一种用于字节码程序模块化验证的逻辑系统*

董渊[1+]，王生原[1]，张丽伟[2]，朱允敏[1]，杨 萍[3]

[1](清华大学 计算机科学与技术系,北京 100084)

[2](清华大学 软件学院, 北京 100084)

[3](北京语言大学 信息科学学院,北京 100084)

A Logic System for Bytecode Modular Certification*

DONG Yuan[1+], WANG Sheng-Yuan[1], ZHANG Li-Wei[2], ZHU Yun-Min[1], YANG Ping[3]

[1](Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

[2](School of Software, Tsinghua University, Beijing 100084, China)

[3](College of Information Science, Beijing Language and Culture University, Beijing 100084, China)

+ Corresponding author: Phn: +86-10-62794240, Fax: +86-10-62771138, E-mail: dongyuan@tsinghua.edu.cn

Received 2009-??-??

Abstract: Bytecode is a kind of interpretive execution instruction and a favorable intermediate presentation, which is widely used in network software and mass device. The work of certifying Bytecode can highly improve the reliability of relevant software and strongly support the construction of proof-preserved compiler, so it is of important theoretical value and practical value. Although some efforts have been made on building logic system for bytecode program, modular certification of bytecode still remains challenging because of the complexity of abstract control stack and the lack of control flow structure information. Moreover, most recent logic system's expression ability is much limited. In this paper, we present a new logic framework which supports modular certification of bytecode programs and is more expressive, and FPCC technology is originally introduced into our framework for bytecode. We provide the formal definition of CBP(Certifying Bytecode Program) logic system for the running environment of Bytecode. Also, we have finished the proof of the soundness theorem and a group of instance programs. Our work is not only a good solution for certifying Bytecode which is run on stack-based virtual machine but also a significant improvement for the construction of proof-preserved compiler environment. In addition, our system is useful for deeper understanding and analysis of network application based on virtual machine.

Key words: Program Modular Certification; Bytecode; Hoare-like Logic

---

* Supported by the National Natural Science Foundation of China under Grant No. 90818019 and 90816006 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant No. 2008AA01Z102 (国家高技术研究发展计划(863))

作者简介: 董渊（1973－），男，博士，讲师，研究领域为操作系统、编译系统和基于语言的可信软件等；王生原(1964－),男,博士,副教授,研究领域为程序设计语言与系统,Petri 网应用; 张丽伟 (1982－),女,硕士研究生,研究领域为基于语言的可信软件; 朱允敏 (1983－),男,硕士,研究领域为基于语言的可信软件;杨萍(1964－),女,副教授,主要研究领域为人工智能.

摘 要: 字节码既是运行于虚拟机的解释指令，也是定义良好的中间表示，是当今网络软件和计算设备中广泛使用的重要技术。字节码验证可以提高相关软件的可信程度，同时为构造证明保持编译器提供中间表示支持，具有重要的实用价值和理论价值。虽然近年提出了一些用于字节码程序的逻辑系统，但由于字节码本身的特点，造成抽象控制栈复杂、控制流结构信息不足，因而字节码程序的“模块化验证”依然是一个巨大的挑战，并没有得到有效解决。本文将 FPCC 方法引入中间表示字节码，借鉴汇编程序的验证方法，设计出一种逻辑系统，给出字节码程序运行环境 BCM（ByteCode Machine）的逻辑系统 CBP（Certifying Bytecode Program）定义，完成系统的合理性证明和一组代表性实例程序的模块化证明，并实现机器自动检查。本文工作为字节码验证提供一种良好的解决方案，同时也向着构造证明保持编译器环境迈出了坚实的一步，还可以为广泛使用的基于虚拟机复杂网络应用程序的深刻理解和深入分析提供理论帮助。

关键词: 程序模块化验证;字节码;类 Hoare 逻辑系统

中图法分类号: TP301 **文献标识码**: A

计算机软件逐步渗透到日常生产和生活中的各个环节，如何验证软件，保证程序正确、顺利执行，成为大家普遍关注和深入研究的一个热点问题，尤其是在国防、航空航天、金融、工业控制等重要领域。2003 年 Tony Hoare的论文“验证式编译器：计算研究的伟大挑战”[1]，是近年来人们关注验证技术的一个缩影。2009 年ACM通讯关于编译器研究的讨论中认定，软件自动化验证将是未来几十年极为重要的挑战之一[2]。

1 字节码验证的价值

本文研究字节码（bytecode）模块化验证的逻辑系统。以JAVA字节码（Java bytecode）[3]和微软.NET CIL[4]为代表的字节码是一种运行于虚拟机的中间语言，由虚拟机解释执行，同时也是定义良好的中间表示，是当今网络软件和计算设备中广泛使用的重要技术。

经典高级语言程序验证大都建立在抽象语言、代数演算或结构、状态机以及软件规范等较高抽象基础之上。经过可信验证的程序，若要将其映射到汇编语言时仍然保持可信，必须保证翻译变换的可信性。构造已验证编译器（Verified Compiler）保持高层到底层语义不变，或者构造证明保持编译器（Proof Preserving Compiler）将源代码及其证明转换为汇编代码及相应证明并进行自动检查[5]。这两种方法的实施都相当困难。

PCC（Proof-Carrying Code）方法将程序验证的对象从高级语言转向低层语言，在编译的代码生成阶段进行证明，在代码运行前使用可信验证器（Verifier）来检查目标代码和验证信息是否符合安全策略[6]。该方法既减轻静态检查负担，又可在验证时充分利用源语言或中间语言代码中蕴含的丰富信息。但是由于验证式编译器能力的限制，其安全策略的表达能力很弱，只能检查非常有限的程序特性。

针对这一问题，Appel等人提出FPCC（Foundational Proof-Carrying Code)方法[7]，系统的安全策略和证明均只基于特定逻辑系统，利用某种证明辅助工具完成某一特定语言的证明，并提供验证过程中的人工交互参与机制。因此程序规范的描述能力大大提高，同时能实现证明的机器自动检查。

基于FPCC思想，采用高阶逻辑同时描述汇编指令的操作语义和验证条件，Yu等人开发了一种基于逻辑的验证方法，可针对汇编级代码的正确性进行半自动证明[8]。随后经过深入研究，逐步实现存储分配、函数调用栈、代码指针、并发程序、自修改代码的验证[9][10]。之后我们与Feng等人的合作研究中提出了AIM系统，首次实现了中断支持的并发程序验证[11]。我们独立提出一系列基于有色网的可验证低级并发编程模型[12]和多核并发程序的验证逻辑系统[13]，中科大独立提出的指针逻辑[26]以及代码安全性证明构造[27]等方法。这些工作为汇编语言验证建立了坚实的理论基础，考虑到现实开发中很少直接编写汇编代码这一实际情况，在更高的语言层次上开展验证研究是一个非常迫切的需求。

从高级语言和汇编语言两个方面的发展趋势来看，都需要在中间表示层面进行深入细致的研究工作，而字节码是最佳选择之一。字节码验证可以大大提高相关软件的可信程度，具有相当的实用价值。将验证语言提高到字节码将带来验证效率方面的显著提升，同时可以为未来从高级语言到汇编语言的证明保持编译器构

造提供中间语言的支持，为可信软件构造提供有力的理论和工具支持。

以往针对JAVA字节码验证的主要研究工作集中于JVM内部检查器，目标是类型正确性，从而保证存储安全性[14]。随着研究工作的深入，人们逐步认识到仅有类型安全性检查远远不够，还需要进一步研究其它更多程序性质。近年人们提出大量用于字节码程序验证的逻辑系统，代表性研究工作包括：Quigley设计了一个用于字节码程序的类Hoare逻辑系统，证明了含有循环的程序片段[15]。Benton提出一种组合逻辑系统，用于一种.NET CIL的命令式语言子集，并给出纯手工证明[18]。Bannwart和Muller提出一种支持对象、引用、方法和继承等面向对象特征的逻辑系统，用来证明类JAVA 字节码[17]。MRG 项目主要着眼于程序资源，提出一种针对字节码的具有资源感知特征的操作语义，以及相应的逻辑系统[16]。特别值得一提的是字节码建模语言BML（Bytecode Modeling Language)的研究工作[19]，该语言作为一种可理解的字节码程序规范，应用开发人员可用它在字节码层面书写程序标注，以规范字节码程序的行为功能，与之对应的JML语言提供JAVA源语言层面的规范描述。该研究同时包含一个并不完整的JML到BML编译器，类似于PCC方法，其局限性在于证明检查仍然需要借助一个复杂的验证器。

现实中程序模块化验证是软件模块化开发所必需的方案。图 1、图 2 给出一个计算阶乘的被调函数 factor 及其调用函数 caller，这样的字节码程序涉及 while 循环控制结构、非局部变量访问和函数调用 / 返回等复杂特征（这里先跳过大括号中的内容，我们将在后面深入讨论）。如何模块化验证这样的代码，关键难点在于其不变量的形式化描述。但是，由于字节码程序抽象控制栈复杂、而控制流结构信息不足，“模块化验证”是一个巨大的挑战，目前尚缺乏有效的解决方案。

```
;int factor(){ r = 1; while(n != 0){ r = r*n; n = n-1; } }
;method factor: factorial, while loop with specification
-{(p0,  g0)}        ;I1(Instruction Sequence 1 ), entry point
0 pushc 1           ;push immediate data 1          | 8 pushc 1          ;push immediate data 1
1 pop r             ;r = 1                          | 9 binop            ;n-1
2 goto 11           ;jump to the end of while loop  | 10 pop n           ;save variable n
-{(p3,  g3)}        ;I2, loop start here            | -{(p11,  g11)}     ;I3
3 pushv r           ;push variable r                | 11 pushv n         ;push var n
4 pushv n           ;push variable n                | 12 pushc 0         ;push imm 0
5 binop*            ;r*n                            | 13 binop#          ;n#0?
6 pop r             ;save variable r                | 14 brture   3      ;conditional goto
7 pushv n           ;push variable n                | 15 ret             ;function ret
```

Fig. 1　Stack-based Bytecode Program, Callee factor

图 1　基于栈的字节码程序及其源代码，被调函数 factor

```
//function caller       | -{(p16,  g16)}    ;spec for caller
void caller(){          | 16 pushc 3        ;push imm 3
int n=3;                | 17 pop n          ;n = 3
call factor;            | 18 call 0         ;call factor()
}                       | -{(p19,  g19)}    ;spec for return poin
                        | 19 ret           ;caller return
```

Fig. 2　Stack-based Bytecode Programm, Caller

图 2　基于栈的字节码程序其源代码，调用函数 caller

本文首次将FPCC思想应用到中间表示字节码，提出一种新的逻辑系统，大幅提升规范表达能力，使用证明辅助工具Coq[20]给出系统的合理性证明和图 1、图 2 代码的证明，所有证明均可以自动检查[21]，很好地解决字节码程序的模块化验证问题。

2　逻辑系统

本文使用一种字节码子集 BC/0(ByteCode Zero)，类似于 JAVA 字节码和.NET CIL。本节给出 BC/0 对应虚拟机 BVM（Byetecode Virtual Machine）的定义，以及 BC/0 指令的操作语义，最后定义逻辑系统 CBP（Certifying Bytecode Program）。

2.1 字节码虚拟机BVM定义

图 3 给出本文字节码 BC/0 语言对应虚拟机 BVM 的定义，BVM 采用类似于 JAVA 虚拟机的双栈结构。整个机器配置（Machine Configuration）称为“世界” (World) $\mathbb{W}$，包含只读的代码堆(Code heap) $\mathbb{C}$、可修改的状态(State) $\mathbb{S}$、函数调用栈(Call Stack) $\mathbb{K}c$ 和程序计数器(Program Counter)pc。代码堆是代码标号(Labels)到指令序列(Instruction Sequences) $\mathbb{I}$ 的部分映射，状态 $\mathbb{S}$ 包含内存堆(Memory Heap) $\mathbb{H}$ 和计算栈(Evaluation Stack) $\mathbb{K}$，函数调用栈 $\mathbb{K}c$ 存放函数调用的返回地址，程序计数器指向代码堆中的当前指令。

(*World*)	$\mathbb{W}$	::=	$(\mathbb{C}, \mathbb{S}, \mathbb{K}c, pc)$		
(*CodeHeap*)	$\mathbb{C}$	::=	$\{f \to \mathbb{I}\}*$	(*State*)	$\mathbb{S}$ ::= $\{\mathbb{H}, \mathbb{K}\}$
(*CStack*)	$\mathbb{K}c$	::=	nil | f :: $\mathbb{K}c$	(*ProCnt*)	pc ::= *n* (*nat nums*)
(*Memory*)	$\mathbb{H}$	::=	$\{k \to w\}*$	(*EStack*)	$\mathbb{K}$ ::= nil | w :: $\mathbb{K}$
(*Labels*)	f, k	::=	*n* (*nat nums*)	(*Word*)	w ::= *i* (*integers*)
(*Instr*)	ι	::=	pushc w | pushv k | pop k | binop m | unop m | brtrue f | call f		
(*commd*)	c	::=	ι | ret | goto f	(*OprNum*)	m ::= {+.../, −...+}
(*InstrSeq*)	$\mathbb{I}$	::=	ι; $\mathbb{I}$ | ret | goto f		

Fig. 3 Definition of a Bytecode Machine

图 3 字节码机器的定义

指令序列 $\mathbb{I}$ 定义为一系列由跳转和返回指令结尾的指令片段，$\mathbb{C}[f]$ 表示 $\mathbb{C}$ 中由 f 开始的一个指令序列，$\mathbb{S}.\mathbb{K}$ 表示状态 $\mathbb{S}$ 中的栈 $\mathbb{K}$。函数 length()来获取栈 $\mathbb{K}$, $\mathbb{K}c$ 的栈顶位置，函数 max()获取栈 $\mathbb{K}$, $\mathbb{K}c$ 的上限。

$$\mathbb{C}[f] \triangleq \begin{cases} c & c=\mathbb{C}(f) \text{ and } c=\text{goto f', or ret} \\ \iota;\mathbb{I} & \iota=\mathbb{C}(f) \text{ and } \mathbb{I}=\mathbb{C}[f+1] \end{cases} \qquad (F\{a \to b\})(x) \triangleq \begin{cases} b & \text{if } x=a \\ F(x) & \text{otherwise} \end{cases}$$

$$validK\ \ n\ \ \mathbb{K} \triangleq length(\mathbb{K}) + n \le \max(\mathbb{K}) \qquad validKc\ \ n\ \ \mathbb{K}c \triangleq length(\mathbb{K}c) + n \le \max(\mathbb{K}c)$$

$$validRa\ \ \mathbb{K}c \triangleq \exists f, \exists \mathbb{K}c', \mathbb{K}c = f :: \mathbb{K}c'$$

Fig. 4 Definition of Representation

图 4 符号定义

2.2 指令操作语义

图 5 定义指令的操作语义，这里 Enable(c) $\mathbb{K}c$ $\mathbb{S}$ 是每一条指令 c 可以执行的最弱前条件，NextKc(c,pc, $\mathbb{S}$) 关系定义状态为 $\mathbb{S}$ 时 pc 所指指令执行后的函数调用栈变化，NextS(c,pc, $\mathbb{K}c$)关系定义栈为 $\mathbb{K}c$ 时 pc 所指指令执行后的状态变化，NextPC(c, $\mathbb{S}$, $\mathbb{K}c$)表示状态 $\mathbb{S}$、调用栈 $\mathbb{K}c$ 时 c 指令执行导致的 pc 变化。程序执行通过机器配置 $\mathbb{W}$ 的逐步转化来刻画，即 $\mathbb{W} \to \mathbb{W}'$ 是通过 pc 所指指令的执行而实现。

NextS(c,pc, $\mathbb{K}c$) $\mathbb{S}$ $\mathbb{S}'$ where $\mathbb{S}$ =($\mathbb{H}$, $\mathbb{K}$)

if c=	if Enable(c) $\mathbb{K}c$ $\mathbb{S}$ =	then $\mathbb{S}'$ =
pushc w	*validK* 0 $\mathbb{K}$	($\mathbb{H}$, w:: $\mathbb{K}$)
pushv f	*validK* 0 $\mathbb{K}$ and $\mathbb{H}$ (f)=w	($\mathbb{H}$, w:: $\mathbb{K}$)
pop f	$\mathbb{K}$ =w:: $\mathbb{K}'$	($\mathbb{H}$ {f→w}, $\mathbb{K}'$)
binop *bop*	$\mathbb{K}$ =w_1::w_2:: $\mathbb{K}'$, w=*bop*(w_1,w_2)	($\mathbb{H}$,w:: $\mathbb{K}'$)
unop *uop*	$\mathbb{K}$ =w1:: $\mathbb{K}'$, w=*uop*(w_1)	($\mathbb{H}$,w:: $\mathbb{K}'$)
brtrue f	$\mathbb{K}$ =w:: $\mathbb{K}'$,w=True or False	($\mathbb{H}$, $\mathbb{K}'$)
call f	*validKc* 1 $\mathbb{K}c$	($\mathbb{H}$, $\mathbb{K}$)
ret	*validRa* $\mathbb{K}c$	($\mathbb{H}$, $\mathbb{K}$)
…		($\mathbb{H}$, $\mathbb{K}$)

NextKc(c,pc,$\mathbb{S}$) $\mathbb{K}c$ $\mathbb{K}c'$ where $\mathbb{S}$=($\mathbb{H}$,$\mathbb{K}$)

if c=	if Enable(c) $\mathbb{K}c$ $\mathbb{S}$=	then $\mathbb{K}c'$=
call f	*validKc* 1 $\mathbb{K}c$	(pc+1):: $\mathbb{K}c$
ret	*validRa* $\mathbb{K}c$	$\mathbb{K}c'$
…	…	$\mathbb{K}c$

NextPC(c,$\mathbb{S}$,$\mathbb{K}c$) pc pc′ where $\mathbb{S}$=($\mathbb{H}$,$\mathbb{K}$)

if c=	if Enable(c) $\mathbb{K}c$ $\mathbb{S}$=	then pc′=
brtrue f	$\mathbb{K}$=w:$\mathbb{K}'$, w=True	f
	$\mathbb{K}$=w:$\mathbb{K}'$, w=False	pc+1
call f	*validKc* 1 $\mathbb{K}c$	pc+1
ret	*validRa* $\mathbb{K}c$	f
goto f		f
…	…	pc+1

$$\frac{c = \mathbb{C}(pc)\quad Enable(c)\ \mathbb{K}c\ \mathbb{S}\quad NextS(c,pc,\mathbb{K}c)\ \mathbb{S}\ \mathbb{S}'\quad NextK\,c_{(c,pc,S)}\ \mathbb{K}c\ \mathbb{K}c'\quad NextPC_{(c,\mathbb{S},\mathbb{K}c)}\ pc\ pc'}{(\mathbb{C},\mathbb{S},\mathbb{K}c,pc)\rightarrow(\mathbb{C},\mathbb{S}',\mathbb{K}c',pc')}\ (pc)$$

Fig. 5 Operational Semantics of BVM

图 5 BVM 机器操作语义

2.3 程序规范

本文直接使用 Coq 证明辅助工具的内嵌逻辑作为程序规范（Specification）书写语言，该逻辑是一种使用归纳定义的高阶谓词逻辑。程序员在每个指令序列开始处插入断言（Assertion）s，断言是谓词二元组(p,g)，插入断言之后的代码见图 1、图 2，其中的大括号给出程序断言。谓词 p 描述程序函数调用栈 $\mathbb{K}c$ 和状态 $\mathbb{S}$ 的性质，谓词 g 描述两个程序状态之间的关系，Coq 中 p、g 均定义为返回值为命题的函数，分别以 $\mathbb{K}c$ 、$\mathbb{S}$ 以及两个 $\mathbb{S}$ 为参数。p 描述当前状态的前条件， g 描述程序当前点与函数返回点之间的状态变化（即程序行为）。图 5 中 Enable(c) $\mathbb{K}c$ $\mathbb{S}$ 就是一个谓词 p，而 NextS(c,pc, $\mathbb{K}c$)就是一个谓词 g，规范定义如图 6 所示。

(*Pred*)	$p \in CStack \rightarrow State \rightarrow Prop$	(*Guarantee*)	$g \in State \rightarrow State \rightarrow Prop$
(*Spec*)	$s ::= (p, g)$	(*MPred*)	$m \in Memory \rightarrow Prop$
(*CdHpSpec*)	$\Psi ::= \{(f_1, s_1), ..., (f_n, s_n)\}$		

Fig. 6 Specifications Constructs for CBP

图 6 CBP 的规范构造符

代码堆 $\mathbb{C}$ 对应的程序规范 Ψ 是代码标号f到相应程序断言s的映射集合。需要注意的是一个代码标号可能对应多个程序断言，其含义类似于一个函数可能有多个特定接口说明。我们使用谓词m来描述内存堆。如图 7 所示，在断言中使用分离逻辑[23]连接符来确保不同函数使用不同的内存空间。

$l \mapsto w \triangleq \lambda\mathbb{H}.\mathbb{H} = \{l \rightarrow w\}$ $\qquad$ $l \rightarrow \_ \triangleq \lambda\mathbb{H}.\exists w.\,(l \mapsto w)\mathbb{H}$

$\mathbb{H}_1 \# \mathbb{H}_2 \triangleq dom(\mathbb{H}_1) \cap dom(\mathbb{H}_1) = \varnothing$ $\qquad$ $\mathbb{H}_1 \uplus \mathbb{H}_2 \triangleq \begin{cases} \mathbb{H}_1 \cup \mathbb{H}_2 & \text{if } \mathbb{H}_1 \# \mathbb{H}_2 \\ undefined & \text{otherwise} \end{cases}$

$m_1 * m_2 \triangleq \lambda\mathbb{H}.\ \exists\mathbb{H}_1, \mathbb{H}_2.\ (\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{H}) \wedge m_1\,\mathbb{H}_1 \wedge m_2\,\mathbb{H}_2$

$p * m \triangleq \lambda\mathbb{S}.\ \exists\mathbb{H}_1, \mathbb{H}_2.\ (\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{S}.\mathbb{H}) \wedge p\,\mathbb{S} \mid \mathbb{H}_1 \wedge m\,\mathbb{H}_2$

Fig. 7 Definitions of “Separation Logic” Assertions

图 7 分离逻辑断言定义

$$\frac{\Psi \mapsto \mathbb{C} : \Psi' \quad \Psi \subseteq \Psi' \quad \Psi \mapsto \{s\}\mathrm{pc}{:}\mathbb{C}[\mathrm{pc}] \quad \{s\}\Psi'\mathbb{S}}{\Psi \mapsto \{s\}(\mathbb{C},\mathbb{S},\mathrm{pc})} \quad \text{(WLD)}$$

$$\frac{\textit{for all } (f,s) \in \Psi' : \ \Psi \mapsto \{s\}\, f{:}\mathbb{C}[f]}{\Psi \mapsto \{s\}(\mathbb{C},\mathbb{S},\mathrm{pc})} \quad \text{(CDHP)}$$

$$\frac{\Psi \mapsto \mathbb{C}_1 : \Psi' \quad \Psi \mapsto \mathbb{C}_2 : \Psi' \quad \mathbb{C}_1 \# \mathbb{C}_2}{\Psi \mapsto \{s\}(\mathbb{C},\mathbb{S},\mathrm{pc})} \quad \text{(LINK)}$$

$$\frac{\begin{array}{l} \iota \notin \{\mathrm{brtrue},\mathrm{call}\} \quad \Psi \mapsto \{(p'',g'')\}\mathrm{pc}+1 : \mathbb{I} \\ p \Rightarrow g_\iota \quad (p \triangleright g_\iota) \Rightarrow p'' \quad (p \circ (g_\iota \circ g'')) \Rightarrow g \end{array}}{\Psi \mapsto \{(p,g)\}\mathrm{pc}{:}\iota;\mathbb{I}} \quad \text{(SEQ)}$$

$$\frac{\begin{array}{l} (f,(p',g')) \in \Psi \quad \Psi \mapsto \{(p'',g'')\}\mathrm{pc}+1 : \mathbb{I} \\ (p \triangleright g_{\mathrm{brtrueT}})\, p' \quad (p \circ (g_{\mathrm{brtrueT}} \circ g')) \Rightarrow g \\ (p \triangleright g_{\mathrm{brtrueF}})\, p'' \quad (p \circ (g_{\mathrm{brtrueF}} \circ g'')) \Rightarrow g \end{array}}{\Psi \mapsto \{(p,g)\}\mathrm{pc}{:}\mathrm{brtrue}\ f;\mathbb{I}} \quad \text{(BRTRUE)}$$

$$\frac{\begin{array}{l} (\mathrm{pc}+1,(p'',g'')) \in \Psi \quad \Psi \mapsto \{(p'',g'')\}\mathrm{pc}+1 : \mathbb{I} \\ (p \triangleright g_{\mathrm{call}})\, p' \quad (p \triangleright g_{\mathrm{fun}}) \Rightarrow p'' \quad (p \circ (g_{\mathrm{fun}} \circ g'')) \Rightarrow g \\ (f,(p',g')) \in \Psi \quad g_{\mathrm{fun}} = ((g_{\mathrm{call}} \circ g') \circ g_{\mathrm{ret}}) \end{array}}{\Psi \mapsto \{(p,g)\}\mathrm{pc} : \mathrm{call}\ f;\mathbb{I}} \quad \text{(CALL)}$$

$$\frac{(p \circ g_{\mathrm{ret}}) \Rightarrow g}{\Psi \mapsto \{(p,g)\}\mathrm{pc} : \mathrm{ret}} \quad \text{(RET)}$$

$$\frac{(f,(p',g')) \in \Psi \quad (p \triangleright g_{\mathrm{goto}}) \Rightarrow p' \quad (p \circ (g_{\mathrm{fun}} \circ g')) \Rightarrow g}{\Psi \mapsto \{(p,g)\}\mathrm{pc} : \mathrm{goto}\ f} \quad \text{(GOTO)}$$

Fig. 8 CBP Inference Rules

图 8 CBP 的推理规则

2.4 逻辑系统CBP

使用如下的判断来定义推理规则，其定义如图 8 所示。:

$\Psi \ \mapsto \ \{s\}\,\mathbb{W}$ (WLD，well-formed world)

$\Psi \ \mapsto \ \mathbb{C} : \Psi'$ (CDHP，well-formed code heap)

$\Psi \ \mapsto \ \{s\}\,\mathbb{I}$ (SEQ，well-formed instruction sequence)

程序不变量。WLD 规则中描述一个程序满足良型性(Well-formedness)的所有前提：

- 根据 CDHP 规则，代码堆 $\mathbb{C}$ 是良型的(Well-formed)。
- 当前模块对内规范 Ψ 是对外规范 Ψ'的子集，Ψ 中包含 $\mathbb{C}$ 所使用到的位于其它代码块的被调用接口规范，Ψ'还包含本模块定义并将将被其他模块调用的接口说明。
- 当前 pc 所对应的断言 s 在 Ψ 中，因此当前指令序列 $\mathbb{C}$[pc]关于规范 s 是良型的。
- 给定外部规范集 Ψ',当前状态 $\mathbb{S}$ 应满足断言 s。

规则所用符号定义如图 9。谓词$p \triangleright g_I$描述p满足初始状态并经过状态转移g_I之后的结果，它是状态转移g_I的最强后条件。两个连续状态转移g和g′的组合用$g \circ g'$来表示，$p \circ g$表示在g的基础已知p得到满足。

程序模块。在 CDHP 规则中，每个模块是由至少一个指令序列组成的小代码堆，每个模块的规范不仅包含当前模块内部代码块的规范，还包含了所有可能被该模块调用的其它代码块的规范，因此 CBP 逻辑支持各个程序模块的独立验证。每个独立模块的良型性通过 CDHP 规则定义，多个互不重叠的良型模块通过 LINK 规则链接起来。WLD 规则保证所有良型模块能够链接成为一个全局良型代码堆。

指令序列。类似传统Hoare逻辑[22]，本逻辑采用前、后条件作为程序规范。SEQ规则是对每个以顺序指令 ι（不含条件跳转和函数调用指令）开始序列的判定形式。它描述在给定内部规范 Ψ 和一个前条件（p，g）下执行以当前pc开始的指令序列是安全的。程序员需要找到一个中间断言（p″,g″）使得剩余指令序列得到满

足，该断言同时也作为当前指令的后条件。我们使用g_l来描述执行指令$\boldsymbol{l}$所引起的状态转移，具体定义如图 9 和图 3 所示。对于顺序指令而言，NextS不依赖于当前pc值，因此用“_”表示任意pc。

找到合适的中间断言（p′, g′）后，检查SEQ规则中的四个条件以确定该指令序列的良型性。第一个前提表明剩余指令序列在该中间断言下是良型的；接下来$p \Rightarrow g_l$检查只要初始状态满足p，那么状态转移g_l就可以完成；第三个前提表示如果当前状态满足p，那么在状态转移g_l后，新状态满足p′。最后一个前提描述在当前状态满足p的情况下，g_l和g′的组合能够满足g。

函数调用和返回。指令序列前条件包含一个描述当前状态的谓词 p 和一个描述当前状态和函数返回状态之间关系的谓词 g，g 通常覆盖多个指令序列。如果一个函数有多个返回点，g 将覆盖从当前点到任意返回点的所有路径。图 10(b)给出了图 10(a)中源代码对应字节码程序中函数 foo 的断言(p,g)及其含义。

图 10(c)表示函数foo在A点（pc=5）处调用函数bar（入口点为B），返回地址为pc+1。函数bar的断言为(p_B,g_B)。A与D两点处的断言分别为(p_A,g_A)和(p_D,g_D)。g_A覆盖的代码段为A-E，而g_D覆盖D-E。

为了确保程序行为的正确性，应满足如下条件：

1) 函数 bar 的前条件应得到满足：

$$\forall \mathbb{S}, \exists \mathbb{S}'.\ p_A\ \mathbb{S} \wedge g_{cal}\ \mathbb{S}\ \mathbb{S}' \rightarrow p_B\ \mathbb{S}'$$

2) 当函数 bar 返回时，调用函数 foo 从 D 点处开始恢复执行。

$$\forall \mathbb{S}, \mathbb{S}'', \mathbb{S}', \mathbb{S}*.\ \ p_A\ \mathbb{S} \rightarrow g_{cal}\ \mathbb{S}\ \mathbb{S}' \rightarrow g_B\ \mathbb{S}'\ \mathbb{S}* \rightarrow g_{ret}\ \mathbb{S}*\mathbb{S}'' \rightarrow p_D\ \mathbb{S}''$$

3) 如果函数 bar 和代码段 D-E 满足它们的断言，那么 A-E 的断言也将得到满足

$$\forall \mathbb{S}, \mathbb{S}'', \mathbb{S}''', \mathbb{S}', \mathbb{S}*.\ \ p_A\ \mathbb{S} \rightarrow g_{cal}\ \mathbb{S}\ \mathbb{S}' \rightarrow g_B\ \mathbb{S}'\ \mathbb{S}* \rightarrow g_{ret}\ \mathbb{S}*\mathbb{S}'' \rightarrow g_D\ \mathbb{S}''\ \mathbb{S}''' \rightarrow g\ \mathbb{S}\ \mathbb{S}'''$$

定义特殊谓词 $g_{fun} \triangleq \lambda \mathbb{S}, \mathbb{S}''.\ \exists \mathbb{S}', \exists \mathbb{S}*, g_{cal}\ \mathbb{S}\ \mathbb{S}' \wedge g_B\ \mathbb{S}'\ \mathbb{S}* \wedge g_{ret}\ \mathbb{S}*\mathbb{S}''$，则上述 2）、3）分别简写为：

$$\forall \mathbb{S}, \mathbb{S}''.\ \ p_A\ \mathbb{S} \rightarrow g_{fun}\ \mathbb{S}\ \mathbb{S}'' \rightarrow p_D\ \mathbb{S}''$$

$$\forall \mathbb{S}, \mathbb{S}''.\ \ p_A\ \mathbb{S} \rightarrow g_{fun}\ \mathbb{S}\ \mathbb{S}'' \rightarrow g_D\ \mathbb{S}''\ \mathbb{S}''' \rightarrow g\ \mathbb{S}\ \mathbb{S}''$$

上述条件都是CALL推理规则的前提。应注意的是这里不要求给出一个特定返回点，只要求在返回状态$\mathbb{S}''$点处，栈中包含一个在内部规范Ψ中有定义的代码指针，这就意味着本逻辑能够用来于支持任意调用约定规则的多返回点函数调用。RET规则较为直观，它要求在该点够顺利完成状态转移，这样能够满足g_{ret}的状态转移也应满足调用函数的g，规则中不需要任何关于返回地址的信息，因此可以实现被调用函数的模块化验证。可以看到g_{fun}描述了函数调用点到函数返回点之间的状态转移，其定义完整刻画了调用指令、被调用函数以及返回指令三者的行为，本文对函数调用返回的处理采用了和SCAP逻辑系统[9]相似的思路，SCAP针对MIPS机器，支持其调用约定，而我们的CBP用于字节码，具有单独的函数调用栈。虽然本文的CBP针对带有抽象函数调用栈的栈式虚拟机，但是g_{fun}的定义给出了一种通用的函数调用返回描述。

$$p \Rightarrow g \triangleq\ \forall \mathbb{S}, \mathbb{K}c.\ p\ \mathbb{K}c\ \mathbb{S} \rightarrow \exists \mathbb{S}',\ g\ \mathbb{S}\ \mathbb{S}' \qquad p \triangleright g \triangleq\ \lambda \mathbb{S}, \mathbb{K}c.\ \exists \mathbb{S}_0,\ p\ \mathbb{K}c\ \mathbb{S}_0 \wedge g\ \mathbb{S}_0\ \mathbb{S}$$

$$g \circ g' \triangleq\ \lambda \mathbb{S}, \mathbb{S}''.\ \exists \mathbb{S}',\ g\ \mathbb{S}\ \mathbb{S}' \wedge g'\ \mathbb{S}'\ \mathbb{S}'' \qquad p \Rightarrow p' \triangleq\ \forall \mathbb{S}, \mathbb{K}c.\ p\ \mathbb{K}c\ \mathbb{S} \rightarrow p\ \mathbb{K}c\ \mathbb{S}'$$

$$g \Rightarrow g' \triangleq\ \forall \mathbb{S}, \mathbb{S}'.\ g\ \mathbb{S}\ \mathbb{S}' \rightarrow g'\ \mathbb{S}\ \mathbb{S}' \qquad p \circ g \triangleq\ \lambda \mathbb{S}, \mathbb{S}'.\ \exists \mathbb{K}c,\ p\ \mathbb{K}c\ \mathbb{S} \wedge g\ \mathbb{S}\ \mathbb{S}'$$

$$g_{brtrueT} \triangleq \lambda \mathbb{S}, \mathbb{S}'.\ NextS_{(brtrue,\_)}\ \mathbb{S}\ \mathbb{S}' \qquad \text{(where } \mathbb{S}'.\mathbb{K} = w::\mathbb{K}',\ w = \text{True)}$$

$$g_{brtrueF} \triangleq \lambda \mathbb{S}, \mathbb{S}'.\ NextS_{(brtrue,\_)}\ \mathbb{S}\ \mathbb{S}' \qquad \text{(where } \mathbb{S}'.\mathbb{K} = w::\mathbb{K}',\ w = \text{False)}$$

$$g_c \triangleq \lambda \mathbb{S}, \mathbb{S}'.\ NextS_{(c,\_)}\ \mathbb{S}\ \mathbb{S}' \qquad \text{(for all other c)}$$

Fig. 9　Connectors for p and g, Local State Transitions

图 9　p g 连接符和指令状态转换标记

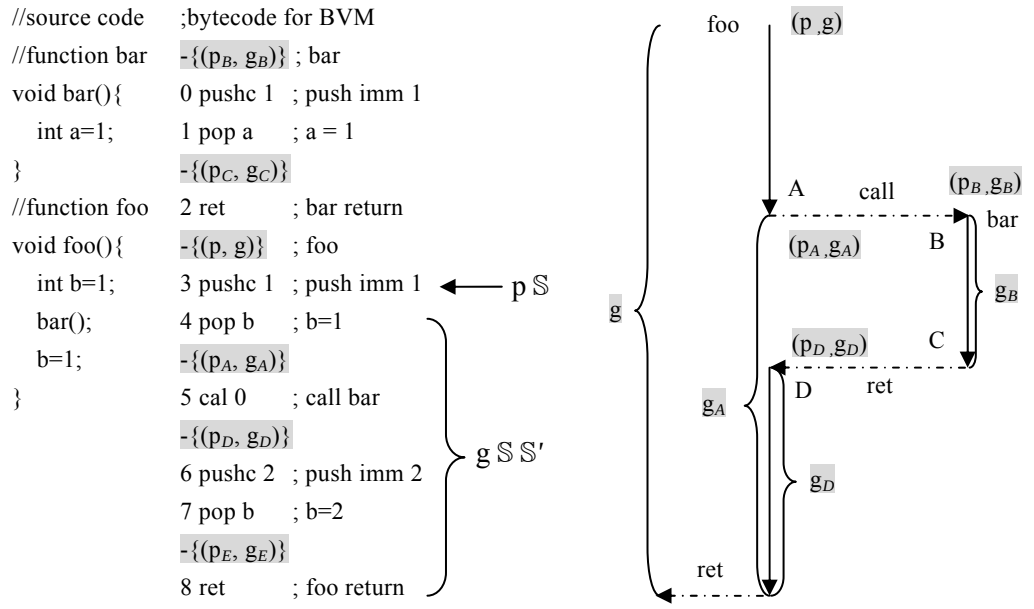

Fig. 10 The Model for Function Call/Return in CBP

图 10 CBP 函数调用/返回模型

栈不变量。为了保证函数调用栈的安全，递归定义“良型控制栈”为：

$$\mathrm{WFST}(g, \mathbb{K}c, \mathbb{S}, \Psi) \triangleq \neg\exists \mathbb{S}'.g\ \mathbb{S}\ \mathbb{S}',\ \text{where } \mathbb{K}c = \mathrm{nil}$$

$$\mathrm{WFST}(g, \mathbb{K}c, \mathbb{S}, \Psi) \triangleq \forall \mathbb{S}'.g\ \mathbb{S}\ \mathbb{S}' \to p'\ \mathbb{S}' \wedge \mathrm{WFST}(g', \mathbb{K}c', \mathbb{S}', \Psi),\ \text{where } \mathbb{K}c = f :: \mathbb{K}c', (p', g') = \Psi(f)$$

当函数调用栈为空时处于最外层的函数没有返回代码指针，因此不存在函数返回点状态$\mathbb{S}'$。

需要满足的栈不变量是，在每个带有程序断言(p,g)的程序点处，程序状态$\mathbb{S}$必须满足 p，并且存在一个良型的控制栈。该不变量形式化定义如下：

$$\{(p, g)\}\ \Psi\ \mathbb{S} \triangleq p\ \mathbb{K}c\ \mathbb{S} \wedge \mathrm{WFST}(g, \mathbb{K}c, \mathbb{S}, \Psi)$$

本定义中并并不需要考虑控制栈的实际深度，也并不需要描述栈的其它属性，这就使得本逻辑非常灵活通用。需要证明在程序执行的每一步，该不变量都能得到满足。这个栈不变量形式化地阐明了为什么不需要一个有效的代码指针就可以对返回指令进行“类型检查”。

其它指令。条件调整指令跳转成功与否依赖于其判定条件是否得到满足，因此BRTRUE规则中使用g_{brtrueT}和g_{brtrueF}来表示在不同的执行情况。无条件跳转指令可用安全执行的充分必要条件式当前断言能够蕴含目标代码处的断言，它可以看作是条件跳转BRTRUE的特例。

CBP 的完备性。基于前进性和保持性引理，CBP 的完备性保证只要初始状态满足 WLD 规则中定义的程序不变量，则整个程序将永远不会陷入滞留状态。程序运行过程中永远满足该不变量，该不变量还可以蕴含部分正确等更多程序性质。逻辑同时保证程序满足 goto 或 call 指令跳转目标地址处的断言，也满足程序模块边界处的断言。该完备性定理已经在辅助工具 Coq 中完成形式化证明，并实现机器自动检查。

Lemma 2.1 (CBP Progress) 如果$\Psi \mapsto \{s\}\mathbb{W}$，那么将存在程序$\mathbb{W}'$，使得$\mathbb{W} \to \mathbb{W}'$。

Lemma 2.2 (CBP Preservation) 如果$\Psi \mapsto \{s\}\mathbb{W}$，并且$\mathbb{W} \to \mathbb{W}'$，那么存在 s′，使得$\Psi \mapsto \{s'\}\mathbb{W}'$。

3 程序实例和实现

本文图 1、图 2(第 1 节)介绍的程序实例中，采用 while 循环和非本地变量实现阶乘函数及其调用函数。本节给出这组程序的证明，用以展示本逻辑系统的模块化验证能力，同时介绍程序断言书写步骤。

3.1 被调函数

$$\text{p-pre} \triangleq (validK\ \ 2\ \ \mathbb{K}) \wedge (validKc\ \ 0\ \ \mathbb{K}c) \wedge (validRa\ \ \mathbb{K}c)$$

$$p_0 \triangleq \text{p-pre} \wedge (r \to \_) * (\exists i >= 0, n \to i), \quad p_3 \triangleq \text{p-pre} \wedge (\mathbb{H}(r) >= 1) \wedge (\mathbb{H}(n) >= 0), \quad p_{11} \triangleq p_3,$$

$$g_0 \triangleq p_0 \to (\mathbb{H}'(r) = \mathbb{H}(n)!), \quad g_3 \triangleq p_3 \to (\mathbb{H}'(r) = \mathbb{H}(r)*\mathbb{H}(n)!), \quad g_{11} \triangleq g_3$$

Fig. 11 Specifications: While Loop Example

图 11 While 循环示例断言

获取指令序列。阶乘函数的源代码和带有断言的字节码程序见图 1(第 1 节)。划分指令序列是验证工作的第一步，由图 3 定义可知，一个指令序列以无条件跳转指令或者函数返回指令结束。由此得到，while 循环实现程序包含 3 个指令序列，标号 0-2 的指令构成第一个序列，第二个序列是标号为 3-15 的指令，指令 11-15 是第三个指令序列。

编写程序规范。接下来，程序员需要给出代码堆规范 Ψ，从指令标号 f 到代码断言 s（即 (p,g)二元组）的映射，我们将 CBP 断言内嵌在字节码代码中，即见图 1 中阴影部分。本例的断言见图 11，为了简化表达，这里省略了谓词的参数（机器状态 $\mathbb{S}$）。

根据推理规则，需要给出以下位置的代码断言：每一个指令序列的开头，调用指令、跳转指令(包括goto和brture)的目标位置，以及紧跟调用指令之后的调用返回行。函数的断言为(p_0,g_0)，p_0要求有足够的运算栈和函数控制栈空间、内存单元中所存放变量r和n值在有效范围内，谓词g_0给出了整个函数的功能说明，即如果p_0得到满足，则内存中的非局部变量r和n在函数运行之后满足阶乘关系(H′(r) = H(n)!)。(p_3,g_3)是循环体的断言，p_3表明有足够的运算栈空间、变量r和n的值仍在有效范围内，g_3表明内存中的变量值必须满足循环不变量。wihle循环条件判断开始处断言(p_{11},g_{11})与(p_3,g_3)相同。

验证并连接。为检查一个以指令 ι 开始序列的良型性，程序员需要寻找恰当的中间断言（即是当前指令 ι 的后条件，又是后续指令序列的前条件），并选用相应推理规则完成指令 ι 的证明。之后，程序员使用 CDHP 规则建立各个独立指令序列的良型性证明，多个互不重叠的良型代码片段可以用 LINK 规则连接在一起，最后应用 WLD 规则把所有代码片段连接并构成全局良型代码堆。

模块化支持。本实例中，CBP 逻辑推理规则中所使用的代码规范都是当前模块的局部规范，如图 8 所示，证明 ret 指令所使用的 RET 规则中没有使用任何关于返回地址的约束条件，也就是说验证被调函数的时候不需要任何有关返回地址的信息，因此 CBP 支持模块化验证。

3.2 调用函数

$$\text{p-pre}' \triangleq (validK\ \ 2\ \ \mathbb{K}) \wedge (validKc\ \ 1\ \ \mathbb{K}c) \wedge (validRa\ \ \mathbb{K}c)$$

$$p_{16} \triangleq \text{p-pre}' \wedge (r \to \_) * (n \to \_), \quad p_{19} \triangleq \text{p-pre}' \wedge (r \to 3!) * (n \to 0), \quad p_0 \triangleq ?$$

$$g_{16} \triangleq p_{16} \to \mathbb{H}'(r) = 3!, \quad g_{19} \triangleq p_{19} \to \mathbb{H}'(r) = 3!, \quad g_0 \triangleq ?$$

Fig. 12 Specifications: Caller of the Recursive Factorial

图 12 递归阶乘的调用者的断言

调用函数对应的源代码、带程序规范的字节码在图 2(第 1 节)中给出，该函数的功能是初始化变量n为 3，调用函数factor计算n的阶乘。函数入口点的断言为(p_{16},g_{16})，其中前条件p_{16}简单规定运行本函数需要足够的运算栈和函数调用栈空间、预留用以存放变量n和r的内存单元， 而谓词g_{16}规定该函数的功能：内存中计算结果r的值为 3 的阶乘。函数返回点断言为(p_{19},g_{19})，p_{19} 表明栈空间和变量n和r的内存单元依然有效，g_{19}和g_{16}相同。根据指令CAL的推理规则，可以看出证明调用者的过程中必须使用到被调函数的程序断言，这里我们使用图 11 中被调函数断言(p_0,g_0)。实际上，函数入口点的断言定义了该函数的接口，只要几个被调函数都实现了相同的接口，调用函数可以使用其中任意一个函数。

3.3 证明实现

本文采用Coq证明辅助工具来实现逻辑系统和上述实例程序的证明,所有定义和证明都可机器自动检查。实现给出 BVM 及其操作语义和 CBP 逻辑系统的形式化表示，BVM 语法采用 Coq 的归纳定义给出，其操作语义和所有的 CBP 推理规则都定义为归纳关系，逻辑系统的合理性证明则根据语法方式进行形式化和证明，同时还给出程序实例的形式化描述和证明。本逻辑系统采用 Coq 系统内部使用的高级逻辑 CiC，这样的方案大大降低了工作量。

本文给出的例子直接采用字节码开发，事实上本文逻辑系统同样可用于人工优化过的字节码程序，以及优化编译工具自动生成代码的证明。

Table 1 Coq proof code statistic
表 1 Coq 证明代码统计

Number	Type	Lines of proof code	
		Value	%
1	Basic coq tactic library	2354	28.4%
2	Bytecode Virtual Machine definition	3285	39.7%
3	CBP logic inference rule and soundness	1166	14.1%
4	Proof code of factorial example	1472	17.8%
5	Total	8277	100%

CBP 逻辑系统的实现包括约 8300 行的 Coq 代码，花费数个人月。其中接近 1/3 分的工作在于一些基本工具实现，包括关于映射与分离逻辑的引理和策略。这些通用工具独立于本逻辑系统以及验证实例，在将来的工作中可以重用，实际上这部分代码大多重用自以往的项目中。其中 3200 多行定义 BVM、及其操作语义和相关引理，约 1100 行为 CBP 推理规则定义及其完备性证明。

实践表明，编写程序规范依赖于程序员，其难度取决与所感兴趣的程序性质以及算法本身的复杂性。只要给出程序规范，字节码证明相对简单。本文图 1、图 2 对应 19 行bytecode程序实例的证明约 1500，其中coq编码的程序代码为 34 行，程序断言 46 行，证明约 1300 行。对比字节码验证和以往汇编代码验证，我们发现，字节码程序本身代码与证明之间的比率与汇编码及其证明之间的比率基本相当。然而，字节码相比于汇编代码具有更紧凑的形式，大多数JAVA字节码到汇编指令编译器的翻译比例约 5-10[24]。可以大致估算，采用本文的逻辑系统对字节码程序进行验证，而不需要直接证明相应的汇编代码，工作量大约降低 5-10 倍。

3.4 未来扩展研究

本逻辑针对的是控制栈和非结构化控制流，因此所定义的字节码子集目前只包含其中的关键指令，后续将扩展支持更多的语言特征。系统首先要扩展的是对象、引用、方法和继承等面向对象特征，以便更好地体现当今字节码程序的发展趋势和使用情况。另外，提供例外处理支持是另一个直接的扩展工作，采用类似于函数调用/返回的思路，加入例外支持应该比较容易。并发程序验证支持是又一个值得深入探讨的热点问题，随着多核处理器的发展而更加突出，即便是广泛使用的并发程序库，比如JDK 同步类等，也存在一系列缺陷[25]。相比而言，并发支持的机器定义比较容易，而针对并发字节码程序的验证逻辑，就不那么轻而易举了。此外，证明过程中引入更多的自动技术，以提高验证效率，也是一个需要深入研究的方向。

4 结束语

相关工作。底层语言栈式控制结构的模块化验证是一个相当困难的问题。Feng等人的SCAP系统[9]提出(p,g)断言和基于调用层数的良型栈（WFST）递规定义，首次在汇编语言层面很好地解决了这个难题，本文则给出了字节码层面的解答。用于字节码程序验证的逻辑系统的代表性研究工作中，Quigley设计了一个用于字节码程序的类Hoare逻辑系统证明了含有循环的程序片段[15]，较好地解决了循环结构重建问题，却无法解决模块化验证问题；Benton的组合一阶逻辑系统针对类.NET CIL命令式语言的虚拟机，该虚拟机采用单独的函数调用栈来支持程序的模块化证明，文中只给出几个代码片段的纯手工证明[18]。SCAP论文针对真实的MIPS机器，而本文所定义的虚拟机BVM具有更高的抽象层次，采用计算栈和函数调用栈分离的双栈结构，其结构类似Benton文中的虚拟机。在借鉴SCAP采用高阶逻辑描述程序断言(p,g)和递规定义良型栈思路的基础

上，本文重新定义了针对BVM的前条件p，进一步给出全新的基于抽象函数调用栈的良型栈递规定义，此外，本文深入讨论函数调用关系的描述，给出一种子程序断言的定义方式 $g_{fun} = ((g_{call} \circ g') \circ g_{ret})$，该定义将具体调用约定隐藏在 g_{call} 和 g_{ret} 中，得到更为通用的函数调用关系描述方案。因此，和SCAP相比，CBP是针对更高抽象机器模型的通用、简洁的逻辑系统；和Benton的系统相比，CBP具有更强的描述能力，并实现证明的机器自动检查。由于这些特性，CBP系统将非常适合用于构建一个字节码到汇编语言的证明保持编译器。

结论。本文提出一种逻辑系统，用于运行于栈式虚拟机的字节码程序验证，该字节码的操作语义定义类似于 JAVA 字节码和.NET CIL，软件模块中可以包含复杂栈式控制结构和非结构化控制流，本逻辑系统很好地解决了字节码软件的模块化验证问题。利用辅助工具 Coq，证明了包括逻辑系统合理性和几个实例程序，所有证明均可机器自动检查。

本文工作为栈式虚拟机上运行的字节码验证提供一个良好的基础，同时也向着构造证明保持编译器迈出了坚实的一步，还能够为广泛使用的一类复杂网络应用程序的深刻理解和深入分析提供理论帮助。

致谢 在此,我们向对本文的工作给予建议的同行,特别是 Yale 大学计算机系 Zhong Shao 教授、TTI-Chicago 计算机系 Xinyu Feng 博士、Lehigh 大学计算机的 Gang Tan 博士等人表示感谢。

References:

[1] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. In Proc. 2003 International Conference on Compiler Construction (CC'03), LNCS Vol. 2622, pages 262–272, Warsaw, Poland, Springer-Verlag Heidelberg, 2003.

[2] M. Hall, D. Padua, K. Pingali, Compiler research: the next 50 years, Communications of the ACM, Vol. 52, No. 2, 60-67, 2009.

[3] T. Lindholm and F. Yellin. The java virtual machine specification (second edition), 1999.

[4] ECMA. Standard ECMA-335 Common Language Infrastructure. 2006.

[5] Xavier Leroy. A formally verified compiler back-end. draft, 2008. http://pauillac.inria.fr/~xleroy/publi/compcert-backend.pdf.

[6] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In Proc. Conf. Programming Language Design and Implementation，pp. 333-344. ACM SIGPLAN，1998.

[7] A. W. Appel. Foundational proof-carrying code. In Proc. 16th IEEE Symposium on Logic in Computer Science, pages 247–258. IEEE Computer Society, June 2001.

[8] Dachuan Yu, Nadeem A. Hamid, and Zhong Shao, Building Certified Libraries for PCC: Dynamic Storage Allocation, In Science of Computer Programming, 50 (1-3):101-127, 2004.

[9] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In Prog. Lang. Design and Impl. (PLDI'06), pages 401–414, New York, NY, USA, June 2006. ACM Press.

[10] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified Self-Modifying Code. In Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07), San Diego, CA, pages 66-77, June 2007.

[11] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In Prog. Lang. Design and Impl. (PLDI'08), pages 170–182, New York, NY, USA, June 2008. ACM Press.

[12] Shengyuan Wang, Yuan Dong, A Verifiable Low-level Concurrent Programming Model Based on Colored Petri Nets, the Proceedings of Petri Nets and Distributed Systems 2008 (workshoip of 29th ATPN conference), Xi'an, China, June 23-24, 2008

[13] Yunmin Zhu, Liwei Zhang, Shengyuan Wang, Yuan Dong and Suqin Zhang, Verifying Parallel Low-level Programs for Multi-core Processor, In Proc. NASAC'08, 282-287, 2008.

[14] Xavier Leroy. Bytecode verification for Java smart card. Software Practice & Experience, 32:319-340, 2002.

[15] C. L. Quigley. A programming logic for java bytecode programs. In Proc. of 16th Int. Conf. on Theorem Proving in Higher-Order Logics, TPHOLs 2003, pages 41–54. Springer-Verlag, 2003.

[16] D. Aspinall, L. Beringer,M. Hofmann, H.-W. Loidl, and A.Momigliano. A program logic for resource verification. In TPHOLs'04. Springer-Verlag, 2004.

[17] F. Bannwart and P. Muller. A program logic for bytecode. In Proceedings of Bytecode05, Electronic Notes in Theoretical Computer Science, pages 255–273. Elsevier, 2005.

[18] N. Benton. A typed, compositional logic for a stack-based abstract machine. In In Proc. 3rd Asian Symposium on Programming Languages and Systems (APLAS), volume 3780 of LNCS, pages 364–380. Springer-Verlag, 2005.

[19] L. Burdy and M. Pavlova. Java bytecode specification and verification. In Proceedings of SAC06. ACM Press, 2006.

[20] Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.1, 2006.

[21] http://soft.cs.tsinghua.edu.cn/dongyuan/~dongyuan/verify/cbp.html

[22] C. A. R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 26(1):53–56, Oct. 1969.

[23] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In Proc. 17th Annual IEEE Symp. on Logic in Comp. Sci. (LICS'02), pages 55–74. IEEE Computer Society, July 2002.

[24] M.Weiss, F. de Ferrire, B. Delsart, C. Fabre, F. Hirsch, E. A. Johnson, V. Joloboff, F. Roy, F. Siebert, and X. Spengler. Turboj, a java bytecode-to-native compiler. In Proc. LCTES98, volume 1474 of LNCS, pages 119–130. Springer-Verlag, 1998.

[25] K. Sen. Race directed randomized dynamic analysis of concurrent programs. In Proc. 2008 ACM Conf. on Prog. Lang. Design and Impl., pages 11–21. ACM Press, June 2008.

附中文参考文献:

[13] 朱允敏，张丽伟，王生原，董渊，张素琴. 面向多核处理器的低级并行程序验证. 全国软件与应用学术会议, 282-287, 2008

[26] 陈意云、华保健、葛琳、王志芳，一种用于指针程序安全性证明的指针逻辑，计算机学报，31(3)，pp.372-380，2008.3。

[27] 郭宇、陈意云、林春晓，一种构造代码安全性证明的方法，软件学报，2008.10，19(10)，pp.2720-2727。