Modular Certification of Bytecode Programs

Yuan Dong[†] Shengyuan Wang[†] Liwei Zhang[†] Yunmin Zhu[†] Ping Yang[‡]

[†]Tsinghua University [‡]Beijing Language and Culture University

{dongyuan, wwssyy}@tsinghua.edu.cn {zhanglw06, zhuym06}@mails.tsinghua.edu.cn yangp@blcu.edu.cn

Abstract

Bytecode which runs on stack-based virtual machine, such as .NET CIL and JAVA bytecode, is the key technology for hardware- and operating system-independence. Although some efforts have been made on building logic system for bytecode programs, modular certification of bytecode still remains challenging because of the complexity of abstract control stack and the lack of control flow structure information.

To tackle these challenges, this paper presents a logic framework to verify bytecode programs. We define A Hoare-logic like system for a bytecode language similar to JAVA bytecode and CIL. This logic can fully support abstract control stack and unstructured control flow for modular certification of bytecode programs. This system is expressive and fully mechanized. We prove its soundness and demonstrate its power by certifying some bytecode programs in the Coq proof assistant. This work not only provides a solid theoretical foundation for reasoning about bytecode, but also makes an important advance toward building a certified proof-preserving compiler.

1. Introduction

Bytecode, such as JAVA bytecode [12] and .NET Common Intermediate Language (CIL) [8], is the key component of the technology responsible for hardware- and operating system- independence. It is the base to protect users from malicious programs. They can run on a stack-based virtual machine directly or can be translated into native machine codes by another compiler.

Why Bytecode? Formal reasoning about bytecode programs is required both for trustworthy web application building and for proof-transforming compilation. Many kinds of program logic for reasoning about high-level language have been proposed [6]. Can we find a way carrying or transforming these proofs to mid-level languages, like bytecode, and checking them just before running? Besides, following Foundational Proof-Carrying Code (FPCC) idea, machine level programs involving the features such as control stack and interrupt based concurrency can be certified with proper logic system [10, 9]. However, the proofs of any non-trivial programs are really hard to carry out manually. Can we obtain the proofs from a much higher level such as bytecode?

Major Challenges. To formally certify bytecode programs, a program logic for virtual machine is required to support both runtime stacks and unstructured control flow.

Runtime stacks are critical components of any modern software. Correct implementation of these constructs is of utmost importance to the safety and reliability of any bytecode programs. Stack operations are very complex and difficult to reason about because they involve subtle low-level invariants: return code pointers should have restricted scopes, making it even harder to track their lifetime.

To overcome the lack of structures in low-level code, many proof-carrying code (PCC) systems [1, 13] support stack-based controls by using continuation-passing style (CPS) which treats return addresses as first-class code pointers. It is a general semantic model to support the control abstractions above, but CPS-based reasoning requires specification of continuation pointers using "impredicative types" [13, 14]), which makes the program specification complex and hard to understand. It is also difficult to specify first-class code pointers modularly in logic: because of the circular references between code pointers and data heap (which may in turn contains code pointers).

SCAP, a simple but flexible Hoare-style framework for modular verification of assembly code with all kinds of stack-based control abstractions was presented by Feng *et al.* [10]. SCAP does follow a much simpler pattern instead of the general first-class code pointers to handle stack-based control abstractions.

Benton [4] proposed a typed, compositional logic for a stack-based abstract machine to support bytecode programs modular reasoning about. To protect return code pointers, he uses a JVM-like higher-level abstract machine with separate data stack and control stack; the latter cannot be touched by regular instructions except for call and ret. Although some efforts [15, 3] on building logic system for bytecode programs have been make, the task still remains challenging because of the complexity of abstract control stack and the lack of control flow structure information due to the flat nature of bytecode programs. The major challenges are:

- Fully stack control support in a simple way for bytecode. Can we deal with it in a natural and general way to support modular certifying as that have been done for low-level code?
- Pervasive logic framework. Is the idea of logic system for assembly code certification applicable to bytecode programs for stack-based virtual machine? How can we link certified bytecode program and certified VM together? Is it feasible to build a logic framework in which the proofs and the semantics of bytecode programs can be preserved during execution.

Our Contributions. In this paper we present a novel Hoare-logic like framework for certifying bytecode programs (named *CBP*) that supports modular verification of bytecode programs with all kinds of stack-based control abstractions and unstructured control flow.

This system is fully mechanized. We give the complete soundness proof and a full verification of an example in the Coq proof assistant [7]. This program logic applies FPCC concepts to bytecode programs for partial correctness properties verification.

Building upon previous work on program verification, we make the following contributions:

- As far as we know, our work presents the first program logic facility for modularly certifying the partial correctness of bytecode programs. This logic specifies an invariant at each program point using a pair of a precondition and a "local" guarantee (which states the obligation that the current function must

$$\begin{array}{rlll}
(\mathit{World}) & \mathbb{W} & ::= & (\mathbb{C}, \mathbb{S}, \mathbb{K}_c, \mathtt{pc}) \\
(\mathit{CodeHeap}) & \mathbb{C} & ::= & \{\mathtt{f} \rightsquigarrow \mathbb{I}\}^* \\
(\mathit{State}) & \mathbb{S} & ::= & (\mathbb{H}, \mathbb{K}) \\
(\mathit{ProgCnt}) & \mathtt{pc} & ::= & n \ \ (\mathit{nat\ nums}) \\
(\mathit{Memory}) & \mathbb{H} & ::= & \{\mathtt{k} \rightsquigarrow \mathtt{w}\}^* \\
(\mathit{EStack}) & \mathbb{K} & ::= & \mathsf{nil} \mid \mathtt{w} :: \mathbb{K} \\
(\mathit{CStack}) & \mathbb{K}_c & ::= & \mathsf{nil} \mid \mathtt{f} :: \mathbb{K}_c \\
(\mathit{Labels}) & \mathtt{f}, \mathtt{k} & ::= & n \ \ (\mathit{nat\ nums}) \\
(\mathit{Word}) & \mathtt{w} & ::= & i \ \ (\mathit{integers}) \\
(\mathit{Instr}) & \iota & ::= & \mathsf{pushc}\ \mathtt{w} \mid \mathsf{pushv}\ \mathtt{k} \mid \mathsf{pop}\ \mathtt{k} \mid \mathsf{binop}\ \mathtt{m} \mid \mathsf{unop}\ \mathtt{m} \\
 & & & \mid \mathsf{brtrue}\ \mathtt{f} \mid \mathsf{call}\ \mathtt{f} \\
(\mathit{Commd}) & \mathtt{c} & ::= & \iota \mid \mathsf{ret} \mid \mathsf{goto}\ \mathtt{f} \\
(\mathit{InstrSeq}) & \mathbb{I} & ::= & \iota; \mathbb{I} \mid \mathsf{ret} \mid \mathsf{goto}\ \mathtt{f} \\
(\mathit{OprNum}) & \mathtt{m} & ::= & \{+ \ldots /, - \cdots +\}
\end{array}$$

Figure 1. Definition of A Bytecode Machine

$$\mathbb{C}[\mathtt{f}] \triangleq \begin{cases} \mathtt{c} & \mathtt{c} = \mathbb{C}(\mathtt{f}) \text{ and } \mathtt{c} = \mathsf{goto}\ \mathtt{f}', \text{ or ret} \\ \iota; \mathbb{I} & \iota = \mathbb{C}(\mathtt{f}) \text{ and } \mathbb{I} = \mathbb{C}[\mathtt{f}+1] \end{cases}$$

$$(F\{a \rightsquigarrow b\})(x) \triangleq \begin{cases} b & \mathit{if}\ x = a \\ F(x) & \mathit{otherwise}\ . \end{cases}$$

Figure 2. Definition of Representations

fulfill before it can return). These guarantees, when chained together, are used to specify the logical control stack. Soundness of our framework is formally proved in the Coq proof assistant.

- This logic framework is also, to our best knowledge, the first taste to extend FPCC concepts which is powerful for machine code certification to mid-level bytecode language. Our experience demonstrates that this kind of Hoare-logic framework is applicable to bytecode programs. As we know, an interpreter is similar to code generator of a compiler. So, it can be seen that this approach is a feasible way to build a pervasive Hoare-like logic framework for proof and semantics preserving compilation from bytecode to machine code.

This work not only provides a foundation for reasoning about bytecode programs, but also makes an important step toward building an environment in which verified mid-level bytecode programs with their proofs can be transformed to machine code with semantics and proof preservation.

2. *CBP* Logic for ByteCode Virtual Machine

In this section, we first present the formal definition and the operational semantics of our bytecode machine *BVM*. Then, we give the program logic *CBP* for certifying bytecode programs.

2.1 Bytecode Machine Definition

In Figure 1, we show *BVM* for our BC/0. The whole machine configuration is called a "World" ($\mathbb{W}$), which consists of a read-only code heap (C), an updatable state ($\mathbb{S}$), a function call stack ($\mathbb{K}_c$), and a program counter ($\mathtt{pc}$). The code heap is a finite partial mapping from code labels ($\mathtt{f}$) to instruction sequences ($\mathbb{I}$). The state $\mathbb{S}$ contains a memory heap ($\mathbb{H}$) and an evaluation stack ($\mathbb{K}$). The program counter $\mathtt{pc}$ points to the current command in $\mathbb{C}$. We define the instruction sequence $\mathbb{I}$ as a sequence of sequential instructions ending with jump or return commands. $\mathbb{C}[\mathtt{f}]$ extracts an instruction sequence starting from $\mathtt{f}$ in $\mathbb{C}$, as defined in Figure 2. We use the dot notation to represent a component in a tuple, *e.g.,* $\mathbb{S}.\mathbb{K}$ means the stack in state $\mathbb{S}$. We also use function $\mathtt{top}()$ and $\mathtt{max}()$ to get the current pointers and the upper bounds of stack $\mathbb{K}$, $\mathbb{K}_c$. Valid $\mathbb{K}$ or $\mathbb{K}_c$ means that current pointer $\mathtt{top}()$ is in domain $[0, \mathtt{max}()]$ and points to some value.

2.2 The *BVM* Operational Semantics

In Figure 3, we also define the machine configuration transition operational semantics of each instruction in a formal way. Here

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

if $\mathtt{c} =$	if $\mathtt{Enable}(\mathtt{c})\ \mathbb{K}_c\ \mathbb{S} =$	then $\mathbb{S}' =$
pushc $\mathtt{w}$	$\mathtt{valid}(\mathbb{K})$	$(\mathbb{H}, \mathtt{w} :: \mathbb{K})$
pushv $\mathtt{f}$	$\mathtt{valid}(\mathbb{K})$ and $\mathbb{H}(\mathtt{f}) = \mathtt{w}$	$(\mathbb{H}, \mathtt{w} :: \mathbb{K})$
pop $\mathtt{f}$	$\mathbb{K} = \mathtt{w} :: \mathbb{K}'$	$(\mathbb{H}\{\mathtt{f} \rightsquigarrow \mathtt{w}\}, \mathbb{K}')$
binop *bop*	$\mathbb{K} = \mathtt{w}_1 :: \mathtt{w}_2 :: \mathbb{K}', \mathtt{w} = \mathit{bop}(\mathtt{w}_1, \mathtt{w}_2)$	$(\mathbb{H}, \mathtt{w} :: \mathbb{K}')$
unop *uop*	$\mathbb{K} = \mathtt{w}_1 :: \mathbb{K}', \mathtt{w} = \mathit{uop}(\mathtt{w}_1)$	$(\mathbb{H}, \mathtt{w} :: \mathbb{K}')$
brtrue $\mathtt{f}$	$\mathbb{K} = \mathtt{w} :: \mathbb{K}', \mathtt{w} =$ True or False	$(\mathbb{H}, \mathbb{K}')$
call $\mathtt{f}$	$\mathtt{valid}(\mathbb{K}_c)$	$(\mathbb{H}, \mathbb{K})$
ret	$\mathbb{K}_c = \mathtt{f} :: \mathbb{K}_c{}'$	$(\mathbb{H}, \mathbb{K})$
...		$(\mathbb{H}, \mathbb{K})$

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

if $\mathtt{c} =$	if $\mathtt{Enable}(\mathtt{c})\ \mathbb{K}_c\ \mathbb{S} =$	then $\mathbb{K}_c{}' =$
call $\mathtt{f}$	$\mathtt{valid}(\mathbb{K}_c)$	$(\mathtt{pc}+1) :: \mathbb{K}_c)$
ret	$\mathbb{K}_c = \mathtt{f} :: \mathbb{K}_c{}'$	$\mathbb{K}_c{}'$
...	...	$\mathbb{K}_c$

$\mathtt{NextPC}_{(\mathtt{c},\mathbb{S},\mathbb{K}_c)}\ \mathtt{pc}\ \mathtt{pc}'$ where $\mathbb{S} = (\mathbb{H}, \mathbb{K})$

if $\mathtt{c} =$	if $\mathtt{Enable}(\mathtt{c})\ \mathbb{K}_c\ \mathbb{S} =$	then $\mathtt{pc}' =$
brtrue $\mathtt{f}$	$\mathbb{K} = \mathtt{w} :: \mathbb{K}', \mathtt{w} =$ True	$\mathtt{f}$
	$\mathbb{K} = \mathtt{w} :: \mathbb{K}', \mathtt{w} =$ False	$\mathtt{pc}+1$
call $\mathtt{f}$	$\mathtt{valid}(\mathbb{K}_c)$	$\mathtt{f}$
ret	$\mathbb{K}_c = \mathtt{f} :: \mathbb{K}_c{}'$	$\mathtt{f}$
goto $\mathtt{f}$		$\mathtt{f}$
...	...	$\mathtt{pc}+1$

$$\frac{\begin{array}{l}\mathtt{c} = \mathbb{C}(\mathtt{pc}) \quad \mathtt{Enable}(\mathtt{c})\ \mathbb{K}_c\ \mathbb{S} \\ \mathtt{NextS}_{(\mathtt{c},\mathtt{pc},\mathbb{K}_c)}\ \mathbb{S}\ \mathbb{S}' \quad \mathtt{NextKc}_{(\mathtt{c},\mathtt{pc},\mathbb{S})}\ \mathbb{K}_c\ \mathbb{K}_c{}' \quad \mathtt{NextPC}_{(\mathtt{c},\mathbb{S},\mathbb{K}_c)}\ \mathtt{pc}\ \mathtt{pc}'\end{array}}{(\mathbb{C}, \mathbb{S}, \mathbb{K}_c, \mathtt{pc}) \longmapsto (\mathbb{C}, \mathbb{S}', \mathbb{K}_c{}', \mathtt{pc}')} \ (\mathrm{PC})$$

Figure 3. operational semantics of *BVM*

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

Figure 4. Stack-Based Bytecode Program

$\mathtt{Enable}(\mathtt{c})\ \mathbb{K}_c\ \mathbb{S}$ gives the weakest condition for instruction $\mathtt{c}$ to execute. The relation $\mathtt{NextS}_{(\mathtt{c},\mathtt{pc},\mathbb{K}_c)}$ shows the transition of states by executing $\mathtt{c}$ with program counter $\mathtt{pc}$ and call stack $\mathbb{K}_c$. While $\mathtt{NextPC}_{(\mathtt{c},\mathbb{S},\mathbb{K}_c)}$ shows how $\mathtt{pc}$ changes after $\mathtt{c}$ is executed with $\mathbb{S}$ and $\mathbb{K}_c$. $\mathtt{NextKc}_{(\mathtt{c},\mathtt{pc},\mathbb{S})}$ gives the $\mathbb{K}_c$ changes after $\mathtt{c}$ execution with program counter $\mathtt{pc}$ and $\mathbb{S}$.

The instruction set captures the most basic and common *BVM*, which is similar to JAVA bytecode or .NET CIL. Semantics of most instructions are straightforward.

The execution of programs is modeled as a small-step transition from one world to another. $\mathbb{W} \longmapsto \mathbb{W}'$ made by executing the instruction pointed to by $\mathtt{pc}$.

A BC/0 bytecode program which involving while loop control structure for *BVM* is shown in Figure 4. It can be compiled from the following source code.

$$\begin{array}{rrcl}
(\mathit{Pred}) & \mathtt{p} & \in & \mathit{CStack} \rightarrow \mathit{State} \rightarrow \mathit{Prop} \\
(\mathit{Guarantee}) & \mathtt{g} & \in & \mathit{State} \rightarrow \mathit{State} \rightarrow \mathit{Prop} \\
(\mathit{Spec}) & \mathtt{s} & ::= & (\mathtt{p}, \mathtt{g}) \\
(\mathit{CdHpSpec}) & \Psi & ::= & \{(\mathtt{f}_1, \mathtt{s}_1), \ldots, (\mathtt{f}_n, \mathtt{s}_n)\} \\
(\mathit{MPred}) & \mathtt{m} & \in & \mathit{Memory} \rightarrow \mathit{Prop}
\end{array}$$

Figure 5. Specification Constructs for *CBP*

$$\begin{array}{l}
\mathsf{True} \triangleq \lambda\mathbb{H}.\ \mathsf{True} \qquad\qquad \mathsf{emp} \triangleq \lambda\mathbb{H}.\ \mathbb{H} = \varnothing \\
l \mapsto \mathtt{w} \triangleq \lambda\mathbb{H}.\ \mathbb{H} = \{l \rightsquigarrow \mathtt{w}\} \quad l \mapsto \_ \triangleq \lambda\mathbb{H}.\ \exists\mathtt{w}.\ (l \mapsto \mathtt{w})\ \mathbb{H} \\
\mathbb{H}_1 \sharp \mathbb{H}_2 \triangleq \mathit{dom}(\mathbb{H}_1) \cap \mathit{dom}(\mathbb{H}_1) = \varnothing \\
\mathbb{H}_1 \uplus \mathbb{H}_2 \triangleq \begin{cases} \mathbb{H}_1 \cup \mathbb{H}_2 & \text{if } \mathbb{H}_1 \sharp \mathbb{H}_2 \\ \mathit{undefined} & \text{otherwise} \end{cases} \\
\mathtt{m}_1 * \mathtt{m}_2 \triangleq \lambda\mathbb{H}.\ \exists\mathbb{H}_1, \mathbb{H}_2.\ (\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{H}) \wedge \mathtt{m}_1\ \mathbb{H}_1 \wedge \mathtt{m}_2\ \mathbb{H}_2 \\
\mathtt{p} * \mathtt{m} \triangleq \lambda\mathbb{S}.\ \exists\mathbb{H}_1, \mathbb{H}_2.\ (\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{S}.\mathbb{H}) \wedge \mathtt{p}\ \mathbb{S}|_{\mathbb{H}_1} \wedge \mathtt{m}\ \mathbb{H}_2
\end{array}$$

Figure 6. Definitions of "Separation Logic" Assertions

```
int factor(){
  r = 1;
  while(n != 0){ r = r*n; n = n-1; }
}
```

You can omit the contents in the shadow box here. We will discuss them more details in the next section. To certify a program like this, the challenge is to formalize and capture the invariant.

2.3 Specification Language

We use the mechanized *meta-logic* which is implemented in the Coq proof assistant [7] as our specification language. The logic corresponds to a higher-order predicate logic with inductive definitions. To specify a program with code heap $\mathbb{C}$, the programmer must insert specifications $\mathtt{s}$ at instruction sequence start points, see Figure 4. As shown in Figure 5, the specification $\mathtt{s}$ is a pair $(\mathtt{p}, \mathtt{g})$. The assertion $\mathtt{p}$ is a predicate over function call stack $\mathbb{K}_c$ and program state $\mathbb{S}$ (its meta-type in Coq is a function that takes $\mathbb{K}_c$ and $\mathbb{S}$ as argument and returns a proposition), while guarantee $\mathtt{g}$ is a predicate over two program states.

As we can see, the $\mathtt{Enable}(\mathtt{c})$ defined in Figure 3 is a special $\mathtt{p}$. And the $\mathtt{NextS}_{(\mathtt{c}, \mathtt{pc})}$ relation is a special form of $\mathtt{g}$ which is over the two adjacent states. We use $\mathtt{p}$ to specify the precondition over function call stack, memory heap and stack. And use $\mathtt{g}$ to specify the guaranteed behavior from the specified program point to the point when the *current* function returns.

Specification Ψ for code heap $\mathbb{C}$ associates code labels $\mathtt{f}$ with corresponding $\mathtt{s}$. Note that multiple $\mathtt{s}$ may be associated with the same $\mathtt{f}$, just as a function may have multiple specified interfaces.

We use the predicate $\mathtt{m}$ to specify the memory heap. To enforce the memory heap partition between different functions, we encode in Figure 6 Separation Logic connectors in our specification language (which is also our meta-logic). We also use standard separation logic primitives [16] as assertion operators. The definitions are self-evident and we do not explain the details here.

2.4 The CBP Program Logic

We use the following judgments to define the inference rules:

$\Psi \vdash \{\mathtt{s}\}\mathbb{W}$	(well-formed world)
$\Psi \vdash \mathbb{C} : \Psi'$	(well-formed code heap)
$\Psi \vdash \{\mathtt{s}\}\mathbb{I}$	(well-formed instruction sequence)

Inference rules of the program logic are shown in Figure 7.

Program Invariants. The WLD rule formulates the program invariant enforced by our program logic:

- The code heap $\mathbb{C}$ needs to be well-formed follow the CDHP rule.

$\boxed{\Psi \vdash \{\mathtt{s}\}\mathbb{W}}$ ***(Well-formed World)***

$$\frac{\Psi \vdash \mathbb{C} : \Psi' \quad \Psi \subseteq \Psi' \quad \Psi \vdash \{\mathtt{s}\}\mathtt{pc} : \mathbb{C}[\mathtt{pc}] \quad \{\mathtt{s}\}\,\Psi'\,\mathbb{S}}{\Psi \vdash \{\mathtt{s}\}\,(\mathbb{C}, \mathbb{S}, \mathtt{pc})} \quad (\textsc{wld})$$

$\boxed{\Psi \vdash \mathbb{C} : \Psi'}$ ***(Well-formed Code Heap)***

$$\frac{\text{for all } (\mathtt{f}, \mathtt{s}) \in \Psi' : \quad \Psi \vdash \{\mathtt{s}\}\mathtt{f} : \mathbb{C}[\mathtt{f}]}{\Psi \vdash \mathbb{C} : \Psi'} \quad (\textsc{cdhp})$$

$$\frac{\Psi_1 \vdash \mathbb{C}_1 : \Psi'_1 \quad \Psi_2 \vdash \mathbb{C}_2 : \Psi'_2 \quad \mathbb{C}_1 \# \mathbb{C}_2}{\Psi_1 \cup \Psi_2 \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi'_1 \cup \Psi'_2} \quad (\textsc{link})$$

$\boxed{\Psi \vdash \{\mathtt{s}\}\mathbb{I}}$ ***(Well-formed Instr. Sequence)***

$$\frac{\begin{array}{c} \iota \notin \{\mathsf{brtrue}\ , \mathsf{call}\ \} \qquad \Psi \vdash \{(\mathtt{p}'', \mathtt{g}'')\}\mathtt{pc}{+}1 : \mathbb{I} \\ \mathtt{p} \Rightarrow \mathtt{g}_\iota \quad (\mathtt{p} \triangleright \mathtt{g}_\iota) \Rightarrow \mathtt{p}'' \quad (\mathtt{p} \circ (\mathtt{g}_\iota \circ \mathtt{g}'')) \Rightarrow \mathtt{g} \end{array}}{\Psi \vdash \{(\mathtt{p}, \mathtt{g})\}\mathtt{pc} : \iota; \mathbb{I}} \quad (\textsc{seq})$$

$$\frac{\begin{array}{ll} (\mathtt{f}', (\mathtt{p}', \mathtt{g}')) \in \Psi & \Psi \vdash \{(\mathtt{p}'', \mathtt{g}'')\}\mathtt{pc}{+}1 : \mathbb{I} \\ (\mathtt{p} \triangleright \mathtt{g}_{\mathsf{brtureT}}) \Rightarrow \mathtt{p}' & (\mathtt{p} \circ (\mathtt{g}_{\mathsf{brtureT}} \circ \mathtt{g}')) \Rightarrow \mathtt{g} \\ (\mathtt{p} \triangleright \mathtt{g}_{\mathsf{brtureF}}) \Rightarrow \mathtt{p}'' & (\mathtt{p} \circ (\mathtt{g}_{\mathsf{brtureF}} \circ \mathtt{g}'')) \Rightarrow \mathtt{g} \end{array}}{\Psi \vdash \{(\mathtt{p}, \mathtt{g})\}\mathtt{pc} : \mathsf{brtrue}\ \mathtt{f}'; \mathbb{I}} \quad (\textsc{brture})$$

$$\frac{\begin{array}{l} (\mathtt{pc}{+}1, (\mathtt{p}'', \mathtt{g}'')) \in \Psi \qquad \Psi \vdash \{(\mathtt{p}'', \mathtt{g}'')\}\mathtt{pc}{+}1 : \mathbb{I} \\ (\mathtt{p} \triangleright \mathtt{g}_{\mathsf{call}}) \Rightarrow \mathtt{p}' \quad (\mathtt{p} \triangleright \mathtt{g}_{\mathsf{fun}}) \Rightarrow \mathtt{p}'' \quad (\mathtt{p} \circ (\mathtt{g}_{\mathsf{fun}} \circ \mathtt{g}'')) \Rightarrow \mathtt{g} \\ (\mathtt{f}', (\mathtt{p}', \mathtt{g}')) \in \Psi \qquad \mathtt{g}_{\mathsf{fun}} = ((\mathtt{g}_{\mathsf{call}} \circ \mathtt{g}') \circ \mathtt{g}_{\mathsf{ret}}) \end{array}}{\Psi \vdash \{(\mathtt{p}, \mathtt{g})\}\mathtt{pc} : \mathsf{call}\ \mathtt{lf}'; \mathbb{I}} \quad (\textsc{call})$$

$$\frac{(\mathtt{p} \circ \mathtt{g}_{\mathsf{ret}}) \Rightarrow \mathtt{g}}{\Psi \vdash \{(\mathtt{p}, \mathtt{g})\}\mathtt{pc} : \mathsf{ret}} \quad (\textsc{ret})$$

$$\frac{(\mathtt{f}', (\mathtt{p}', \mathtt{g}')) \in \Psi \quad (\mathtt{p} \triangleright \mathtt{g}_{\mathsf{goto}}) \Rightarrow \mathtt{p}' \quad (\mathtt{p} \circ (\mathtt{g}_{\mathsf{goto}} \circ \mathtt{g}')) \Rightarrow \mathtt{g}}{\Psi \vdash \{(\mathtt{p}, \mathtt{g})\}\mathtt{pc} : \mathsf{goto}\ \mathtt{f}'} \quad (\textsc{goto})$$

Figure 7. CBP Inference Rules

- The imported interface Ψ is a subset of the exported interface Ψ', therefore $\mathbb{C}$ is self-contained and each imported specification has been certified.
- Current $\mathtt{pc}$ has a specification $\mathtt{s}$ in Ψ, thus the current instruction sequence $\mathbb{C}[\mathtt{pc}]$ is well-formed with respect to $\mathtt{s}$.
- Given exported Ψ', the current state $\mathbb{S}$ satisfies the assertion $\mathtt{s}$.

Program Modules. In the CDHP rule, Ψ contains specifications for external code (imported by the local module $\mathbb{C}$), while Ψ' contains specifications for code blocks in the module $\mathbb{C}$ for other modules. Thus, the *CBP* logic supports *separate verification* of program modules. Modules are modeled as small code heaps which contain at least one code block. The specification of a module contains not only specifications of the code blocks in the current module, but also specifications of external code blocks which will be called by this module. The well-formedness of each individual module is established via the CDHP rule. Then, two non-intersecting well-formed modules can then be linked together via the LINK rule. The WORLD rule requires that all modules be linked into a well-formed global code heap.

Sequential Instructions. Like traditional Hoare-logic [11], our logic also uses the pre- and postcondition as specifications for programs. The SEQ rule is a *schema* for instruction sequences starting with an instruction ι (ι cannot be conditional jump or function call instructions). It says it is safe to execute the instruction sequence $\mathbb{I}$ starting at the code label $\mathtt{pc}$, given the imported interface in Ψ and a precondition $(\mathtt{p}, \mathtt{g})$. An intermediate specification $(\mathtt{p}'', \mathtt{g}'')$ with respect to which the remaining instruction sequence is well-formed should be found. It is also used as a post-condition for the current instruction ι. We use $\mathtt{g}_\iota$ to represent the state transition made by the instruction ι, which is defined in Figure 9 and Figure 3. Since $\mathtt{NextS}$ does not depend on the current program counter for these instructions "_" is used to represent arbitrary $\mathtt{pc}$.

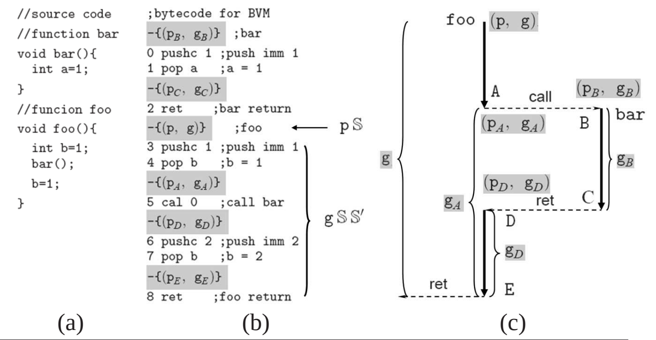
The definitions in Figure 8 are used in these rules. The predicate $\mathtt{p} \triangleright \mathtt{g}_\iota$ specifies the state resulting from the state transition $\mathtt{g}_\iota$, knowing the initial state satisfies $\mathtt{p}$. It is the strongest post condition

$$
\begin{array}{llll}
\mathtt{p} \Rightarrow \mathtt{g} & \triangleq \forall \mathbb{S}.\, \mathtt{p}\,\mathbb{S} \rightarrow \exists \mathbb{S}', \mathtt{g}\,\mathbb{S}\,\mathbb{S}' & \mathtt{p} \triangleright \mathtt{g} & \triangleq \lambda \mathbb{S}.\, \exists \mathbb{S}_0, \mathtt{p}\,\mathbb{S}_0 \wedge \mathtt{g}\,\mathbb{S}_0\,\mathbb{S} \\
\mathtt{g} \circ \mathtt{g}' & \triangleq \lambda \mathbb{S}, \mathbb{S}''.\, \exists \mathbb{S}'.\, \mathtt{g}\,\mathbb{S}\,\mathbb{S}' \wedge \mathtt{g}'\,\mathbb{S}'\,\mathbb{S}'' & \mathtt{p} \Rightarrow \mathtt{p}' & \triangleq \forall \mathbb{S}.\, \mathtt{p}\,\mathbb{S} \rightarrow \mathtt{p}'\,\mathbb{S} \\
\mathtt{g} \Rightarrow \mathtt{g}' & \triangleq \forall \mathbb{S}, \mathbb{S}'.\, \mathtt{g}\,\mathbb{S}\,\mathbb{S}' \rightarrow \mathtt{g}'\,\mathbb{S}\,\mathbb{S}' & \mathtt{p} \circ \mathtt{g} & \triangleq \lambda \mathbb{S}, \mathbb{S}'.\, \mathtt{p}\,\mathbb{S} \wedge \mathtt{g}\,\mathbb{S}\,\mathbb{S}'
\end{array}
$$

Figure 8. Connectors for p and g

$$
\begin{array}{lll}
\mathtt{g}_{\mathsf{brtureT}} & \triangleq \lambda \mathbb{S}, \mathbb{S}'.\mathtt{NextS}_{(\mathsf{brture},\_)}\,\mathbb{S}\,\mathbb{S}' & (\text{where } \mathbb{S}.\mathbb{K} = w :: \mathbb{K}', w = \mathsf{True}) \\
\mathtt{g}_{\mathsf{brtureF}} & \triangleq \lambda \mathbb{S}, \mathbb{S}'.\mathtt{NextS}_{(\mathsf{brture},\_)}\,\mathbb{S}\,\mathbb{S}' & (\text{where } \mathbb{S}.\mathbb{K} = w :: \mathbb{K}', w = \mathsf{False}) \\
\mathtt{g}_{\mathsf{c}} & \triangleq \lambda \mathbb{S}, \mathbb{S}'.\mathtt{NextS}_{(\mathsf{c},\,\_)}\,\mathbb{S}\,\mathbb{S}' & (\text{for all other } \mathsf{c})
\end{array}
$$

Figure 9. Local State and Program Point Transitions



```
//source code          ;bytecode for BVM
//function bar         -{(p_B, g_B)}  ;bar
void bar(){            0 pushc 1 ;push imm 1
  int a=1;             1 pop a    ;a = 1
}                      -{(p_C, g_C)}
//funcion foo          2 ret      ;bar return
void foo(){            -{(p, g)}      ;foo
  int b=1;             3 pushc 1 ;push imm 1
  bar();               4 pop b    ;b = 1
  b=1;                 -{(p_A, g_A)}
}                      5 cal 0    ;call bar
                       -{(p_D, g_D)}
                       6 pushc 2 ;push imm 2
                       7 pop b    ;b = 2
                       -{(p_E, g_E)}
                       8 ret      ;foo return
```

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

after $\mathtt{g}_\iota$. The composition of two subsequent transitions $\mathtt{g}$ and $\mathtt{g}'$ is represented as $\mathtt{g} \circ \mathtt{g}'$, and $\mathtt{p} \circ \mathtt{g}$ refines $\mathtt{g}$ with the extra knowledge that the initial state satisfies $\mathtt{p}$.

The predicate $\mathtt{p} \Rightarrow \mathtt{g}_\iota$ means that the state transition $\mathtt{g}_\iota$ would not get stuck as long as the starting state satisfies $\mathtt{p}$. The second premise in the SEQ rule means if the current state satisfies $\mathtt{p}$, after state transition $\mathtt{g}_\iota$, the new state satisfies $\mathtt{p}'$. The last premise in the SEQ rule requires the composition of $\mathtt{g}_\iota$ and $\mathtt{g}''$ fulfilling $\mathtt{g}$, knowing the current state satisfies $\mathtt{p}$.

To check the well-formedness of an instruction sequence beginning with ι, the programmer needs to find an intermediate specification $(\mathtt{p}'', \mathtt{g}'')$, which serves both as the postcondition for ι and as the precondition for the remaining instruction sequence. As shown in the SEQ rule, we check that:

- the remaining instruction sequence is well-formed with regard to the intermediate specification;
- the `NextS` relations for these instructions require the target stack pointer to be in the domain of stack, therefore the premise $\mathtt{p} \Rightarrow \mathtt{g}_\iota$ requires that $\mathtt{p}$ contains the target stack cell ownership;
- $\mathtt{p}''$ is satisfied by the resulting state of ι; and
- if the remaining instruction sequence satisfies its guarantee $\mathtt{g}''$, the original instruction sequence satisfies $\mathtt{g}$.

Function Call and Return. A precondition for an instruction sequence contains a predicate $\mathtt{p}$ specifying the current state, and a *guarantee* $\mathtt{g}$ describing the relation between the current state and the state at the return point of the current function (if the function ever returns). Figure 10(b) shows the meaning of the specification $(\mathtt{p}, \mathtt{g})$ for the function `foo` defined in Figure 10(a). Note that $\mathtt{g}$ may cover multiple instruction sequences. If a function has multiple return points, $\mathtt{g}$ governs all the traces from the current program point to any return point.

Figure 10(c) illustrates a function call to `bar` (point `B`) from `foo` at point `A` (label $\mathtt{pc} = 5$), with the return address $\mathtt{pc} + 1$ (point `D`). The specification of `bar` is $(\mathtt{p}_B, \mathtt{g}_B)$. Specifications at `A` and `D` are $(\mathtt{p}_A, \mathtt{g}_A)$ and $(\mathtt{p}_D, \mathtt{g}_D)$ respectively, where $\mathtt{g}_A$ governs the code segment `A-E` and $\mathtt{g}_D$ governs `D-E`.

To ensure that the program behaves correctly, we need to enforce the following conditions:

- the precondition of function `bar` should be satisfied, *i.e.*,
  $$\forall \mathbb{S}, \exists \mathbb{S}'.\mathtt{p}_A\,\mathbb{S} \wedge \mathtt{g}_{\mathsf{cal}}\,\mathbb{S}\,\mathbb{S}' \rightarrow \mathtt{p}_B\,\mathbb{S}';$$
- after `bar` returns, caller `foo` resumes its execution from `D`,
  $$\forall \mathbb{S}, \mathbb{S}'', \mathbb{S}', \mathbb{S}^*.\mathtt{p}_A\,\mathbb{S} \rightarrow \mathtt{g}_{\mathsf{cal}}\,\mathbb{S}\,\mathbb{S}' \rightarrow \mathtt{g}_B\,\mathbb{S}'\,\mathbb{S}^* \rightarrow \mathtt{g}_{\mathsf{ret}}\,\mathbb{S}^*\,\mathbb{S}'' \rightarrow \mathtt{p}_D\,\mathbb{S}'';$$
- if the function `bar` and the code segment `D-E` satisfy their specifications, the specification for `A-E` is satisfied, *i.e.*,
  $$\forall \mathbb{S}, \mathbb{S}'', \mathbb{S}''', \mathbb{S}', \mathbb{S}^*.\mathtt{p}_A\,\mathbb{S} \rightarrow \mathtt{g}_{\mathsf{cal}}\,\mathbb{S}\,\mathbb{S}' \rightarrow \mathtt{g}_B\,\mathbb{S}'\,\mathbb{S}^* \rightarrow \mathtt{g}_{\mathsf{ret}}\,\mathbb{S}^*\,\mathbb{S}'' \rightarrow \mathtt{g}_D\,\mathbb{S}''\,\mathbb{S}''' \rightarrow \mathtt{g}\,\mathbb{S}\,\mathbb{S}''';$$

From these premises, we can define a special guarantee $\mathtt{g}_{\mathsf{fun}}$for callee $\mathtt{g}_{\mathsf{fun}} \triangleq \lambda \mathbb{S}, \mathbb{S}''.\exists \mathbb{S}', \exists \mathbb{S}^*, \mathtt{g}_{\mathsf{cal}}\,\mathbb{S}\,\mathbb{S}' \wedge \mathtt{g}_B\,\mathbb{S}'\,\mathbb{S}^* \wedge \mathtt{g}_{\mathsf{ret}}\,\mathbb{S}^*\,\mathbb{S}''$. Thus, we can rewrite last two premises as:

$$\forall \mathbb{S}, \mathbb{S}''.\mathtt{p}_A\,\mathbb{S} \rightarrow \mathtt{g}_{\mathsf{fun}}\,\mathbb{S}\,\mathbb{S}'' \rightarrow \mathtt{p}_D\,\mathbb{S}'';$$

and

$$\forall \mathbb{S}, \mathbb{S}'', \mathbb{S}'''.\mathtt{p}_A\,\mathbb{S} \rightarrow \mathtt{g}_{\mathsf{fun}}\,\mathbb{S}\,\mathbb{S}'' \rightarrow \mathtt{g}_D\,\mathbb{S}''\,\mathbb{S}''' \rightarrow \mathtt{g}\,\mathbb{S}\,\mathbb{S}'''.$$

Above conditions are enforced by the CAL rule shown in Figure 7. It can be seen that $\mathtt{g}_{\mathsf{fun}}$ describes the state transition from function call point to the return point.

It also should be noticed that we do not require a particular return value but only require that stack contain a code pointer specified in local Ψ at the return state $\mathbb{S}''$, which is provable based on the knowledge of $\mathtt{p}$ and $\mathtt{g}_{\mathsf{fun}}$. This means that out logic can be used to certify any convention for multi-return function call.

The RET rule is straightforward. It simply requires that the function has finished its guaranteed transition at this point. So a state transition $\mathtt{g}_{\mathsf{ret}}$ should satisfy the remaining behavior of the callee function $\mathtt{g}$. In this rule, we do not need to know any information about return address. So it can be used to support modular certification of any callee function without knowing the caller.

Call Stack Invariant. Generalizing the safety requirement, we recursively define the "well-formed function call stack with depth *n*" as follows:

$$
\begin{array}{l}
\mathsf{WFST}(\mathtt{g}, \mathbb{K}_c, \mathbb{S}, \Psi) \triangleq \neg \exists \mathbb{S}'.\, \mathtt{g}\,\mathbb{S}\,\mathbb{S}', \quad \text{where } \mathbb{K}_c = \mathsf{nil}. \\
\mathsf{WFST}(\mathtt{g}, \mathbb{K}_c, \mathbb{S}, \Psi) \triangleq \forall \mathbb{S}'.\mathtt{g}\,\mathbb{S}\,\mathbb{S}' \rightarrow \mathtt{p}'\,\mathbb{S}' \wedge \mathsf{WFST}(\mathtt{g}', \mathbb{K}_c{}', \mathbb{S}', \Psi), \\
\text{where } \mathbb{K}_c = \mathtt{f} :: \mathbb{K}_c{}', (\mathtt{p}', \mathtt{g}') = \Psi(\mathtt{f}).
\end{array}
$$

When the function call stack is empty, we are in the outermost function which has no return code pointer. Thus, there exists no $\mathbb{S}'$ at which the function can return, *i.e.*, $\neg \exists \mathbb{S}'.\, \mathtt{g}\,\mathbb{S}\,\mathbb{S}'$.

Then the stack invariant we need to enforce is that, at each program point with specification $(\mathtt{p}, \mathtt{g})$, the program state $\mathbb{S}$ must satisfy $\mathtt{p}$ and there exists a well-formed control stack in $\mathbb{S}$. The invariant is formally defined as:

$$\{(\mathtt{p}, \mathtt{g})\}\,\Psi\,\mathbb{S} \triangleq \mathtt{p}\,\mathbb{S} \wedge \mathsf{WFST}(\mathtt{g}, \mathbb{K}_c, \mathbb{S}, \Psi).$$

The actual depth of the function call stack is not considered in this definition. We do not specify the other features of the stack in the invariant, which makes our logic very general and flexible to use.

We should prove that this invariant holds at every step of program execution. The stack invariant essentially explains why we can have such a simple RET rule, which "typechecks" the return instruction without requiring a valid code pointer.

Other Instructions. The execution of brture may either fall through or jump to the target code label, depending on whether the condition holds. So in the BRTURE rule, we use $\mathtt{g}_{\mathsf{brtureT}}$ and $\mathtt{g}_{\mathsf{brtureF}}$ to represent identity transitions with extra knowledge about stack $\mathbb{K}$. We also need to know that $\mathtt{p}$ contains the ownership of the

$$
\begin{array}{llll}
\mathrm{p}_0 & \triangleq (r \rightsquigarrow \_) * (\exists i >= 0,\ n \mapsto i), & \mathrm{g}_0 & \triangleq \mathbb{H}'(\mathsf{r}) = \mathbb{H}(\mathsf{n})! \\
\mathrm{p}_3 & \triangleq (\mathbb{H}(r) >= 1) \wedge (\mathbb{H}(n) >= 0), & \mathrm{g}_3 & \triangleq \mathbb{H}'(\mathsf{r}) = \mathbb{H}(\mathsf{r}) * \mathbb{H}(\mathsf{n})! \\
\mathrm{p}_{11} & \triangleq \mathrm{p}_3, & \mathrm{g}_{11} & \triangleq \mathrm{g}_3
\end{array}
$$

Figure 11. Specifications: While Loop Example

target stack cell. A direct jump is safe (rule GOTO) if the current assertion can imply the assertion of the target code label as specified in Ψ. It should be viewed as a specialization of BRTURE.

Soundness of CBP. The soundness of the program logic is carried out following the syntactic approach in Coq proof assistant. Based on the progress and preservation lemmas, the soundness of *CBP* guarantees that the complete system after linking never gets stuck as long as the initial state satisfies the program invariant defined by the WLD rule. Furthermore, the invariant will be always holding during execution, from which we can derive rich properties of programs. It also guarantees that the specifications in Ψ hold when the corresponding program points are reached by goto or call instructions and hold at the boundary of program modules. The soundness (theorem **??**) proof has been formally encoded in Coq.

Lemma 2.1 (CBP Progress) If $\Psi \vdash \{\mathrm{s}\}\mathbb{W}$, then there exists a program $\mathbb{W}'$, such that $\mathbb{W} \longmapsto \mathbb{W}'$.

Lemma 2.2 (CBP Preservation) If $\Psi \vdash \{\mathrm{s}\}\mathbb{W}$, and $\mathbb{W} \longmapsto \mathbb{W}'$, then there exists s', $\Psi \vdash \{\mathrm{s}'\}\mathbb{W}'$.

3. Example and Implementation

3.1 Modular Certification: While loop Factorial Function

A factorial function implemented with while loop and non-local variables is shown in this section to demonstrate particular features of our logic, and to show how to write specification and how to prove bytecode programs with *CBP*.

Get Instruction Sequences. Factorial function source code and the bytecode program with its specifications for *BVM* are shown in Figure 4 (Section 2). Finding the instruction sequence is the first step to certify a program. From the definition in Figure 1, we know that an instruction sequence is a set of instructions ending with unconditional jump jmp or function return ret.

Thus, it can be seen that there are three instruction sequences in while loop program. Labels 0∼2 form the first instruction block. And the second one is the instruction block from label 3 to 11. And labels 11∼15 is the block of remain instructions.

Write Specification for Instruction Sequences. Then the programmer needs to give code heap specification Ψ, which is a finite mapping from code labels f to code specifications s which is a pair (p,g). *CBP* specifications for code heap are embedded in the code, enclosed by -{} in shadow box. Specifications of this example are given in Figure 11. To simplify our presentation, we write the predicate p in the form of a proposition with free variables referring to components of the state $\mathbb{S}$.

Following the inference rules, the code specifications should be given for these points: the head of a instruction sequence, the target labels of function call instruction call and jump instructions (including goto and brture), and the function call return address which is just after call instruction call.

The specification of the this procedure is given as $(\mathrm{p}_0,\mathrm{g}_0)$. From p_0, we know that the values of variables r and n which stored in memory heap are inside the proper scope. The guarantee g_0 specifies the behavior of the function: the non-local variables r and n which are saved in memory fulfill $(\mathbb{H}'(\mathsf{r}) = \mathbb{H}(\mathsf{n})!)$.

$(\mathrm{p}_3,\mathrm{g}_3)$ is the assertion for while loop body. The precondition p_3 means that the values of variables r and n are still inside the proper scope. The guarantee g_3 says that the result which is stored in memory heap must fulfill the loop fixpoint. The specification $(\mathrm{p}_{11},\mathrm{g}_{11})$ at the begin point of this while loop is equal to $(\mathrm{p}_3,\mathrm{g}_3)$.

```
//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 point
                    | 19 ret          ;caller return
```

Figure 12. Caller of Factorial Function

$$
\begin{array}{llll}
\mathrm{p}_{16} & \triangleq (r \rightsquigarrow \_) * (n \rightsquigarrow \_), & \mathrm{g}_{16} & \triangleq \mathbb{H}'(\mathsf{r}) = 3! \\
\mathrm{p}_{19} & \triangleq (r \mapsto 3!) * n \mapsto 0), & \mathrm{g}_{19} & \triangleq \mathrm{g}_{16} \\
\mathrm{p}_0 & \triangleq ?, & \mathrm{g}_0 & \triangleq ?
\end{array}
$$

Figure 13. Specifications: Caller of the Recursive Factorial

Certify and Link Them Together. To check the well-formedness of an instruction sequence beginning with ι, a programmer should apply the appropriate inference rules and find intermediate assertions such as $(\mathrm{p}',\mathrm{g}')$, which serves both as the postcondition for ι and as the precondition for the remaining instruction sequence.

After that, a programmer is also required to establish the well-formedness of each individual module via the CDHP rule. Two non-intersecting well-formed code heaps can then be linked together via the LINK rule. The WLD rule requires that all code heaps be linked into one single well-formed global one.

Support Modular Certification. All the code specifications Ψ used in CBP rules are the *local* specifications for the current module. Thus, CBP supports modular reasoning about function call/return in the sense that caller and callee can be in different modules and be certified separately. When specifying the callee procedure, we do not need any knowledge about the return address in its precondition. The RET rule for the instruction "ret" does not have any constraints on the return address.

3.2 Modular Certification: Caller of Factorial Function

Source code and bytecode program with specification of the caller for the while loop factorial example are shown in Figure 12.

This function just initializes the variables n, and then calls function factor. The specification at the entry point is $(\mathrm{p}_{16},\mathrm{g}_{16})$. The precondition p_{16} simply says that the memory cells for variables n and r are there for this function to run. The guarantee g_{16} specifies the behavior of the caller procedure: the result r in memory heap is the factorial of 3. The specification of returns point is $(\mathrm{p}_{19},\mathrm{g}_{19})$. p_{19} means that the memory cells for variables n and r are still there. The guarantee g_{19} is just the same as g_{16}.

From CAL inference rule, we know that the specification of the callee's entry point should be added. The specification $(\mathrm{p}_0,\mathrm{g}_0)$ in Figure 11 can be used. Furthermore, the specification of function entry point defines its interface. Caller can use any callees which share the same interface.

3.3 Implementation with Coq

Our logic framework presented in this paper has been applied to bytecode programs for our verified stack-based virtual machine. We have formalized *BVM*, its operational semantics, and the program logic *CBP*.

The syntax of our machine is encoded in Coq using inductive definitions. Operational semantics of the machine and all the inference rules of program logic are defined as inductive relations. The soundness of the framework itself is formalized and proved in Coq following the syntactic approach.

CiC, the underlying higher-order logic in Coq, rather than a new assertion language known as the deep embedding approach is adapted as our assertion language. This shallow embedding approach greatly reduces the workload of formulating our logic sys-

Component Name	Number of lines
Basic Utility Definitions & Lemmas	2,354
Machine & Operational Semantics	3,285
CBP Rules & Soundness	1,166
IR Examples Source Code and Spec.	168*
Caller Main Spec. & Proof	1,304
Total	8277

* They are the Coq source files containing the encoding of the 19 lines real bytecode code, including program specification and factorial related lemmas .

Figure 14. The Verified Package in Coq

tems. The proof is also formalized and implemented in Coq and is machine-checkable.

These examples are usually implemented directly in bytecode and are hard to certify using the existing approaches. Manually optimized bytecode or code generated by optimizing compilers can also be certified using our systems.

The implementation of *CBP* logic includes around 3300 lines of Coq encoding of *BVM* and its operational semantics, 1200 lines encoding of *CBP* rules and the soundness proof. We have written several hundred lines of Coq tactics to certify practical examples, including the while-loop and function call/return.

The Coq implementation has taken several man-months, out of which a significant amount of efforts have been put on the implementation of basic facilities, including lemmas and tactics for partial mappings and Separation Logic assertions. These common facilities are independent of the task of certifying examples and can be reused in future projects.

It is found in our experience that human smartness still plays an important role to come up with proper program specifications, and the difficulty depends on the property one is interested in and the subtlety of the algorithms itself. Given proper specifications, proof construction of bytecode is mostly routine work. Some premises of inference rules can be automatically derived after defining lemmas for common instructions.

Compared the experiences in *CBP* with that in SCAP, we found that the code size ratios of bytecode programs to their proofs and assembly codes to their proofs looks almost the same. While bytecode is a fairly compact format compared to native code. Most JVM instructions use only 1 or 2 bytes. Moreover, they are sophisticated instructions that cannot be translated into a single native processor instruction as a rule. In fact, our CertVM expand code size by the factor of 15, while most Java compilers expand code size by a factor of 5 to 10 [21]. With out logic, we only write proof for bytecode programs rather than write proof for the corresponding assembly codes directly. So the workload will be greatly reduced by a factor of 5 to 10. That will be a significant improvement for fully certified subroutines with machine checkable proofs.

Extensions and Future Work. An important and useful extension is object-oriented features such as objects, references, methods, and inheritance. Extension of the program logic to support exception handling is straightforward but interesting, following the similar idea of function call/return. Reasoning about exceptions is not much different from reasoning about functions. First, there is an action of setting an exception handler. It is similar to function call, as the code must save all the information necessary to resume execution from that point. Raising an exception is similar to a return, except that this return does not just go to the previous function, but rather to the closest exception handler.

In *CBP* logic system, we support most of the complex stack-based control abstractions and unstructured control flow. But, we do not support concurrency yet. Concurrency is one of the hot topics, still far away from satisfaction. There are a number of subtle problems even in the well-used bytecode programs such as JDK synchronized classes [18]. It is actually an easy task to extend the machine to support concurrency. But it is not so easy to define a simple logic system to certify concurrent bytecode programs. We will try it in the near future.

4. Related Work and Conclusion

Logic for Bytecode. Although the interest in specification and certification of bytecode applications is relatively recent concerns, much works have been done in this field. Bytecode Modeling Language (BML) [5] focuses on writing understandable specifications for bytecode. It allows the application developer to specify the behaviour of an application in the form of annotations at the level of the bytecode. In particular, JAVA source code specifications can be compiled into BML specifications.

Some other efforts focus on the development of a sound proof system. Developed a simple Hoare-like logic for bytecode programs within Isabelle, Quigley [15] has demonstrated that it is possible to define a programming logic for bytecode programs that allows the proof of bytecode programs containing loops. Focusing on the program logic for reasoning about program resource consumption, MRG project [2] presents the resource-aware operational semantics of an abstract fragment of JVM Language (named Grail), the program logic, and the proof. A program logic [3] which combines Hoare triples for methods with instruction specifications is presented for a JAVA-like bytecode language by Bannwart and Müller. Their logic supports lots of object-oriented features such as objects, references, methods, and inheritance. Benton [4] proposed a typed, compositional logic for a stack-based abstract machine to verify bytecode programs which are written in an imperative subset of .NET CIL. He uses a higher-level abstract machine with separate data stack and control stack; the latter cannot be touched by regular instructions except call and ret.

Reasoning about Control Stacks. Reasoning about control stacks is extremely difficult for mid-level and low-level code programs due to the flat nature.

STAL [13] and its variations [20] can also type-check function call/return and stack unwinding, but they all treat return code pointers as first-class code pointers and stacks as “closures”. Using compound stacks, STAL can type-check exceptions and weak-continuations, but this approach is rather limited. If multiple exception handlers defined at different depths of the stack are passed to the callee, the callee has to specify their order on the stack, which breaks modularity.

Tan and Appel [19] use the implicit finite unions structure to study the low-level language. As a result, they arrived at continuation-style Hoare logic explainable by indexed model, with a rather convoluted interpretation of Hoare triples involving explicit fixpoint approximations. Saabas and Uustalu [17] introduced a compositional natural semantics and Hoare logic based on the implicit finite unions structure for a simple low-level language with expressions. They applied their logic to a stack-based language with basic stack operation like “push”. Ni and Shao’s work [14] combines the syntactic approach used in type systems with logic systems to support code pointer specification. They introduce a syntactic construct cptr to describe the embedded code pointer, which is interpreted in the meta-logic at a lower level to avoid circularity.

Feng *etc.* [10] proposed SCAP to modularly certify assembly code with stack-based control abstractions. Instead of treating the return code pointers as first class code pointers, SCAP follows the producer/consumer model to reason about stack-based control abstractions, *e.g.*, function call/return, exceptions and thread context switch routines. In addition, SCAP does not enforce any specific stack layout, therefore it can be used to support sequential stacks, linked stacks, and heap-allocated activation records.

Sharing the same producer/consumer model, this paper applies the FPCC concept bytecode programs. We build a Hoare-like logic system to certify bytecode programs which run on verified virtual machine. As the examples shown, program with complex control stack operations be certified within our logic.

Conclusion. This paper presents a logic framework to verify bytecode programs. This paper defines a Hoare-style logic for modularly specifying and certifying bytecode programs with complex stack-based control abstractions and unstructured control flow. Our bytecode language is similar to JAVA bytecode and CIL.

To certify a bytecode program, a programmer's task is only required to find the specification and establish the well-formedness of individual module. This logic system is fully mechanized: the complete soundness proof and a full verification of several examples are carried out in the Coq proof assistant [7].

Our work provides a foundation for reasoning about bytecode programs for stack-based virtual machine and makes a solid advance toward building a proof-transforming compilation environment. We believe this work may serve as a solid theoretical foundation to understand and reason about the popular and complex web applications which runs on stack-based virtual machine.

Acknowledgments

Acknowledgments

The authors thank Prof. Zhong Shao of Yale, Dr. Xinyu Feng of TTI-Chicago, Dr. Gang Tan of Lehigh University, and Dr. Juan Chen of Microsoft for useful discussions about FPCC. Yingying Sun helped prove a early version of logic framework. Yuan Dong, Shengyuan Wang, Liwei Zhang and Yunmin Zhu are supported in part by National Natural Science Foundation of China (under grant No. 60573017 and No. 90816006), Hi-Tech Research And Development Program of China (under grant No. 2008AA01Z102), and Basic Research Foundation of Tsinghua National Laboratory for Information Science and Technology (TNList). Any opinions, findings, and contributions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] 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.

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

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

[4] 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.

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

[6] C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proc. 2000 ACM Conf. on Prog. Lang. Design and Impl.*, pages 95–107, New York, 2000. ACM Press.

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

[8] ECMA. *Standard ECMA-335 Common Language Infrastructure*. 2006.

[9] 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.

[10] 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.

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

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

[13] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In *Proc. 1998 Int'l Workshop on Types in Compilation: LNCS Vol 1473*, pages 28–52. Springer-Verlag, 1998.

[14] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *POPL'06*, pages 320–333, 2006.

[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] 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.

[17] A. Saabas and T. Uustalu. Compositional type systems for stack-based low-level languages. In *Proc. of 12th Computing, Australasian Theory Symp., (CATS 2006)*, pages 27–39. Australian, 2006.

[18] 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.

[19] G. Tan and A. W. Appel. A compositional logic for control flow. In *VMCAI'06*, volume 3855 of *LNCS*, pages 80–94. Springer, 2006.

[20] J. C. Vanderwaart and K. Crary. A typed interface for garbage collection. In *Types in Lang. Design and Impl. (TLDI'03)*, pages 109–122, 2003.

[21] 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.