软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2015,26(2):364−379 [doi: 10.13328/j.cnki.jos.004787]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一个机器检测的 Micro-Dalvik 虚拟机模型*

何炎祥[1,2], 江 南[1,3], 李清安[1,2], 张 军[1,4], 沈凡凡[1]

[1](武汉大学 计算机学院,湖北 武汉 430072)

[2](软件工程国家重点实验室(武汉大学),湖北 武汉 430072)

[3](湖北工业大学 计算机学院,湖北 武汉 430070)

[4](东华理工大学 软件学院,江西 南昌 330013)

通讯作者: 江南, E-mail: nanjiang@whu.edu.cn

摘 要: 给出了一个寄存器架构的虚拟机模型 Micro-Dalvik,包括虚拟机指令集和虚拟机运行时状态的形式化,并以大步操作语义(big-step operational semantics)的方式给出了指令单步执行的状态转换以及定义在单步执行上的自反传递闭包来表达虚拟机程序的运行时状态转换.最后,以定理的形式描述了语义满足的性质,并得到证明.这个模型的指令集包括了大部分 Dalvik 虚拟机指令,为获得形式语义的清晰化,它在 Dalvik VM 指令集上进行了必要的抽象,对其实质没有改变,因而具有较大的实用性.该形式化模型通过了定理证明助手 Isabelle/HOL 的验证.

关键词: 大步操作语义;形式化验证;定理证明;寄存器架构的虚拟机

中图法分类号: TP303

中文引用格式: 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.软件学报,2015,26(2):364−379. http://www.jos.org.cn/1000-9825/4787.htm

英文引用格式: He YX, Jiang N, Li QA, Zhang J, Shen FF. Machine-Checked model for Micro-Dalvik virtual machine. Ruan Jian Xue Bao/Journal of Software, 2015,26(2):364−379 (in Chinese). http://www.jos.org.cn/1000-9825/4787.htm

Machine-Checked Model for Micro-Dalvik Virtual Machine

HE Yan-Xiang[1,2], JIANG Nan[1,3], LI Qing-An[1,2], ZHANG Jun[1,4], SHEN Fan-Fan[1]

[1](Computer School, Wuhan University, Wuhan 430072, China)

[2](State Key Laboratory of Software Engineering (Wuhan University), Wuhan 430072, China)

[3](Computer School, Hubei University of Technology, Wuhan 430068, China)

[4](Software College, East China Institute of Technology, Nanchang 330013, China)

Abstract: This paper introduces Micro-Dalvik, a formalized Dalvik-like VM model to exhibit core features of the register-based architecture. This formal specification describes the instruction set and runtime state, and the runtime behaviors of the instructions as state transitions based on big-step operational semantics. The Micro-Dalvik VM instruction set is more abstract than comparable Dalvik VM to attain clarity of its semantics, but bears no further simplification of the meaning of instructions. Some properties of Micro-Dalvik VM semantics are stated based on this formal specification and sketch of the proofs is provided. This formalization is implemented in the theorem proof assistant Isabelle/HOL.

Key words: big-step operational semantics; formal verification; theorem proving; register-based VM

虚拟机(virtual machine,简称 VM)是对硬件环境资源的一种软件模拟.从语言实现方面考虑,将高级语言编译到虚拟机上执行是目前流行的做法.根据架构的不同,虚拟机分为栈架构的虚拟机和寄存器架构的虚拟机两大类型.栈架构的虚拟机以 Java 虚拟机(JVM)为典型代表,其默认操作数栈的栈顶元素为指令的操作数,大多数

* 基金项目: 国家自然科学基金(91118003, 61170022)
收稿时间: 2014-07-15; 修改时间: 2014-10-31; 定稿时间: 2014-11-26

指令本身不包括操作数,从而使得其指令集紧凑.寄存器架构的虚拟机以 Android Dalvik VM 为典型代表,相对而言,这种虚拟机必须在指令中显式地指定操作数,从而增加了虚拟机取指的负载[1].此外,一般认为翻译到栈架构指令的编译器更易于实现[1,2].因此,许多虚拟机的实现者们更倾向于栈架构的虚拟机.然而,编译的相对容易以及代码的紧凑带来的代价是性能的丢失.研究结果表明:在标准测试中,寄存器架构的虚拟机总体上执行时间比栈架构的要少 32.3%[1,3];同时,寄存器架构的虚拟机更接近于真实的处理器[2].因此,我们选取寄存器架构的虚拟机作为研究对象.

语义用来指明程序语言的含义,通常使用自然语言描述.形式语义为理解程序的行为以及对程序的行为进行论证提供了数学基础[4].然而,形式语义目前尚未得到普遍认同的标准描述符号以及描述方法[5].尽管如此,在程序语言语义研究者们多年的努力下,结构化的操作语义(structural operational semantics)以及自然语义(natural semantics)已成为研究者们描述语言语义的常用方法[6−13].这两种语义既有指称语义的数学严谨性,又具有传统操作语义的直观性.结构化的操作语义和自然语义分别也被称为小步操作语义(small-step operational semantics)和大步操作语义(big-step operational semantics).从其名称可以看出:前者描述了指令执行的中间状态,而后者关心指令完全执行后的状态[14−17].从某种意义上看,大步操作语义与指称语义非常接近.虚拟机的形式模型作为编译的目标机器,是我们整个可信编译研究工作的一个重要组成部分.围绕可信编译的研究,我们更关注于语句完全执行的效果(本文暂不考虑程序并发执行的编译正确性),因此,本文对虚拟机语言语义的形式化是以大步操作语义来展开研究的.

程序语言形式语义的研究开始针对的只是实验性质的简单语言(有人称为 toy language).定理证明器(theorem prover)和定理证明助手(theorem proof assistant)的出现,如 LCF theorem prover,Isabelle/HOL,Coq 等,使得形式语义的研究开始朝着实用方向迈进,并具有更高的可靠性[18,19].在 Isabelle/HOL 中进行语义的形式化,可以视为函数式编程与逻辑的融合.其函数式编程非常类似于 Haskell,而其逻辑推理器接受任何符合自然演绎逻辑的规则,并使用这些规则自动地进行证明搜索.我们使用的就是定理证明助手 Isabelle/HOL.

栈架构虚拟机的形式化研究早于寄存器架构的虚拟机研究,发表的研究成果也较多.文献[20−25]从不同角度,使用定理证明工具或者手工给出了 JVM 的形式化模型.近几年,Android 平台以及其虚拟机的形式化研究也开始出现,如文献[26−28].其中,文献[20]给出了一种类似 Java 的源语言到类似 JVM 栈架构虚拟机指令的编译验证,其特点在于其源语言、虚拟机以及编译器设计为统一的模型;文献[26]给出了 Android 应用程序的操作语义,其实现使用的是作者自己开发的符号执行器(symbolic executor),但至文章发表时,并未完全实现;文献[27]使用 OCaml 代码给出了 Dalvik 比较清晰的语义模型,所使用的符号和方法与本文的形式化有较大差别,在接下来的讨论中,我们将其与本文的形式化进行了对比.

基于以上分析,我们以 Android 平台寄存器架构的虚拟机 Dalvik 为原型,使用 Isabelle/HOL2009 构建了一个形式化模型 Micro-Dalvik.它包括较为完整的虚拟机指令集和运行时状态的形式化,以大步操作语义的方式给出了该虚拟机的状态转换,最后陈述并证明了该形式化语义所满足的定理.由于寄存器机没有操作数栈,在这个模型中,除了所有的操作数都必须显式指定其存放的寄存器外,栈机中默认放在操作数栈的栈顶的返回值和抛出的异常都必须有相应的存放位置;并且,因为异常机制包括抛出异常和处理异常两个方面,所以使该问题更加复杂化.在接下来的讨论中,将解释我们这个模型是如何处理这个问题的.

本文第 1 节首先将 JVM 和 Dalvik VM 指令进行对比和分析,然后给出 Micro-Dalvik 指令集及程序的定义.第 2 节首先对 Micro-Dalvik 状态进行形式化,然后定义每条指令的语义函数,即,单步执行的状态转换以及定义在单步执行上的自反传递闭包来表达虚拟机程序的状态转换;最后,以定理的形式描述语义满足的性质并得到证明.第 3 节是总结与下一步的工作.附录 A 给出一个 Java 源程序实例.附录 B 给出这个实例转换到本文讨论的形式化模型 Micro-Dalvik VM 所对应的程序.作为比较,附录 C 给出这个实例中的方法转换到 Android 平台的 Dalvik VM 指令.

1 Micro-Dalvik 指令及程序

1.1 JVM和Dalvik VM指令集比较

JVM 和 Dalvik VM 分别是栈架构和寄存器架构的虚拟机的典型代表.它们的指令操作码从 00H 到 ffH,实际指令数分别大约为 202 和 218.Dalvik VM 指令由 Android 平台的 dx 工具对 JVM 指令进行翻译转换得到,大多数 JVM 指令在 Dalvik VM 指令中都有对应的形式.表 1 对大部分 JVM 与 Dalvik VM 指令进行了对比(未包括数组相关指令以及同步指令).

Table 1 Comparison between JVM and Dalvik VM Instructions

表 1 JVM 与 Dalvik VM 指令集对比

编号	含义	JVM 指令	Dalvik 指令
1	常量操作	*aconst_null* *iconst_*(*m*1|0|1|2|3|4|5) (*l*|*d*)*const_*(0|1) *fconst_*(0|1|2) *bipush byte*|*sipush b*1, *b*2 *ldc index* *ldc_w*|*ldc*2_*w index*1, *index*2	*const*/4 *vA*, *#+B* *const*/16 *vAA*, *#+BBBB* *const vAA*, *#+BBBBBBBB* *const*/*high*16 *vAA*, *#+BBBB*0000 *const-wide*/16 *vAA*, *#+BBBB* *const-wide*/32 *vAA*, *#+BBBBBBBB* *const-wide vAA*, *#+BBBBBBBBBBBBBBBB* *const-wide*/*high*16 *vAA*, *#+BBBB*000000000000 *const-string vAA*, *string@BBBB* *const-class vAA*, *type@BBBB*
2	取成员变量值 成员变量赋值	*getfield index*1, *index*2 *putfield index*1, *index*2	*iget*[*-wide*|*object*|*boolean*|*byte*|*char*|*short*] *vA*, *vB*, *field@CCCC* *iput*[*-wide*|*object*|*boolean*|*byte*|*char*|*short*] *vA*, *vB*, *field@CCCC*
3	算术指令	(*i*|*l*|*f*|*d*)(*add*|*sub*|*mul*|*div*|*rem*)	(*add*|*sub*|*mul*|*div*|*rem*)-(*int*|*long*|*float*|*double*) *vAA*, *vBB*, *vCC* (*add*|*sub*|*mul*|*div*|*rem*)-(*int*|*long*|*float*|*double*)/2*addr vA*, *vB*
4	比较指令	*lcmp* (*f*|*d*)*cmp*(*l*|*g*) *if*(*eq*|*ne*|*lt*|*ge*|*gt*|*le*) *b*1, *b*2 *if_icmp*(*eq*|*ne*|*lt*|*ge*|*gt*|*le*) *b*1, *b*2	*cmpl vAA*, *vBB*, *vCC* *cmp*(*l*|*g*)-(*float*|*double*) *vAA*, *vBB*, *vCC* *if*-(*eqz*|*nez*|*ltz*|*gez*|*gtz*|*lez*) *vAA*, *+BBBB* *if*-(*eq*|*ne*|*lt*|*ge*|*gt*|*le*) *vA*, *vB*, *+CCCC*
5	跳转指令	goto *b*1, *b*2	*goto +AA*, *goto*/16 *+AAAA*, *goto*/32 *+AAAAAAAA*
6	返回指令	(*i*|*f*|*d*|*a*) *return* *return*	*return* [*-wide*|*-object*] *vAA* *return-void*
7	方法调用	*invokevirtual index*1, *index*2	*invoke-virtual*[*/range*] ...
8	创建对象	new *index*1, *index*2	*new-instance vAA*, *type@BBBB*
9	抛出异常	*athrow*	*throw vAA*
10	造型	*checkcast index*1, *index*2	*check-cast vAA*, *type@BBBB*
11	–	(*i*|*l*|*f*|*d*|*a*)(*load*|*store*)_(0|1|2|3) (*i*|*l*|*f*|*d*|*a*)(*load*|*store*) *index*	*move*[*-wide*] *vA*, *vB*　　*move*[*-wide*]/*from*16 *vA*, *vBBBB* *move*[*-wide*]/16 *vAAAA*, *BBBB*　　*move-object vA*, *vB* *move-object*/*from*16 *vAA*, *vBBBB*　　*move-object*/16 *vAAAA*, *vBBBB*
12	–	–	*move*-(*result*|*result-wide*|*result-object*) *vAA*
13	–	–	*move-exception vAA*

从表 1 可以看出:架构的不同使得这两种虚拟机指令集的设计存在明显的不同,但也有共同之处.参照 JVM 规范[29]和文献[30]以及 Dalvik VM 的实现[31],比较如下:

- 首先,由于 Dalvik VM 没有操作数栈,它使用了额外的指令,即,*move-result* 指令存放方法的返回值,使用 *move-exception* 指令存放捕获到的异常对象.*move-result* 指令用在需要方法的返回值时,紧跟方法调用指令;而 *move-exception* 出现在异常处理块的第 1 句.此外,对于所有的 *load* 和 *store* 及栈上的操作指令,Dalik-VM 均无对应的形式,而是具有一系列的寄存器间传递数据的 *move* 指令.见表 1 中编号第 11 行~第 13 行的指令.
- 第二,两种架构的指令集都使将操作数类型编码在操作码中.比如,表 1 中编号第 3 行的 JVM 指令 *iadd*,*ladd*,*fadd*,*dadd* 以及 Dalvik 指令 *add-int*,*add-long*,*add-float*,*add-double* 分别对 *int*,*long*,*float*,*double* 类型的操作数进行算术加运算.编号第 4 行中的 JVM 指令 *lcmp*,*fcmpl*,*dcmpl* 和 Dalvik 指令 *cmp-long*,*cmpl-float*,*cmpl-double* 分别对 *long*,*float* 和 *double* 类型的操作数进行比较.
- 第三,许多 Dalvik VM 指令的不同仅仅在于其用于存放操作数所使用寄存器范围的不同.例如,表 1 中

编号第 11 行中的指令,*move vA,vB*,*move*/*from*16 *vAA*,*vBBBB*,*move*/16 *vAAAA*,*vBBBB* 都是用来在寄存器间传递非对象的数据,*move vA,vB* 的源和目的寄存器编号占 4 位;*move*/*from*16 *vAA*,*vBBBB* 的源寄存器编号占 16 位,目的寄存器编号占 8 位;而 *move*/16 *vAAAA*,*vBBBB* 的源寄存器和目的寄存器编号均占 16 位.

- 最后,方法调用指令在 Dalvik VM 的实现上区分了参数个数,*invoke-virtual* 和 *invoke-virtual*/*range* 分别对应参数个数小于 5 和大于或等于 5 的方法调用.

1.2 Micro-Dalvik虚拟机指令集及程序

1.2.1 Micro-Dalvik 虚拟机指令集

基于上节对虚拟机指令的分析,我们对 Dalvik VM 指令进行了形式化.形式化时,在充分保证指令集具有实用性的前提下,进行了必要的抽象.文献[20,27]均使用了这类抽象方法,这样的抽象并不在实质上改变 Dalvik VM 指令的含义,而同时获得指令紧凑性,使得接下来指令的语义尽可能地清晰分明.

表 2 列出了本文讨论的 Micro-Dalvik 虚拟机指令集,主要进行了两方面的抽象.

Table 2 Instruction set of Micro-Dalvik VM

表 2 Micro-Dalvik 虚拟机指令集

Micro-Dalvik 指令	含义
Const nat val	将常量值放到寄存器 *nat* 中
Iget nat natobj vname cname	将指定成员变量的值放到寄存器 *nat* 中
Iput nat natobj vname cname	将寄存器 *nat* 中的值赋给指定的成员变量
AriBinop aribinop natdest natsrc natsrc	算术运算,结果存到寄存器 *natdest*
Cmp natdest natsrc natsrc	比较,结果(0:相等;1:大于,–1:小于)存到寄存器 *natdest* 中
Ifop ifop nat branchoffset	与 0 进行比较,如果比较结果为真,则以指定的偏移进行跳转
Goto branchoffset	以指定的偏移进行跳转
ReturnVoid	返回空值
Return nat	返回指定寄存器中的值
Invoke "(*nat list*)" *mname*	以指定寄存器列表中的值为实参,调用方法 *mname*(参数个数<5)
InvokRange nat nat mname	以指定首、末寄存器中的值为实参,调用方法 *mname*(参数个数≥5)
NewInstance nat cname	创建指定类型的对象,存到寄存器 *nat* 中
Throw nat	抛出寄存器 *nat* 中的异常
CheckCast nat cname	将指定寄存器 *nat* 中的对象引用造型为指定的类型 *cname*
Move nat nat	将一个寄存器中的值赋值到另一寄存器
MoveResult nat	将方法调用的返回值存到寄存器 *nat* 中
MoveException nat	将捕获到的异常放到寄存器 *nat* 中

对于那些只是操作数类型不同的指令,只对应一条指令.比如,对 *int*,*long*,*float*,*double* 这 4 种类型的数据进行算术加、减、乘、除、取余运算的 20 条指令,我们定义了一条指令 *AriBinop aribinop natdest natsrc natsrc*,其中,*aribinop* 表示这 5 种算术运算,*natdest* 和 *natsrc* 分别表示目的寄存器和源寄存器.类似地,对成员变量进行取值的 7 条指令,我们仅定义了一条 *Iget nat natobj vname cnam*e 指令.

对于那些只是用于存放操作数所使用寄存器范围不同的指令,只对应一条指令.譬如,对于 *move* vA,vB,*move*/*from*16 *vAA*,*vBBBB*,*move*/16 *vAAAA*,*vBBBB*,结合上述对操作数类型的抽象,Dalvik VM 的 9 条用于寄存器间传递数据的 *Move* 指令,我们定义了一条指令,即 *Move nat nat*,实现源寄存器到目的寄存器之间的数据传递.

文献[27]没有对异常进行形式化,因而不包括 *MoveException* 指令;对于方法的返值,它使用了一个特殊的寄存器 r_{ret},来存放,因此它没有一个单独的 *MoveResult* 指令.

1.2.2 Micro-Dalvik 虚拟机程序

对于 Micro-Dalvik 虚拟机程序的形式化,我们采取了文献[20]中统一模型的方法,即源语言程序和目标机器程序共用一个形式化定义,区别只在于方法体的不同.公式(1)对程序的定义是这个统一模型的基础.其中,$'m$ 是类型参数,用于指定是源程序的方法体还是目标程序的方法体.

$$\left.\begin{array}{l} types\ \ 'm\ prog = \ 'm\ cdecl\ list \\ \quad\quad 'm\ cdecl = cname \times\ 'm\ class \\ \quad\quad 'm\ class = cname \times vdecl\ list \times\ 'm\ mdecl\ list \\ \quad\quad vdecl = vname \times ty \\ \quad\quad 'm\ mdecl = mname \times ty\ list \times ty \times\ 'm \end{array}\right\} \tag{1}$$

公式(1)表明:程序是类声明 *cdecl* 的列表;类声明是类名 *cname* 和类定义 *class* 的二元组;类定义是其超类名、成员变量列表 *vdecl list* 和方法声明列表 *mdecl list* 的三元组;变量声明是变量名 *vname* 和类型 *ty* 的二元组;方法声明是方法名 *mname*、参数类型列表 *ty list*、返回值类型 *ty* 以及方法体*'m* 的四元组.

因此,如果将类型参数*'m* 用表示 Dalvik VM 方法体的类型来取代,就可以定义出 Dalvik VM 程序.Dalvik 虚拟机上运行的程序是 Java 字节码经 dx 工具转换后的代码,这种字节码相对于 Java 字节码而言,其执行环境变化较大.按照 Dalvik 虚拟机的实现,我们将 Micro-Dalvik 虚拟机程序形式化为公式(2):

$$\left.\begin{array}{l} types\ dvm\_prog = dvm\_mb\ prog \\ \quad\quad dvm\_mb = nat \times instr\ list \times ex\_table \\ \quad\quad ex\_table = ex\_entry\ list \\ \quad\quad ex\_entry = pc \times pc \times cname \times pc \end{array}\right\} \tag{2}$$

其中,Micro-Dalvik VM 的程序即是将*'m prog* 中的*'m* 使用 Micro-Dalvik VM 方法体 *dvm_mb* 替换后的结果. Micro-Dalvik VM 方法体由寄存器个数 *nat*(不包括实参个数)、虚拟机指令列表 *instr list* 和异常表 *ex_table* 所组成的三元组,其中,异常表是异常项 *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.

2 Micro-Dalvik VM 语言的语义

2.1 程序状态

McCarthy 提出语义就是程序运行时作用在其状态向量上的效果[32].状态向量对于虚拟机而言,就是运行时内存区域上的数据.在 Dalvik VM 实现中,主要运行时内存区域上的数据包括寄存器中存放的值以及堆中对象所具有的各成员变量的取值,这两个区域在本文中分别称为运行时栈和堆.每个方法对应一个桢(以下称为栈桢),存储在运行时栈中.

Dalvik VM 的运行时栈类似于 JVM 的 Java VM 栈.但是,Java VM 栈中的栈桢包括本地变量和操作数栈,操作数栈除了作为指令操作数的默认存放位置以外,还用来存放被调用方法的返值以及抛出的异常.Dalvik VM 运行时栈中的栈桢不包括操作数栈,调用方法时,Dalvik 虚拟机向运行时栈压入两个栈桢,除了用于方法本身的普通栈桢以外,还包括一个中断栈桢(break frame),用来存放方法调用的返回信息[33].中断栈桢也用来存放产生的异常信息,用于异常处理.

基于以上分析,我们将 Micro-Dalvik VM 程序的状态形式化为公式(3):

$$\left.\begin{array}{l} frame = xcptval \times val\ list \times cname \times mname \times ty\ list \times pc \\ dvm\_state = val\ option \times heap \times frame\ list \times retval \end{array}\right\} \tag{3}$$

公式(3)表示的是:Micro-Dalvik VM 状态由是否产生异常、堆、栈桢列表和方法返回值组成,如果有异常产生,则其中的 *val option* 的值为异常对象的地址(*Addr a*);否则,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,其中的 *xcptval* 的值为异常对象的地址.堆的定义沿用了文献[20]的形式,如公式(4)所示,其中,堆表示为内存地址到对象的映射,对象是对象类名和成员变量取值映射函数的二元组,成员变量取值映射函数是成员变量名和定义该成员变量的类的二元组到对应值的映射.

$$\left.\begin{array}{l} heap = nat \rightharpoonup obj \\ obj = cname \times fields \\ fields = vname \times cname \rightharpoonup val \end{array}\right\} \tag{4}$$

文献[27]中将状态由调用栈 C'、路径条件 Φ'、堆 H' 以及静态成员变量 S' 组成,即,$\Sigma=\langle C',\Phi',H',S'\rangle$.其中,堆的形式化是类似的,然而,堆中内址地址映射的对象包括了数组.调用栈中的栈桢由程序计数、方法体以及寄存器中对应变量的取值组成;路径条件记录了运行时的每个条件分支.与公式(3)相比,由于它使用一个特殊的寄存器 r_{ret} 存放返回值,因而没有公式(3)中与 *retval* 对应的部分.按照文中所述,我们理解这个特殊寄存器的作用应该等价于 *retval*.然而在虚拟机语义形式化中,寄存器是与栈桢相关联的,而这个特殊的寄存器显然没有呈现这种特征,因此我们认为,公式(3)在表达语义上比文献[27]更直观.此外,文献[27]没有考虑异常,因此也没有公式(3)中与 *val option* 对应的部分.

2.2 Micro-Dalvik VM的状态转换

以大步操作语义的观点来看,语义是程序、初始状态和终止状态之间的关系,而关系是元组的集合,因此,如果将组成程序的一条语句 c 在状态 s 下经运行后,到达一个状态 s' 作为一个三元组(c,s,s'),程序的语义就是这些元组的集合,称每条语句对应的三元组为一条语义规则[9].对于本文讨论的 Micro-Dalvik VM,每条指令在执行时的状态转换构成一个三元组,所有这些三元组构成 Micro-Dalvik VM 程序指令的语义规则集.

为了更直观地在 Isabelle/HOL 中表示虚拟机的语义,将单条指令的执行状态转换定义为一个函数:

$$run::dvm\_prog \Rightarrow dvm\_state \Rightarrow dvm\_state.$$

该函数取得当前程序计数对应的指令,然后调用函数 *run_instr*::[*instr*,*dvm_prog*,*heap*,*xcptval*,*val list*,*cname*,*mname*,*ty list*,*pc*,*frame list*,*retval*]⇒*dvm_state* 得到该条指令执行后的状态;如果有异常产生,则将该异常对象作为参数,再调用函数 *lookupHandler*::*dvm_prog*⇒*val*⇒*heap*⇒*frame list*⇒*retval*⇒*dvm_state* 在异常表中查找是否有匹配的异常处理块.根据查找的结果,程序继续正常执行,或者将异常向外抛出到调用的方法进行查找,直至最外层方法.如果仍然没有找到匹配的异常处理块,则程序终止于一个异常状态.

函数 *run_instr* 是表达 Micro-Dalvik VM 语义的关键,它给出每条指令的状态转换,其定义如图 1 所示.

Const 指令状态转换是将指定寄存器的值更新为常量操作数;*Move* 指令将目的寄存器的值更新为源寄存器的值.抛出异常、算术比较以及 *MoveResult*,*MoveException* 等指令的定义非常直接,在此不作详述.

New 指令的状态转换使用了 *newAddr* 和 *blank* 函数[20].*newAddr* 在堆中为对象分配空间,若分配失败,则返回值为 *None*;否则,返回分配的地址.对象可能从父类继承成员变量,因此,*blank* 函数首先对指定的对象类沿着定义该类的类层次关系查找到它具有的所有成员变量,这些成员变量表示为二元组的列表,该二元组的类型为 $(vname\times cname)\times ty$,子类定义的成员变量在前,父类定义的成员变量在后,然后对它们赋初值.因此,*Iget* 和 *Iput* 指令都具有一个额外的操作数,用来指明定义该成员变量的类.这两个指令先获得指定的对象,按公式(4)中对象的定义,*Iget* 获得对象的指定成员变量名和指定定义该成员变量的类所映射的值,*Iput* 更新指定成员变量名和指定定义该成员变量的类到其值的映射.因此,对于父类和子类具有同名的成员变量,*Iget* 和 *Iput* 视为不同的成员变量进行取值和赋值.

方法调用的状态转换最为复杂.*Invoke ns M'* 的语义定义如下:

- 首先,获得参数列表的第 1 个元素,按照 JVM 字节码翻译到 Dalvik VM 字节码的翻译方法,该元素代表调用方法的对象,由此得到对象的类型为 C'.
- 调用 *regFetch reg* (*tl ns*)函数获得除第 1 个实参外的所有实参的值列表,使用 $map(the\circ typeof_h)vs$ 得到方法的形参类型,然后,*method P C' M' Ts'* 查找到形参为 *Ts'* 的方法 *M'* 的描述为$(D,T,regs_0,ins,xt)$.
- 接着创建一个大小为 $regs_0$ 的列表,用来存放新的栈桢的本地变量,并将实参值复制到这个列表的末尾,这个新的列表即新栈桢寄存值的列表.
- 最后,将新的栈桢放在原栈桢列表的首部,作为当前运行栈桢.

$$run\_instr(Const\ n)\ P\ h\ xcptv\ reg\ C\ M\ Ts\ pc\ frs\ retv = (None, h, (xcptv, reg[n := v], C, M, Ts, pc+1)\#frs, retv)$$

$$run\_instr(Iget\ n\ obj\ F\ C')\ P\ h\ xcptv\ reg\ C\ M\ Ts\ pc\ frs\ retv = (let\ v = reg!obj; xp' = \text{if}\ v = Null\ \text{then}\ \lfloor NullPtrExcp \rfloor \text{ else } None; (D, fs) = the(h(the\_Addr\ v)); pc' = \text{case}\ xp'\ \text{of}\ None \Rightarrow pc+1 \mid \lfloor addr \rfloor \Rightarrow pc \text{ in } (xp', h, (xcptv, reg[n := (the(fs(F, C')))], C, M, Ts, pc')\#frs, retv))$$

$$run\_instr(Iput\ n\ obj\ F\ C')\ P\ h\ xcptv\ reg\ C\ M\ Ts\ pc\ frs\ retv = (let\ v = reg!obj; xp' = \text{if}\ v = Null\ \text{then}\ \lfloor NullPtrExcp \rfloor \text{ else } None; a = the\_Addr\ v; (D, fs) = the(h\ a); h' = h(a \mapsto (D, fs((F, C') \mapsto reg!n))); pc' = \text{case}\ xp'\ \text{of}\ None \Rightarrow pc+1 \mid \lfloor addr \rfloor \Rightarrow pc \text{ in } (xp', h, (xcptv, reg, C, M, Ts, pc')\#frs, retv))$$

$$run\_instr(AriBinop\ ariop\ dest\ src_1\ src_2)\ P\ h\ xcptv\ reg\ C\ M\ Ts\ pc\ frs\ retv = (let\ i_1 = the\_Intg(reg!src_1); i_2 = the\_Intg(reg!src_2); result = (\text{case}\ op\ \text{of}\ Add \Rightarrow (i_1 + i_2) \mid Sub \Rightarrow (i_1 - i_2) \mid Mul \Rightarrow (i_1 * i_2) \mid Div \Rightarrow (i_1\ div\ i_2) \mid Rem \Rightarrow (i_1 \bmod i_2) \text{ in } (None, h, (xcptv, reg[dest := (Intg\ result)], C, M, Ts, pc+1)\#frs, retv))$$

$$run\_instr(Cmp\ dest\ src_1\ src_2)\ P\ h\ xcptv\ reg\ C\ M\ Ts\ pc\ frs\ retv = (\text{let}\ srcOprand_1 = reg!src_1; srcOprand_2 = reg!src_2; b_1 = \text{if}\ srcOprand_1 = (Bool\ \text{True})\ \text{then 1 else if}\ srcOprand_1 = (Bool\ \text{False})\ \text{then 0 else}\ -1; b_2 = \text{if}\ srcOprand_2 = (Bool\ \text{True})\ \text{then 1 else if}\ srcOprand_2 = (Bool\ \text{False})\ \text{then 0 else}\ -1; v_1 = \text{if}\ b_1 = -1\ \text{then}\ the\_Intg\ srcOprand_1\ \text{else}\ b_1; v_2 = \text{if}\ b_2 = -1\ \text{then}\ the\_Intg\ srcOprand_2\ \text{else}\ b_2; result = (\text{case}\ v_1 = v_2\ \text{of True} \Rightarrow 0 \mid \text{False} \Rightarrow (\text{case}\ v_1 > v_2\ \text{of True} \Rightarrow -1 \mid \text{False} \Rightarrow 1)) \text{ in } (None, h, (xcptv, reg[dest := (Intg\ result)], C, M, Ts, pc+1)\#frs, retv))$$

$$run\_instr(Ifop\ cmpop\ n\ offset)\ P\ h\ xcptv\ reg\ C\ M\ Ts\ pc\ frs\ retv = (let\ operand = reg!n; newpc = nat(int\ pc + offset); newpc' = pc+1; pc' = (\text{case}\ cmpop\ \text{of}\ IfEqz \Rightarrow \text{if}\ operand = 0\ \text{then}\ newpc\ \text{else}\ newpc'; \mid IfNez \Rightarrow \text{if}\ operand \neq 0\ \text{then}\ newpc\ \text{else}\ newpc' \mid IfLtz \Rightarrow \text{if}\ operand < 0\ \text{then}\ newpc\ \text{else}\ newpc'; \mid IfGez \Rightarrow \text{if}\ operand \geqslant 0\ \text{then}\ newpc\ \text{else}\ newpc' \mid IfGtz \Rightarrow \text{if}\ operand > 0\ \text{then}\ newpc\ \text{else}\ newpc'; \mid IfLez \Rightarrow \text{if}\ operand \leqslant 0\ \text{then}\ newpc\ \text{else}\ newpc' \text{ in } (None, h, (xcptv, reg, C, M, Ts, pc')\#frs, retv))$$

$$run\_instr(Goto\ offset)\ P\ h\ xcptv\ reg\ C\ M\ Ts\ pc\ frs\ retv = (None, h, (xcptv, reg, C, M, Ts, nat(int\ pc + offset))\#frs, retv)$$

$$run\_instr(ReturnVoid)\ P\ h\ xcptv\ reg\ C\ M\ Ts\ pc\ frs\ retv = (\text{if}\ frs = [\cdot]\ \text{then}\ (None, h, [\cdot], Unit) \text{ else let } (xcpt', reg', C', M', Ts', pc') = hd\ frs \text{ in } (None, h, (xcpt', reg', C', M', Ts', pc'+1)\#tl\ frs, retv))$$

$$run\_instr(Return\ n)\ P\ h\ xcptv\ reg\ C\ M\ Ts\ pc\ frs\ retv = (\text{if}\ frs = [\cdot]\ \text{then}\ (None, h, [\cdot], Unit) \text{ else let}(xcpt', reg', C', M', Ts', pc') = hd\ frs \text{ in } (None, h, (xcpt', reg', C', M', Ts', pc'+1)\#tl\ frs, reg!n))$$

$$run\_instr(Invoke\ ns\ M')\ P\ h\ xcptv\ reg\ C\ M\ Ts\ pc\ frs\ retv = (let\ n_{obj} = hd\ ns; v = reg!n_{obj}; n = length\ ns; xp' = \text{if}\ v = Null\ \text{then}\ \lfloor NullPtrExcp \rfloor \text{ else } None; C' = fst(the(h(the\_Addr\ v))); vs = regFetch\ reg(tl\ ns); Ts' = map\ (the \circ typeof_h)\ vs; (D, T, regs_0, ins, xt) = method\ P\ C'\ M'\ Ts'; reg' = replicate\ regs_0\ undefined; reg' = reg'@(v\#vs); f' = (Unit, reg', D, M', Ts', 0) \text{ in } (xp', h, f'\#(xcptv, reg, C, M, Ts, pc)\#frs, retv))$$

$$run\_instr(Invoke\ n_s\ n_e\ M')\ P\ h\ xcptv\ reg\ C\ M\ Ts\ pc\ frs\ retv = (let\ v = reg!n_s; n = n_e - n_s + 1; xp' = \text{if}\ v = Null\ \text{then}\ \lfloor NullPtrExcp \rfloor \text{ else } None; C' = fst(the(h(the\_Addr\ v))); ns = upt\ n_s(n_e+1); (vs = regFetch\ reg\ (tl\ ns); Ts' = map\ (the \circ typeof_h)\ vs; (D, T, regs_0, ins, xt) = method\ P\ C'\ M'\ Ts'; reg' = replicate\ regs_0\ undefined; reg' = reg'@(v\#vs); f' = (Unit, reg', D, M', Ts', 0) \text{ in } (xp', h, f'\#(xcptv, reg, C, M, Ts, pc)\#frs, retv))$$

$$run\_instr(NewIntance\ n\ C')\ P\ h\ xcptv\ reg\ C\ M\ Ts\ pc\ frs\ retv = (\text{case}\ newAddr\ h\ \text{of}\ None \Rightarrow (\lfloor OutOfMemoryExcp \rfloor, h, (xcptv, reg, C, M, Ts, pc)\#frs, retv) \mid \lfloor a \rfloor \Rightarrow (None, h(a \mapsto blank\ P\ C'), (xcptv, reg[n := Addr\ a], C, M, Ts, pc+1)\#frs, retv))$$

$$run\_instr(Throw\ n)\ P\ h\ xcptv\ reg\ C\ M\ Ts\ pc\ frs\ retv = (let\ xp' = \text{if}\ reg!n = Null\ \text{then}\ \lfloor NullPtrExcp \rfloor \text{ else } \lfloor the\_Addr(reg!n) \rfloor \text{ in } (xp', h, (xcptv, reg, C, M, Ts, pc)\#frs, retv))$$

$$run\_instr(CheckCast\ n\ C')\ P\ h\ xcptv\ reg\ C\ M\ Ts\ pc\ frs\ retv = (let\ v = reg!n; xp' = \text{if}\ \sim castOK\ P\ C'\ h\ v\ \text{then}\ \lfloor ClassCastExcp \rfloor \text{ else } None; pc' = \text{case}\ xp'\ \text{of}\ None \Rightarrow pc+1 \mid \lfloor addr \rfloor \Rightarrow pc \text{ in } (xp', h, (xcptv, reg, C, M, Ts, pc')\#frs, retv))$$

$$run\_instr(Move\ dest\ src)\ P\ h\ xcptv\ reg\ C\ M\ Ts\ pc\ frs\ retv = (None, h, (xcptv, reg[dest := reg!src], C, M, Ts, pc+1)\#frs, retv))$$

$$run\_instr(MoveResult\ n)\ P\ h\ xcptv\ reg\ C\ M\ Ts\ pc\ frs\ retv = (None, h, (xcptv, reg[n := retv), C, M, Ts, pc+1)\#frs, Unit))$$

$$run\_instr(MoveException\ n)\ P\ h\ xcptv\ reg\ C\ M\ Ts\ pc\ frs\ retv = (None, h, (Unit, reg[n := xcptv], C, M, Ts, pc+1)\#frs, retv))$$

Fig.1 State transitions of Micro-Dalvik VM

图 1 Micro-Dalvik 虚拟机的状态转换

函数 *method* 调用 *seesMethod*,并通过函数 *seesMethod* 调用 *seesMethodDef*,这两个函数是实现方法覆盖机制的主要函数.函数 *seesMethodDef* 的归纳步和函数 *seesMethod* 如公式(5)和公式(6)所示:

$$\left.\begin{array}{l} seesMethodDefInduct: \\ \left[\!\!\left[\begin{array}{l} getClassDef\ P\ C = \lfloor (D, fs, ms) \rfloor; C \neq Object; seesMethodDef\ P\ D\ MmNameTs; \\ MmNameTs' = MmNameTs + + \\ \qquad (Option.map\ (\lambda TyMbody.\ (TyMbody, C)) \circ \\ \qquad\ (map\_of\ (\lambda(mName, Ts, T, mobdy).((mName, Ts), T, mbody))\ ms))) \end{array}\right]\!\!\right] \Rightarrow \\ seesMethodDef\ P\ C\ MmNameTs' \end{array}\right\} \tag{5}$$

$$seesMethod\ P\ C\ M\ Ts\ T\ m\ D \equiv \exists Mm.\ seesMethodDef\ P\ C\ Mm \wedge Mm(M, Ts) = \lfloor ((T, m), D) \rfloor \tag{6}$$

公式(5)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,公式(6)根据给定的(m,Ts)可以查找到其对应的方法体m.其实现是: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λ演算将表示方法的四元组变换成三元组,其中,第 1 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 1 个λ演算形成函数组合.++运算将最后的效果实现为:如果父类和子类具有同名同参数列表的方法,函数 *method* 确定的方法为子类中定义的的方法.

方法调用的另一种形式 $Invoke\ n_s\ n_e\ M'$用在参数个数大于 4 的方法调用中.操作数 n_s 和 n_e 分别表示实参寄存器的起止编号.按照 JVM 字节码翻译到 Dalvik VM 字节码的翻译方法,代表着调用方法的对象.除了使用函数 *upt* 获得参数寄存器的列表以外,其他都与 *Invoke ns M'*类似.

文献[20]实现的方法覆盖只需要满足方法名相同,从这种意义上讲,本文形式化的方法覆盖更接近于规范中的方法覆盖.文献[27]的方法调用语义区分了静态方法调用和动态方法调用,但是文中指出其忽略了方法查找的定义,仅说明这是一种标准方法.

返回指令 *Return n* 的语义定义首先判断运行时栈中是否还存在栈桢:如果栈桢为空,则程序结束;如果不为空,则将调用 *Return* 所在的方法的栈桢作为当前栈桢,其原程序计数增 1,新的状态四元组的第 4 个元素的值为寄存器 *n* 中的值.*ReturnVoid* 的语义规则类似,由于返回值为空,新状态的 *retv* 保持不变.

函数 *lookupHandler* 实现异常处理的语义,其定义如公式(7)所示:

$$\left.\begin{array}{l} lookupHandler\ P\ a\ h\ [\cdot]\ retv) = (\lfloor a \rfloor, h, [\cdot], retv) \\ lookupHandler\ P\ a\ h\ (fr\#frs)\ retv) = \\ \quad (\text{let}\ (None, reg, C, M, Ts, pc) = fr \\ \quad\ \text{in case}\ matchExptable\ P\ (cnameOf\ h\ a)\ pc\ (expTableOf\ P\ C\ M\ Ts)\ \text{of} \\ \qquad None \Rightarrow lookupHandler\ P\ h\ a\ frs\ retv \\ \qquad |\lfloor h\_pc \rfloor \Rightarrow (None, h, (\lfloor a \rfloor, reg, C, M, Ts, h\_pc)\#frs, retv)) \end{array}\right\} \tag{7}$$

其中,函数 *expTableOf* 获得形参类型为 *Ts* 的方法 *M* 的异常表.*matchExptable* 在异常表中进行查找,如果找到与抛出的异常匹配的表项,则该表项的处理块程序计数作为新状态中栈桢的程序计数,新状态中栈桢的第 1 个元素为该异常对象的地址,从而,异常处理块的第 1 条指令 *MoveException n* 可以从当前状态的栈桢中取出该异常对象的地址,并移至寄存器 *n* 中,程序跳转到异常处理块继续正常向下运行.

以上以函数方式定义了 Micro-Dalvik VM 单步执行的状态转换,程序的执行由任意有限数量的单步执行组成,可以定义在单步执行状态转换二元组集合的自反传递闭包上[20].公式(8)使用 *inductive_set* 给出了单步状态转换的集合定义方式,公式(9)给出了程序执行的状态转换关系.

$$\left.\begin{array}{l} inductive\_set\ runRel :: dvm\_prog \Rightarrow (dvm\_state \times dvm\_state)set \\ \qquad \text{and}\ runRel' :: dvm\_prog \Rightarrow dvm\_state \Rightarrow dvm\_state \Rightarrow bool\ \text{for}\ P :: dvm\_prog \\ \text{where}\ runRel'\ P\ \sigma\ \sigma' \equiv (\sigma, \sigma') \in runRel\ P\ | \\ \quad runRel\ I : run\ (P, \sigma) = \sigma' \Rightarrow runRel'\ P\ \sigma\ \sigma' \end{array}\right\} \tag{8}$$

$$runProg\ P\ \sigma\ \sigma' \leftrightarrow (\sigma, \sigma') \in (runRel\ P)^* \tag{9}$$

其中,$(runRel\ P)^*$代表集合 *runRel P* 的自反传递闭包.关系定义的方式将有助于下一步研究工作中的某些证明,

这个关系方式定义的状态转换等价于函数方式定义的状态转换,我们在下一节中予以证明.

2.3 语义性质定理证明

每条语句执行的状态转换都对应一条语义定理,即执行该语句,由当前状态转换到一个新的状态.这些定理在 Isabelle/HOL 中可直接运用状态转换函数的定义得到证明,并运用于编译前后语义保持证明中.由于可信编译使用归纳证明的方式来证明编译前后语义的保持,语义先是以函数方式定义的,用来直观地表达执行性;而后给出的关系方式的定义能够方便地用于归纳证明,故本节给出了上述函数方式定义的语义和关系方式定义的语义之间的等价性定理,并利用 Isabelle/HOL 的两个主要证明工具——经典推理器(classical reasoner)和简化器(simplifier)证明了这些定理,从而进一步验证了这个 Micro-Dalvik VM 模型的正确性.

定理 1. 对于程序 P,使用归纳定义的状态转换关系 *runRel* 等价于函数 *run* 的定义,即

$$runRel\ P=\{(\sigma,\sigma').run(P,\sigma)=\lfloor\sigma'\rfloor\}.$$

证明:根据公式(8),定理 1 的证明等价于证明:对于所有的 a,aa,ab,b,ac,ad,ae 和 ba,公式(10)和公式(11)成立:

$$runRel'\ P\ (a,aa,ab,b)\ (ac,ad,ae,ba)\Rightarrow run(P,a,aa,ab,b)=\lfloor(ac,ad,ae,ba)\rfloor \tag{10}$$

$$run(P,a,aa,ab,b)=(ac,ad,ae,ba)\Rightarrow runRel'\ P\ (a,aa,ab,b)\ (ac,ad,ae,ba) \tag{11}$$

公式(8)归纳定义的集合蕴含一个消除规则 *runRel.cases*,如公式(12)所示:

$$[\![runRel'\ ?P\ ?a1.0\ ?a2.0;\ \forall\sigma\ \sigma'.[\![?a1.0=\sigma;\ ?a2.0=\sigma';\ run(?P,\sigma)=\lfloor\sigma'\rfloor]\!]\Rightarrow ?Pa]\!]\Rightarrow ?Pa \tag{12}$$

使用相等替换,由公式(12)可以推出公式(10)成立.使用引入规则 *runRel*I 和相等替换,可以推出公式(11)成立.公式(10)和公式(11)的成立,表示证明结束.使用 Isabelle/HOL 的证明语言,上述证明过程如图 2 所示.

```
lemma runRel_eq[simp]: runRel P = {(σ,σ').run(P,σ) = ⌊σ'⌋}
apply auto
apply (erule runRel.cases)
    apply simp
  apply (rule runRelI)
    apply simp
done
```

Fig.2 Proof of Theorem 1 in Isabelle/HOL

图 2 利用 Isabelle/HOL 证明定理 1

证明完毕. □

定理 2. 对于程序 P,使用归纳定义的状态转换关系 *runRel'* 等价于函数 *run* 的定义,即

$$runRel'\ P\ \sigma\ \sigma'=(run(P,\sigma)=\lfloor\sigma'\rfloor).$$

证明:

$$\begin{aligned}runRel'\ P\ \sigma\ \sigma' &= (run(P,\sigma)=\lfloor\sigma'\rfloor) \overset{(runRel\_eq)}{=} ((\sigma,\sigma')\in\{(\sigma,\sigma').run(P,\sigma)=\lfloor\sigma'\rfloor\})\\ &= (run(P,\sigma)=\lfloor\sigma'\rfloor)=(run(P,\sigma)=\lfloor\sigma'\rfloor)=(run(P,\sigma)=\lfloor\sigma'\rfloor).\end{aligned}$$

在 Isabelle/HOL 的证明中,由于定理 *runRel_eq* 声明为 *simp*,该定理可以直接作为简化规则(simplification rule,或者称为重写规则,rewriting rule)被 *simplifer* 自动调用.因此,使用 *simp* 方法,该定理可以直接得到证明. □

定理 3. 执行程序 P 的状态转换等价于满足状态转换函数 *run* 的状态二元组集合的自反传递闭包,即

$$runProg\ P\ \sigma\ \sigma'=((\sigma,\sigma')\in\{(\sigma,\sigma').run\ P\ \sigma=\lfloor\sigma'\rfloor\}^*).$$

证明:

$$\begin{aligned}runProg\ P\ \sigma\ \sigma' &= ((\sigma,\sigma')\in\{(\sigma,\sigma').run\ P\ \sigma=\lfloor\sigma'\rfloor\}^*) \overset{(runProg\_def)}{=} ((\sigma,\sigma')\in(runRel\ P)^*)\\ &= ((\sigma,\sigma')\in\{(\sigma,\sigma').run(P,\sigma)=\lfloor\sigma'\rfloor\}^*) \overset{(runRel\_eq)}{=} ((\sigma,\sigma')\in\{(\sigma,\sigma').run(P,\sigma)=\lfloor\sigma'\rfloor\}^*)\\ &= ((\sigma,\sigma')\in\{(\sigma,\sigma').run(P,\sigma)=\lfloor\sigma'\rfloor\}^*).\end{aligned}$$

该证明过程在 Isabelle/HOL 中可以通过调用 *simp* 方法直接得到证明. □

以上定理是 Micro-Dalvik VM 语义所满足性质的一部分,其他性质可类似地证明,在此不再赘述.

3 总结与下一步工作

本文给出了一个类 Dalvik VM 的模型 Micro-Dalvik VM,包括虚拟机的指令集和虚拟机运行时状态的形式化;并以大步操作语义的方式给出了指令单步执行的状态转换以及定义在单步执行上的自反传递闭包,以表达虚拟机程序的运行时状态转换;最后,以定理的形式描述了语义满足的性质,并得到证明.在这个模型中,其指令系统包括了大部分 Dalvik 虚拟机指令.它在 Dalvik VM 指令上进行了必要的抽象,对其实质没有改变,因而具有较大的实用性.该形式化模型通过了定理证明助手 Isabelle/HOL 2009 的验证.目前,我们暂未考虑同步和数组等的相关操作,这些省略是为了使当前可信编译的构造和证明不过于复杂.它可能对本文工作的实用性有一定的限制,但对文中所描述的语义没有实质性影响,我们会在后续工作中进一步完善.

本文讨论的 Micro-Dalvik VM 模型是可信编译研究工作的一部分,但虚拟机在实际应用中可作为一个独立运行的单元,因此,这个模型也具有独立的应用价值.Google 也提出了一种称为 ART(android runtime)的虚拟机,在用户安装应用时,就进行预编译字节码到本地机的操作,将 Dalvik 原本在程序运行时的编译动作提前到应用安装时,应用的运行效率得以提高.本文的工作是在字节码这一层,无论是 Dalvik 还是 ART,它们都要以这个字节码为基础,将其执行到本地机.Google 继续进行其性能上的改进,也从某种意义上说明本文的研究工作仍然具有较好的实际价值.

本文描述的 Micro-Dalvik 虚拟机语言是类型化的语言,对它的语义形式化独立于其类型系统.对类型系统及字节码的验证,我们将另文讨论.下一步,我们将结合已经完成的类 Java 语言的形式化,将类 Java 的源语言翻译到目标 Micro-Dalvik 虚拟机指令,并基于语义一致性原理,在源语言的抽象语法上进行结构化归纳证明,实现可信编译.

致谢 在此,我们向对本文的工作给予支持和建议的同行表示感谢,特别向武汉大学计算机学院何炎祥教授领导的讨论班上的同学和老师表示感谢.

References:

[1] Ehringer D. The Dalvik virtual machine architecture. Technical Report, 2010. http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf

[2] Davis B, Beatty A, Casey K, Gregg D, Waldron J. The case for virtual register machines. In: Proc. of the 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME 2003). ACM Press, 2003. 41−49. [doi: 10.1145/858570.858575]

[3] Security Engineering Research Group. Analysis of Dalvik virtual machine and class path library. Technical Report, Pakistan: Institute of Management Sciences Peshawar, 2009.

[4] He YX, Wu W. Theory and key technology of trusted compiler. Beijing: Science Press, 2013. 15−58 (in Chinese).

[5] Sebesta RW. Concepts of Programming Languages. 10th ed., Boston: Addison-Wesley Educational Publishers, Inc., 2012.

[6] Slonneger K, Kurtz BL. Formal Syntax and Semantics of Programming Languages. Boston: Addison Wesley Longman, 1995.

[7] 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. Journal of Frontiers of Computer Science and Technology, 2011,5(1):1−22 (in Chinese with English abstract). [doi: 10.3778/j.issn.1673-9418.2011.01.001]

[8] Xu C, He YX, Wu W, Chen Y, Liu JB. Verifying implementation correctness of compiling optimization based on simulation relation. Chinese Journal of Electronics, 2012,40(11): 2171−2176 (in Chinese with English abstract). [doi: 10.3969/j.issn.0372-2112.2012.11.005]

[9] Nipkow T, Klein G. Concrete semantics—A proof assistant approach. Technical Report. http://www21.in.tum.de/~nipkow/Concrete-Semantics

[10] He YX, Wu W, Chen Y, Li QA, Liu JB. A kind of safe typed memory model for C-like languages. Journal of Computer Research and Development, 2012,49(11):2440−2449 (in Chinese with English abstract).

[11] Nipkow T, Paulson CL, Wenzel M. A Proof Assistant for Higher-Order Logic. Berlin, Heidelberg: Springer-Verlag, 2010.

[12] Wenzel M. The Isabelle/Isar Reference Mannual. Berlin, Heidelberg: Springer-Verlag, 2010.

[13] David von Oheimb. Analyzing Java in Isabelle/HOL—Formalization, type safety and Hoare logic [Ph.D. Thesis]. Munich: Technical University of Munich, 2000.

[14] Plotkin GD. The origins of structural operational semantics. Journal of Logic and Algebraic Programming, 2004,60-61:3−15. [doi: 10.1016/j.jlap.2004.03.009]

[15] Plotkin GD. A structural approach to operational semantics. Journal of Logic and Algebraic Programming, 2004,60-61:17−139. [doi: 10.1016/j.jlap.2004.05.001]

[16] Kahn G. Natural semantics. In:. Brandenburg FJ, Vidal-Naquet G, Wirsing M, eds. Proc. of the 4th Annual Symp. on Theoretical Aspects of Computer Science. London: Springer-Verlag, 1987. 19−21. [doi: 10.1007/BFb0039592]

[17] Clement D, Despeyroux J, Despeyroux T, Hascoet L, Kahn G. Natural semantics on the computer. In: Fuchi K, Nivat M, eds. Proc. of the France-Japan AI and CS Symp. 1986. 49−89.

[18] Blazy S, Leroy X. Mechanized semantics for the clight subset of the C language. Journal of Automated Reasoning, 2009,43(3): 263−288. [doi: 10.1007/s10817-009-9148-3]

[19] Zadarnowski P. C lambda calculus & compiler verification [Ph.D. Thesis]. Sydney: University of New South Wales, 2011.

[20] Klein G, Nipkow T. A machine-checked model for Java-like language, virtual machine and compiler. ACM Trans. on Programming Languages and Systems, 2006,28(4):619−695. [doi: 10.1145/1146809.1146811]

[21] Barthe G, Dufay G, Jakubiec L, Serpette B, Sousa SM. Formal executable semantics of the JavaCard platform. In: Sands D, ed. Proc. of the Programming Languages and Systems Conf. New York: Springer-Verlag, 2001. 302−319. [doi: 10.1007/3-540-45309-1_20]

[22] Liu HB, Moore JS. Executable JVM model for analytical reasoning: A study. Science of Computer Programming—Special Issue on Advances in Interpreters, Virtual Machines and Emulators, 2005,57(3):253−274. [doi: 10.1016/j.scico.2004.07.004]

[23] Hagiya M, Tozawa A. On a new method fot dataflow analysis of Java virtual machine. In: Proc. of the Static Analysis Symp. 1998. 17−32. [doi: 10.1007/3-540-49727-7_2]

[24] Alves-Foss J. Formal syntax and semantics of java. LNCS 1523, New York: Springer-Verlag, 1999.

[25] Stark R, Schmid J, Borger E. Java and the Java Virtual Machine—Definition, Verification, Validation. New York: Springer-Verlag, 2001.

[26] Payet É, Spoto F. An operational semantics of android activities. In: Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation. New York: ACM Press, 2014. 121−132. [doi: 10.1145/2543728.2543748]

[27] Jeon J, Micinski KK, Foster JS. SymDroid: Symbolic execution for Dalvik bytecode. Technical Report, CS-TR-5022, Department of Computer Science, University of Maryland, 2012.

[28] Erik RW. Formalization and analysis of Dalvik bytecode. Science of Computer Programming—Special Issue on Bytecode, 2012,92: 25−55.

[29] Lindholm T, Yellin F, Bracha G, Buckley A. The Java Virtual Machine Specification. Reading: Addison-Wesley, 2013.

[30] Paller G. Dalvik opcodes. http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html

[31] Android. Bytecode for the Dalvik VM. http://source.android.com/tech/dalvik/dalvik-bytecode.html

[32] McCarthy J, Painter J. Correctness of a compiler for arithmetic expression. In: Proc. of the IFTP Congress. 1967. 21−28.

[33] He XF.Study on runtime recovering technology in Dalvik virtual [MS. Thesis]. Chengdu: University of Electronic Science and Technology of China, 2013 (in Chinese with English abstract).

附中文参考文献:

[4] 何炎祥,吴伟.可信编译构造理论与关键技术.北京:科学出版社,2013.15−58.

[7] 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.计算机科学与探索,2011,5(1): 1−22. [doi: 10.3778/j.issn.1673-9418. 2011.01.001]

[8] 徐超,何炎祥,吴伟,陈勇,刘健博.基于模拟关系的编译优化实现正确性验证方法.电子学报,2012,40(11):2171−2176. [doi: 10.3969/j.issn.0372-2112.2012.11.005]

[10] 何炎祥,吴伟,陈勇,李清安,刘健博.一种用于类 C 语言环境的安全的类型化内存模型.计算机研究与发展,2012,49(11):2440–2449.

[33] 何晓峰.Dalvik 虚拟机下 Runtime 加载技术研究[硕士学位论文].成都:电子科技大学,2013.

附录 A: 一个 Java 源程序示例

```
class A{ int x;
   int getX(·){ return x;}
   void putX(int x){ this.x=x;}
   void sum(·){ int i, sum=0;
      for (i=0; i<x; i=i+1) {
         sum=sum+i;}
   }
}
class B extends A{ int y;
   int getY(·){ return y;}
   void putY(int y){ this.y=y;}
   void sum(·){ int i, sum=0; ;
      for (i=1; i<y; i=i+1)
         sum=sum+i;}
   void test(·){
      A a=new A(·);
      a.putX(10); a.sum(·);
         a=new B(·); a.sum(·);
         B b=new B(·);
      b.putY(100) b.sum(·);
         try{
            a=null;
            if (b.getX(·)<50)
               b.putX(100);
            else
               b.putX(200);
         }catch(NullPointerException e){
            throw e;} }
}
```

附录 B: 附录 A 中的程序实例转换到 Micro-Dalvik VM 所对应的程序

```
Program==[A_decl,B_decl]
A_decl==(A_name,A_class)
A_name==“A”
A_class==(Object,[(“x”,Integer)],
              [(“getX”,[·],Integer,(2,A_getX_ins,[·])),
               (“putX”,[Integer],Void,(4,A_putX_ins,[·])),
               (“sum”,[·],Void,(5,A_sum_ins,[·]))])
A_getX_ins==[(Move 0 2),(Move 1 0),(Iget 1 1 “x” “A”),
                 (Move 0 1),(Return 0)]
```

A_putX_ins==[(*Move* 0 4),(*Move* 1 5),(*Move* 2 0),
(*Move* 3 1),(*Iput* 3 2 "*x*" "*A*"),*ReturnVoid*]
A_sum_ins==[(*Move* 0 5),(*Const* 3 (*Intg* 0),(*Move* 2 3),
(*Const* 3 (*Intg* 0)),(*Move* 1 3),(*Move* 3 1),
(*Move* 4 0),(*Iget* 4 4 "*x*" "*A*"),(*Cmp* 3 3 4),
(*Ifop IfLez* 3 10),(*Move* 3 2),(*Move* 4 1),
(*AriBinop Add* 3 3 4),(*Move* 2 3),(*Move* 3 1),
(*Const* 4 (*Intg* 1)),(*AriBinop Add* 3 3 4),(*Move* 1 3),
(*Goto* (–13)),*ReturnVoid*]
B_decl==(*B_name*,*B_class*)
B_name=="*B*"
B_class==(*A_name*,[("*y*",*Integer*)],
[("*getY*",[·],*Integer*,(2,*B_getY_ins*,[·])),
("*putY*",[*Integer*],*Void*,(4,*B_putY_ins*,[·]),
("*sum*",[·],*Void*,(5,*B_sum_ins*,[·]),
("*test*",[·],*Void*,(7,*B_test_ins*,[(20,32,*NullPointer*,33)]))])
B_getY_ins==[(*Move* 0 2),(*Move* 1 0),(*Iget* 1 1 "*y*" "*B*"),
(*Move* 0 1),(*Return* 0)]
B_putY_ins==[(*Move* 0 4),(*Move* 1 5),(*Move* 2 0),
(*Move* 3 1),(*Iput* 3 2 "*y*" "*B*"),*ReturnVoid*]
B_sum_ins==[(*Move* 0 5),(*Const* 3 (*Intg* 0),(*Move* 2 3),
(*Const* 3 (*Intg* 0)) (*Move* 1 3),(*Move* 3 1),
(*Move* 4 0),(*Iget* 4 4 "*y*" "*B*"),(*Cmp* 3 3 4),
(*Ifop IfLez* 3 10),(*Move* 3 2),(*Move* 4 1),
(*AriBinop Add* 3 3 4),(*Move* 2 3),(*Move* 3 1),
(*Const* 4 (*Intg* 1)),(*AriBinop Add* 3 3 4),(*Move* 1 3),
(*Goto* (–13)),*ReturnVoid*]
B_test_ins==[(*Move* 0 7),(*NewInstance* 4 "*A*"),(*Move* 1 4),
(*Move* 4 1),(*Const* 5 (*Intg* 10)),(*Invoke* [4,5] "*putX*"),
(*Move* 4 1),(*Invoke* [4] "*sum*"),(*NewInstance* 4 "*B*"),
(*Move* 1 4),(*Move* 4 1),(*Invoke* [4] "*sum*"),
(*NewInstance* 4 "*B*"),(*Move* 2 4),(*Move* 4 2),
(*Const* 5 (*Intg* 100)),(*Invoke* [4,5] "*putY*"),(*Move* 4 2),
(*Invoke* [4] "*sum*"),(*Move* 4 2),(*Invoke* [4] "*getX*"),
(*MoveResult* 4),(*Const* 5 (*Intg* 50)),(*Cmp* 4 4 5),
(*Ifop IfLez* 4 5),(*Move* 4 2),(*Const* 5 (*Intg* 100)),
(*Invoke* [4,5] "*putX*"),*eturnVoid*,(*Move* 4 2),
(*Const* 5 (*Intg* 200)),(*Invoke* [4,5] "*putX*"),(*Goto* –4),
(*MoveExcetpion* 4),(*Move* 3 4),(*Move* 4 3),
(*Throw* 4)]

附录 C: 附录 A 中的程序实例中的方法转换到 Android 平台的 Dalvik VM 指令

A*.*getX:(·)***I*** *registers_size*: 0003
0000: *move-object v*0, *v*2
0001: *move-object v*1, *v*0

0002: *iget v*1, *v*1, *A.x*:*I* //*field*@0000
0004: *move v*0, *v*1
0005: *return v*0
A.putX:(***I***)***V*** *registers_size*: 0006
0000: *move-object* v0, *v*4
0001: *move v*1, *v*5
0002: *move-object v*2, *v*0
0003: *move v*3, *v*1
0004: *iput v*3, *v*2, *A.x*:*I* //*field*@0000
0006: *return-void*
A.sum:(·)***V*** *registers_size*: 0006
0000: *move-object v*0, *v*5
0001: *const*/4 *v*3, #*int* 0 //#0
0002: *move v*2, *v*3
0003: *const*/4 *v*3, #*int* 0 //#0
0004: *move v*1, *v*3
0005: *move v*3, *v*1
0006: *move-object v*4, *v*0
0007: *iget v*4, *v*4, *A.x*:*I* //*field*@0000
0009: *if-ge* v3, v4, 0015 //+000*c*
000b: *move v*3, *v*2
000c: *move v*4, *v*1
000d: *add-int*/2*addr v*3, *v*4
000e: *move v*2, *v*3
000f: *move v*3, *v*1
0010: *const*/4 *v*4, #*int* 1 //#1
0011: *add-int*/*lit*8 *v*3, *v*3, #*int* 1 //#01
0013: *move v*1, *v*3
0014: *goto* 0005 //−000*f*
0015: *return-void*
B.getY:(·)***I*** *registers_size*: 0003
0000: *move-object v*0, *v*2
0001: *move-object v*1, *v*0
0002: *iget* v1, v1, *B.y*:*I* //*field*@0000
0004: *move v*0, *v*1
0005: *return v*0
B.putY:(***I***)***V*** *registers_size*: 0006
0000: *move-object v*0, *v*4
0001: *move v*1, *v*5
0002: *move-object v*2, *v*0
0003: *move* v3, v1
0004: *iput v*3, *v*2, *B.y*:*I* //*field*@0000
0006: *return-void*
B.sum:(·)***V*** *registers_size*: 0006
0000: *move-object v*0, *v*5
0001: *const*/4 *v*3, #*int* 0 //#0

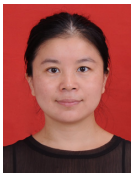
0002: *move* *v*2, *v*3
0003: *const*/4 *v*3, #*int* 1 //#1
0004: *move* *v*1, *v*3
0005: *move* *v*3, *v*1
0006: *move-object* *v*4, *v*0
0007: *iget* *v*4, *v*4, *B.y*:*I* //*field*@0000
0009: *if-ge* *v3*, *v*4, 0015 //+000*c*
000b: *move* *v*3, *v*2
000c: *move* *v*4, *v*1
000d: *add-int*/2*addr* *v*3, *v*4
000e: *move* *v*2, *v*3
000f: *move* *v*3, *v*1
0010: *const*/4 *v*4, #*int* 1 //#1
0011: *add-int*/*lit*8 *v*3, *v*3, #*int* 1 //#01
0013: *move* *v*1, *v*3
0014: *goto* 0005 //–000*f*
0015: *return-void*

B*.*test:(·)***V*** *registers_size*: 0008
0000: *move-object* *v*0, *v*7
0001: *new-instance* *v*4, *A* //*type*@0001
0003: *move-object* *v*6, *v*4
0004: *move-object* *v*4, *v*6
0005: *move-object* *v*5, *v*6
0006: *invoke-direct*{*v*5}, *A*.⟨*init*⟩:(·)*V* //*method*@0000
0009: *move-object* *v*1, *v*4
000a: *move-object* *v*4, *v*1
000b: *const*/16 *v*5, #*int* 10 //#000a
000d: *invoke-virtual*{*v*4,*v*5}, *A.putX*:(*I*)*V* //*method*@0001
0010: *move-object* *v*4, *v*1
0011: *invoke-virtual* {*v*4}, *A.sum*:(·)*V* //*method*@0002
0014: *new-instance* *v*4, *B* //*type*@0002
0016: *move-object* *v*6, *v*4
0017: *move-object* *v*4, *v*6
0018: *move-object* *v*5, *v*6
0019: *invoke-direct* {*v*5}, *B*.⟨*init*⟩:(·)*V* //*method*@0003
001c: *move-object* *v*1, *v*4
001d: *move-object* *v*4, *v*1
001e: *invoke-virtual* {*v*4}, *A.sum*:(·)*V* //*method*@0002
0021: *new-instance* *v*4, *B* //*type*@0002
0023: *move-object* *v*6, *v*4
0024: *move-object* *v*4, *v*6
0025: *move-object* *v*5, *v*6
0026: *invoke-direct* {*v*5}, *B*.⟨*init*⟩:(·)*V* //*method*@0003
0029: *move-object* *v*2, *v*4
002a: *move-object* *v*4, *v*2
002b: *const*/16 *v*5, #*int* 100 //#0064

002d: *invoke-virtual*{*v*4,*v*5}, *B.putY*:(*I*)*V* //*method*@0008
0030: *move-object* *v*4, *v*2
0031: *invoke-virtual*{*v*4}, *B.sum*:(·)*V* //*method*@0009
0034: *move-object* *v*4, *v*2
0035: *invoke-virtual*{*v*4}, *B.getX*:(·)*I* //*method*@0004
0038: *move-result* *v*4
0039: *const*/16 *v*5, #*int* 50 //#0032
003b: *if-ge* *v*4, *v*5, 0044 //+0009
003d: *move-object* *v*4, *v*2
003e: *const*/16 *v*5, #*int* 100 //#0064
0040: *invoke-virtual*{*v*4,*v*5}, *B.putX*:(*I*)*V* //*method*@0007
0043: *return-void*
0044: *move-object* *v*4, *v*2
0045: *const*/16 *v*5, #*int* 200 //#00*c*8
0047: *invoke-virtual*{*v*4,*v*5}, *B.putX*:(*I*)*V* //*method*@0007
004a: *goto* 0043 //–0007
004b: *move-exception* *v*4
004c: *move-object* *v*3, *v*4
004d: *move-object* *v*4, *v*3
004e: *throw* *v*4
padding: 0
tries:
try 0035...004*a*
catch java.lang.NullPointerException→004*b*
handlers:
size: 0001
0001: *catch java.lang.NullPointerException*→004*b*

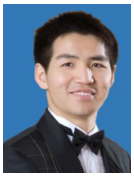
何炎祥(1952－),男,湖北应城人,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为可信软件,软件工程.

张军(1978－),男,博士生,副教授,CCF 会员,主要研究领域为可信软件,高性能计算.



江南(1976－),女,博士生,讲师,CCF 会员,主要研究领域为可信软件,可信编译.

沈凡凡(1987－),男,博士生,主要研究领域为可信软件,计算机体系结构,存储系统.



李清安(1986－),男,博士,讲师,CCF 会员,主要研究领域为编译优化,嵌入式系统.