\frac{\begin{array}{l}P_1\vdash_1[e_1,(h_0,ls_0)]\Rightarrow[ThrowEx\ a,(h_1,ls_1)]\\ h_1a=\lfloor(D,fs)\rfloor;P\vdash D\leqslant^* C;i<|ls_1|\\ P_1\vdash_1[e_2,(h_1,ls_1[i:=Addr\ a])]\Rightarrow[e_2',(h_2,ls_2]\end{array}}{P_1\vdash_1[try\ e_1catch\ (C\ i),(h_0,ls_0)]\Rightarrow[e_2',(h,ls_2)]}$$

上式表明:如果执行 e_1 时系统抛出了异常,状态转换为 (h_1,ls_1);该异常能够被捕获,即异常类型是捕获异常类型的子类型,并且变量的索引号不超出本地变量最大索引号;在对 ls_1 进行更新,即 $(h_1,ls_1[i:=Addr\ a])$ 状态下执行 e_2,计算得到 e_2',到达状态 (h_2,ls_2);则执行 e 后得到 e_2',到达状态为 (h_2,ls_2).

令 $\sigma_0=(None,\ h_0,(xt,ls_0,\ C,M,ts,pc)\#frs,retv)$

其中,pc 满足:$P,C,M,ts,pc\triangleright cEpr_2\ idx\ e$

因此,pc 也满足:$P,C,M,ts,pc\triangleright cEpr_2\ idx\ e_1$

因此,由:

$P_1\vdash_1[e_1,(h_0,ls_0)]\Rightarrow[ThrowEx\ a,(h_1,ls_1)]$,

根据归纳假定得到:$\exists pc_1$,使得:

$pc\leqslant pc_1\wedge pc_1<pc+|cEpr_2\ idx\ e_1|\wedge$
$\neg\ caught\ P\ pc_1\ h_1(cEprex_2\ idx\ e_1\ pc)\wedge P_1\vdash\sigma_0$
$\xrightarrow{dvm}lookupH\ P\ a\ h_1((xt,ls_1,C,M,ts,pc_1)\#frs)\ rtv$

由于此处考虑的是捕获到异常的语义,*lookupHandler* 函数返回一个机器状态,于是有:

$P_1\vdash\sigma_0\xrightarrow{dvm}(None,\ h_1,(xt,ls_1[i:=Addr\ a],$
$C,M,ts,pc_1')\#frs,retv)$

其中,$pc_1'=pc+|cEpr_2\ idx\ e_1|+1$.

于是有:$P,C,M,pc_1'\triangleright cEpr_2\ idx\ e_2$. 令 σ_1 表示状态:

$(None,h_1,(xt,ls_1[i:=Addr\ a],C,M,ts,pc_1')\#frs$

因此,由

$P_1\vdash_1[e_2,(h_1,ls_1[i:=Addr\ a])]\Rightarrow[e_2',(h_2,ls_2]$

根据归纳假定得到:

如果 $e_2'=Val\ v$,则有:

$P\vdash\sigma_1\xrightarrow{dvm}(None,\ h_2,(xt,ls_2,$
$C,M,ts,pc_1'+|cEpr_2(idx+1)\ e_2|)\#frs,retv)$,

$e_2'=ThrowEx\ a$,则存在 pc_2,使得:

$pc_1'\leqslant pc_2\wedge pc_2<pc_1'+|cEpr_2(idx+1)\ e_2|\wedge$
$\neg\ caught\ P\ pc_2\ h_2(cEprex_2\ idx'\ e_2\ pc_1')\wedge$
$P\ \sigma_1\xrightarrow{dvm}lookupH\ P\ a\ h_2$
$((xt,ls_2,C,M,ts,pc_2)\#frs)\ retv$

如果

又由 *try..catch* 的编译规则:

$cEpr_2\ idx\ (try\ e_1catch(C\ i)\ e_2)=let$
$try=cEpr_2\ idx\ e_1;catch=$
$[MoveException\ idx\ ]\ @\ cEpr_2(idx+1)\ e_2$
$in\ try@\ [Goto\ (int\ |catch|+1)]@catch$

,得到:

$pc_1'+|cEpr_2(idx+1)e_2|=$
$pc+|cEpr_2\ idx\ e_1|+1+|cEpr_2(idx+1)\ e_2|=;$
$pc+|cEpr_2\ idx\ e|$

又因为:

$pc_1'\leqslant pc_2,pc_2<pc_1'+|cEpr_2(idx+1)\ e_2|$,

所以有:$pc\leqslant pc_2,pc_2<pc+|cEpr_2\ idx\ e|$.

因此,由机器状态转换的传递性,得到待证目标:假定 $P,C,M,ts,pc\triangleright cEpr_2\ idx\ e$,若

$P_1\vdash_1[try\ e_1\ catch\ (C\ i),(h_0,ls_0)]\Rightarrow[e_2',(h_2,ls_2)]$,则:1)如果 $e_2'=Val\ v$,则有:

$P\vdash\sigma_0\xrightarrow{dvm}(None,\ h_2,(xt,ls_2,$
$C,M,ts,pc+|cEpr_2\ idx\ e|)\#frs,retv)$

2)如果 $e_2'=ThrowEx\ a$,则存在 pc_2,使得:

$pc\leqslant pc_2\wedge pc_2<pc\ +|cEpr_2\ idx\ e|\wedge$
$\neg\ caught\ P\ pc_2\ h_2(cEprex_2\ idx\ e\ pc\ )\wedge P\vdash\sigma_0$
$\xrightarrow{dvm}lookupH\ P\ a\ h_2((xt,ls_2,C,M,ts,pc_2)\#frs)\ rtv$

证毕.

定理 2 通过定理 3 得到证明,即将定理 2 的前提应用到已经得到证明的定理 3 上,从而得到定理 2. 证明过程如下.

证明

由定理 2 的前提:

$seesMethod\ P_1\ C\ M\ Ts\ T\ (n,body)\ C$

可以得到:

$cPrg_2\ P_1,C,M,ts,0\triangleright cEpr_2\ idx\ body$

令 $\sigma_0=(None,h,[(Unit,ls,C,M,ts,0)],Unit)$

联合定理 2 的另一前提:

$P_1 \vdash_1 [body,(h,ls)] \Rightarrow [e',(h',ls')]$

应用到定理 3,得到:

如果 $e' = Val\ v$,则

$$cPrg_2\ P_1 \vdash \sigma_0 \xrightarrow{dvm}$$

$(None,h',[(Unit,ls',C,M,ts,|cEpr_2\ idx\ body|)],Unit)$,因此有:

$$cPrg_2\ P_1 \vdash \sigma_0 \xrightarrow{dvm} (None,h',[],Unit)$$

如果 $e' = ThrowEx\ a$,则存在 pc,使得:

$0 \leqslant pc \wedge pc < |cEpr_2\ idx\ body| \wedge$

$\neg\ caught\ P\ pc\ h'\ (cEprex_2\ idx\ body\ 0) \wedge$

$cPrg_2\ P \vdash \sigma_0 \xrightarrow{dvm}$

$lookupH\ P\ a\ h'[(Unit,ls',C,M,ts,pc)]\ Unit$

因此有:

$$cPrg_2\ P_1 \vdash \sigma_0 \xrightarrow{dvm} (|a|,h',[],Unit)$$

综合两种情况,得到:

$$cPrg_2\ P_1 \vdash \sigma_0 \xrightarrow{dvm} (exception\ e',h',[],Unit)$$

证毕.

4.2.4　J_prog 到 dvm_prog 的编译正确性

已经证明了编译 J_prog 到 Ji_prog 以及编译 Ji_prog 到 dvm_prog 的正确性,因此,J_prog 到 dvm_prog 的编译正确性定理可以构造为:

定理 4

$$\frac{\begin{array}{l} wf\_J\_prog\ P\ seesMethod\ P\ C\ M\ Ts\ T\ (pns,body)\ C \\ P \vdash [body,(h,[this\#pns[\mapsto]vs])] \Rightarrow [e',(h',l')] \\ |vs| = |pns| + 1 \quad |rest| = maxLocals\ body + maxWS\ body \end{array}}{\begin{array}{l} J2DVM\ P \vdash (None,h,[(Unit,rest@vs,C,M,[],0)],Unit) \\ \xrightarrow{dvm} exception\ e',h',[],Unit) \end{array}}$$

组合定理 1 和定理 2,即由源程序到中间程序的翻译正确性,以及中间程序到目标程序的正确性都得到证明后,源程序到目标程序的正确性即建立.

整个编译证明的重点在定理 3 的归纳证明上,它包括 42 个 case 分析,异常捕获是最为复杂、因此最具有代表性的证明 case 之一,我们给出了其证明过程,其他 case 的证明方法类似,在此不作详述.

5　结论和下一步研究

本文在定理证明助手 Isabelle/HOL 的支持下,设计并实现了一个机器验证的编译器,源语言是类 Java 的面向对象语言$_m$Java,目标机器是类 Dalvik 的寄存器架构虚拟机 Micro-Dalvik. 式(1)所定义的的子集考虑到了一个 Java 程序的核心特性,包括类的定义(实例变量和实例方法),对象创建、继承关系、方法覆盖、静态重载、简单类型和引用类型以及异常抛出和捕获机制. 从这种意义上讲,它不是一个假想的语言和目标机器,具有较大实用性. 未被考虑的包机制和访问权限、数组和线程、构造方法、类变量和类方法以及 finally 语句和 finalize 方法等将在本文基础上进行扩展,譬如给式(2)中的类名 cname 加上包名,形成限定名来建模包的概念,可以增加一个类型来描述 Java 语言规范中定义的四种访问权限,加入到类声明和方法声明的定义中,因此式(2)中的′m cdecl 由二元组将变成三元组,′m mdecl 由三元组将变成四元组,这些改变将导致成员变量和成员方法查找的重新定义.

需要指出的是,我们形式化的是顺序 OO 语言,并未对并发特性进行编译验证. 关于并发特性的 OO 语言编译验证,文献[40]在文献[16]的基础上进行了针对 Java 线程的讨论. 文献[41]给出了一个针对 C/C ++ 并发特性的编译证明,这个编译证明相当抽象;同时,作者也指出:距离完整的 C ++ 正确性证明仍然还很漫长,未来他们考虑得到一个具体的操作语义以及编译 C ++ 代码片段. 这些工作对我们将支持并发的 Java 编译到 Dalvik VM 有一定的借鉴意义. 本文未包括的另一个语言特性是数组. 考虑到 Java 在运行时动态创建数组,并在运行时对数组赋值进行类型检查,我们拟在本文的基础上,证明该编译器也保持了类型的可靠性,围绕类型系统,再去完整地分析数组问题.

因此,下一步我们将进一步解决完善这个编译器能够支持的语法特性,并对目标虚拟机 Micro-Dalvik 类型系统的可靠性进行研究和证明.

参考文献

[1] 何炎祥,吴伟. 可信编译理论与关键技术[M]. 北京:科学出版社,2013.
He YX, Wu W. Theory and Key Technology of Trusted Compiler[M]. Beijing: Science Press, 2013. (in Chinese)

[2] Boyle MJ, Resler DR, Winter LV. Do you trust your compiler[J]. Computer, 1999, 32(5): 65-73.

[3] 何炎祥,吴伟,刘陶,李清安,陈勇,胡明昊,刘健博,石谦. 可信编译理论及其核心实现技术:研究综述[J]. 计算机科学与探索,2011,5(1):1 -22.
He YX, Wu W, Liu T, Li QA, Chen Y, Hu MH, Liu JB, Shi Q. Theory and key implementation techniques of trusted compiler: A survey[J]. Journal of Frontiers of Computer Science and Technology, 2011, 5(1): 1 -22. (in Chinese)

[4] Stepney S. High Integrity Compilation: A Case Study[M]. Hatfield: Prentice Hall International (UK) Ltd, 1993.

[5] McCarthy J, Painter J. Correctness of a compiler for arithmetic expression[A]. Proceedings of the Symposium in Applied Mathematics[C]. Rhode Island, USA, 1967. 33 -41.

[6] Plotkin DG. The Origins of structural operational semantics

[J]. The Journal of Logic and Algebraic Programming, 2004,6:3 – 15.
[7] Morris LF. Advice on structuring compilers and proving them correct[A]. Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL′73)[C]. Boston: ACM Press, 1973. 144 – 152.
[8] Zadarnowski P. C lambda calculus & compiler verification [D]. Sydney: University of New South Wales, 2011.
[9] Burstall MR. Proving properties of programs by structural induction[J]. The Computer Journal, 1969, 12(1): 41 – 48.
[10] Nipkow T, Paulson CL, Wenzel M. A Proof for Higher-Order Logic[M]. Berlin: Springer-Verlag, 2010.
[11] Yves B, Pierre C. Interactive Theorem Proving and Program Development: Coq′Art: the Calculus of Inductive Constructions[M]. Berlin: Springer-Verlag, 2004.
[12] Polak W. Compiler Specification and Verification[Z]. California: Stanford University, Lecture Notes in Computer Science: 124, Secaucus: New York, 1981.
[13] Blazy S, Dargaye Z, Leroy X. Formal verification of a C compiler front-end[A]. Proceedings of the 14th International Symposium on Formal Methods[C]. Hamilton: Springer Berlin Heidelberg, 2006. 460 – 475.
[14] Leroy X. A formally verified compiler backend[J]. Journal of Automated Reasoning, 2009, 43(4): 363 – 446.
[15] Leroy X. Formal verification of a realistic compiler[J]. Communications of the ACM, 2009, 52(7): 107 – 115.
[16] Klein G, Nipkow T. A machine-checked model for Java-like language, virtual machine and compiler[J]. ACM Transactions on Programming Languages and Systems, 2006, 28(4): 619 695.
[17] Strecker M. Formal verification of a Java compiler in Isabelle[A]. Proceedings of 18th International Conference on Automated Deduction[C]. New York, USA, 2002. 63 – 67.
[18] Hoare CAR, He J, Sampaio A. Normal form approach to compiler design[J]. Acta Informatic, 1993, 30: 701 – 739.
[19] Sampaio A. An algebraic approach to the design of compilers: 4 (AMAST Series in Computing)[Z]. World Scientific, 1997.
[20] Duran A A, Cavalcanti A, Sampaio A. An algebraic approach to the design of compilers for object-oriented languages[J]. Formal Aspects of Computing, 2010, 22: 489 – 535.
[21] Milner R, Weyhauch R. Proving compiler correctness in a mechanized logic[J]. Machine Intelligence, 1972, 7(3): 51 – 70.
[22] Leroy X. Mechanized semantics for compiler verification [A]. Proceedings of the10th Asian Symposium on Programming Language and System[C]. Kyoto, Japan, 2012. 386 – 388.
[23] Nipkow T, Oheimb D V. Javalight is type-safe: definitely [A]. Proceedings of the 25th ACM Symposium on Principles of Programming Languages[C]. New York, USA, 1998. 161 – 170.
[24] Stark R, Schmid J, Borger E. Java and the Java Virtual Machine Definition, Verification, Validation[M]. New York: Springer, 2001.
[25] Drossopoulou S, Eisenbach S. Describing the semantics of Java and proving type soundness[A]. Formal Syntax and Semantics of a Java[C]. New York: Springer, 1999. 41 – 82.
[26] Syme D. Proving Java type soundness[A]. Formal Syntax and Semantics of Java[C]. New York: Springer, 1999. 83 – 118.
[27] Oheimb DV, Nipkow T. Machine-checking the Java specification: proving type-safety[A]. Formal Syntax and Semantics of a Java[C]. New York: Springer, 1999. 119 – 156.
[28] Oheimb DV. Analyzing Java in Isabelle/HOL: formalization, type safety and Hoare logic[D]. Munich: Technical University of Munich, 2001.
[29] Payet é, Spoto F. An operational semantics of android activities[A]. Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation [C]. New York: ACM Press, 2014. 121 – 132.
[30] Jeon JS, Micinski KK, Foster JS. SymDroid: symbolic execution for Dalvik bytecode[R]. CS-TR-5022, Department of Computer Science, University of Maryland, College Park, 2012.
[31] Erik RW. Formalization and analysis of Dalvik bytecode [J]. Science of Computer Programming-Special Issue on Bytecode, 2012, 92: 25 – 55.
[32] Nipkow T, Klein G. Concrete Semantics: with Isabelle/HOL[M]. Berlin: Springer-Verlag, 2015.
[33] Kahn G. Natural semantics[A]. Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science[C]. London, Britain, 1987. 22 – 39.
[34] J Gosling, B Joy, G Steele, G Bracha. The Java Language Specification[M]. Addison-Wesley, 2005.
[35] Paller G. Dalvik Opcodes[OL]. http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html, 2012.
[36] Android Source Code[OL]. git://android.git.kernel.org/platform/development/pdk/docs/guide/dalvik.jd.
[37] Ehringer D. The Dalvik Virtual Machine Architecture[OL]. http://davidehringer.com/software/android/, 2010.
[38] Security Engineering Research Group. Analysis of Dalvik

virtual machine and class path library[R]. Pakistan:Institute of Management Sciences Peshawar,2009.

[39] 何炎祥,江南,李清安,张军,沈凡凡. 一个机器检测的 Micro-Dalvik 虚拟机模型[J]. 软件学报,2015,26(2):364 - 379.
HE Yan-Xiang, JIANG Nan, LI Qing-An, ZHANG Jun, SHEN Fan-Fan. Machine-checked model for micro-Dalvik virtual machine[J]. Journal of Software,2015,26(2):364 - 379. (in Chinese)

[40] Lochbihler A. Verifying a compiler for Java threads[A]. Programming Languages and Systems[C]. Berlin Heidelberg:Springer,2010. 427 - 447.

[41] M Batty, K Memarian, S Owens, S Sarkar, P Sewell. Clarifying and compiling c/c + + concurrency:from c + + 11 to POWER[A]. Proceedings of the Symposium on Principles of Programming Languages POPL2012[C]. USA, 2012. 509 - 520.

作者简介

江　南　女,1976 年 4 月出生,湖北武汉人. 2003 年毕业于湖北工学院电子与计算机科学系,获硕士学位,其后留校任教,2009 年 1 月至 2009 年 8 月在美国佐治亚理工学院作访问研究,2012 开始在武汉大学计算机学院攻读博士学位,从事可信软件、可信编译的研究.
E-mail:nanjiang@ whu. edu. cn

何炎祥　男,1952 年 1 月出生,湖北应城人. 教授、博士生导师. 1973 年、1986 年和 1999 年分别在武汉大学、美国 Oregon 大学和武汉大学获学士、硕士和博士学位. 现任教于武汉大学计算机学院,主要从事可信软件、分布并行处理和软件工程等方面的研究工作.