Modular Certification of Low-level Intermediate Representation Programs

Yuan Dong*, Shengyuan Wang*, Liwei Zhang† and Ping Yang‡
*Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
†School of Software, Tsinghua University, Beijing 100084, China
‡College of Information Science Beijing Language and Culture University, Beijing 100083, China
Email: {dongyuan,wwssyy}@tsinghua.edu.cn, zhanglw06@mails.tsinghua.edu.cn, yangp@blcu.edu.cn

***Abstract*—Modular certification of low-level intermediate representation (IR) programs is one of the key steps of proof-transforming compilation. The major challenges are the complexity of abstract control stacks and the lack of control flow information due to their flat nature. To tackle these challenges, we present in this paper a novel Hoare-style logic framework for modular certification of p-machine style bytecode as IR programs. This logic can fully support abstract control stacks and unstructured control flows for modular certification of IR programs involving while loops, procedure call/return, recursive procedures, and even nested procedures. It applies Foundational Proof-Carrying Code (FPCC) concepts to IR level. This system is expressive and fully mechanized. We prove its soundness and demonstrate its power by certifying the implementation of some IR programs in the Coq proof assistant. This work not only provides a solid theoretical foundation for reasoning about IR programs, but also makes an important advance toward building proof-transforming compilation environment in which certified IR code with proofs can be compiled to machine checkable proof-carrying low-level assembly code.**

***Keywords*-proof-carrying code; modular certification; nested procedure; intermediate representation program;**

I. Introduction

Proof-transforming compilation transforms a proof of a high-level program into a proof of its low-level compiled form as the program is being compiled it is a feasible approach to build large scale trustworthy program. Although lots of program logic for high-level language have been proposed in last forty years [1], [2], the proved properties cannot be preserved during traditional compilation. Machine level programs can be certified with proper logic systems [3], [4], [5]. However, the code of any non-trivial programs of these logic are really tedious to carry out by hand.

Modular certification of intermediate representation (IR) programs is the key of proof-transforming compilation. As a well defined IR, bytecode for stack-based machine is suitable to serve as a mid-level language for a proof-transforming compiler. A subset of Niklaus Wirth's p-machine style bytecode (P-Code) [6] is used in this paper.

To handle modern language with nested procedure like Pascal, GNU C extension and FORTRAN-90, our IR supports nested procedure. This interesting feature makes our IR language elegant and expressive. But the stack access becomes far more complicated. Our P-code machine has a single stack mixing both data and return addresses. So data store instructions can overwrite return addresses and thus mess with control-stack invariants. Almost all the instructions can access the stack, while p-code machine has no stack protection mechanics. Correct implementation of these constructs is of utmost importance to the safety and reliability of any IR programs.

```
-{(p_2, g_2)}  ;callee, procedure fact
2  int   4 ;stack frame 15 cal 1 2 ;call fact
3  lod 1 5 ;load n      -{(p_16, g_16)}
4  lit   0 ;load imm 0  16 ret     ;fact return
5  opr   9 ;n#0?        ;main caller
6  jpc  16 ;loop judge  -{(p_17, g_17)}
7  lod 1 4 ;load r      17 int   6 ;stack frame
8  lod 1 5 ;load n      18 lit   1 ;load imm 1
9  opr   4 ;r*n         19 sto 0 4 ;r=1
10 sto 1 4 ;save r      20 lit   5 ;load imm 5
11 lod 1 5 ;load n      21 sto 0 5 ;n=5
12 lit   1 ;load imm 1  22 cal 0 2 ;call fact
13 opr   3 ;n-1          -{(p_23, g_23)}
14 sto 1 5 ;save n      23 ret     ;main return
```

Figure 1. Nested Procedure Example

A complex IR program which involves recursive nested procedure is shown in Figure 1. Let's omit the contents in the shadow box. It is compiled from the Pascal like code in Figure 2. During calling function `fact`, instruction `cal 0 2` on label 22 saves its return pointer on stack, then updates stack frame and stack pointer, and jumps to function `fact` on label 2. When `fact` returns, the control jumps back to label 23, where `fact` restores return pointer, stack frame and stack pointers, then jumps back to its caller's code.

To certify a program like this, the major challenge is to formalize and capture the invariant of the nested procedure stack access of callee. Although there have been some efforts [7], [8], [9] on building logic systems for bytecode programs, none of them can verify this program because of the lack of nested procedure control stack support.

Our Contributions. This paper presents a Hoare-style logic framework $NCBP$ (Certifying Bytecode Programs with Nested Procedures) for modular certification of IR

```
var r, n;
procedure fact;
  begin
    if n#0 then
    begin
     r := r*n;
     n := n-1;
     call fact;
    end;
  end;
begin
  r:=1;
  n:=5;
  call fact;
end.
```

Figure 2. Source Code of Nested Procedure

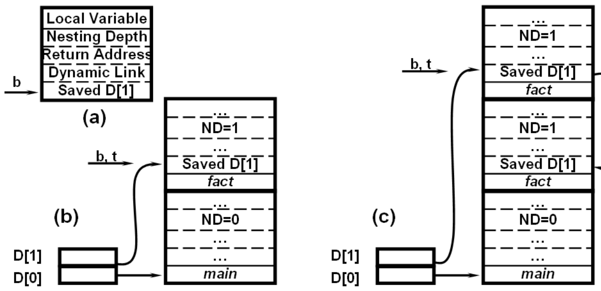


Figure 3. Stack and Display of PCM

programs with all kinds of stack-based control abstractions including while loops, procedure call/return, recursive procedures, and even nested procedures. Our logic support modular certification so the callee can be certified without knowing where it will return. This logic applies Foundational Proof-Carrying Code (FPCC) concepts to IR programs. We give the complete soundness proof and a full certification of several examples in the Coq proof assistant [10].

This work not only provides a foundation for reasoning about IR programs for stack-based machine, but also makes an important step toward proof-transforming compilation. This paper makes the following contributions:

- As far as we know, our work presents the first program logic facility for certifying the partial correctness of IR programs involving complex stack operations such as non-local access of recursive nested procedure.
- Using the "producer/consumer" model, we formalize the invariant of the stack-based procedure call/return control abstraction in a general way to describe caller's state transition from function call point to the return point. As an important advantage, it can be used as a general framework for other kinds of function call/return with proper instances of calling convention.
- To our best knowledge, our logic framework is also the first to apply FPCC concepts which is powerful for machine code certification to a common IR language. Our experience demonstrates that it should be feasible to build a pervasive Hoare-like logic framework for proof-transforming compilation from IR to machine code.

The rest of this paper is organized as follows: we first formalize a stack-based virtual machine and give its operational semantics (Sec II). Then we present a generic Hoare-style logic framework $NCBP$ system for certifying recursive nested procedure and show how to certify IR programs in the framework (Sec III and IV). Finally we discuss the implementation with Coq proof assistant tools and related research work, and then draw a conclusion.

II. Formal Definition of The Abstract Machine

We use P-machine style bytecode intermediate representation to show the capability of our method. P-code is a popular facility for language implementation and compiler construction. It runs on a stack-based abstract machine, PCM (P-Code Machine).

Our IR supports nested procedures, so the stack access becomes far more complicated. An efficient method using an auxiliary display array is adapted in our machine to handle non-local and cross-procedure access.

A. Stack and Display for Nested Procedure

In Figure 2, the only function at nesting depth 0 is the outermost procedure, $main$. Within $main$, we declare procedures $fact$ at nesting depth 1 which does factorial computation. The code suggests that the outer procedure calls $fact$ and $fact$ accesses itself recursively.

PCM has 3 registers — a program counter pc, which points at the current instruction in the code heap, a base register r_b, which marks the beginning of the active stack frame, and a top-of-stack register (or stack pointer) r_t which points to the top of the stack. To simplify the presentation, we use b and t to represent values contained in the register r_b and r_t.

Stack Frame. PCM has only one shared stack which provides access links and return address, local variables, and the arguments to local instructions. Stack frames is shown in Figure 3(a). There are four elements in a stack frame: a saved display for current nesting depth $\mathsf{sd}[\mathsf{nd}]$, a dynamic link dl which points to the frame base of caller procedure, a return address ra, and current nesting depth nd.

The procedure call is issued with instruction $\mathtt{cal}\ \mathsf{l}\ \mathsf{f}$ where l specifies the difference in nesting depths and f points to the target address. It works as follows: First, this instruction reserves the first four cells of the above stack frame, and sets current nesting depth nd, saved display sd, dynamic link dl, then saves the return address as $\mathsf{pc}+1$. Then, it sets register r_b to current stack frame base. After that, it updates current display $\mathbb{D}[\mathsf{nd}]$ to current stack frame base b. Finally it jumps to the callee procedure's address f.

A procedure always begins with the instruction $\mathtt{lit}\ \mathsf{w}$ to set register r_t to $\mathsf{t}+\mathsf{w}$. So w essentially specifies the space reserved for locals (the number of parameters plus 4). When procedure returns, the instruction $\mathtt{ret}$ restores current saved display $\mathbb{D}[\mathsf{nd}]$ from stack frame to $\mathbb{D}$, recovers its return address and stack pointer, then jumps back to the caller's code.

Display. Display consists of one pointer for each nesting depth. At all times, $\mathbb{D}[\mathtt{i}]$ points to the highest activation record on the stack for any procedure at nesting depth $\mathtt{i}$.

$$\begin{array}{rlcl}
(\textit{World}) & \mathbb{W} & ::= & (\mathbb{C}, \mathbb{S}, \mathsf{pc}) \\
(\textit{Code}) & \mathbb{C} & ::= & \{\mathsf{f} \rightsquigarrow \mathbb{I}\}^* \\
(\textit{State}) & \mathbb{S} & ::= & (\mathbb{K}, \mathbb{D}, \mathbb{R}) \\
(\textit{Stack}) & \mathbb{K} & ::= & \{\mathsf{k} \rightsquigarrow \mathsf{w}\}^* \\
(\textit{Display}) & \mathbb{D} & ::= & \{\mathsf{k} \rightsquigarrow \mathsf{w}\}^*
\end{array}
\qquad
\begin{array}{rlcl}
(\textit{RegFile}) & \mathbb{R} & ::= & \{\mathsf{r} \rightsquigarrow \mathsf{w}\}^* \\
(\textit{Labels}) & \mathsf{f}, \mathsf{k} & ::= & n(nat) \\
(\textit{Word}) & \mathsf{w} & ::= & i(int) \\
(\textit{Register}) & \mathsf{r} & ::= & \mathsf{r}_b \mid \mathsf{r}_t \\
(\textit{OprNum}) & \mathsf{m} & ::= & \{1 \ldots 13\}
\end{array}$$

$$\begin{array}{rl}
(\textit{Instr}) & \iota ::= \texttt{int w} \mid \texttt{lit w} \mid \texttt{lod l k} \mid \texttt{sto l k} \mid \\
& \qquad \texttt{opr m} \mid \texttt{jpc f} \mid \texttt{cal l f} \\
(\textit{Commd}) & \mathsf{c} ::= \iota \mid \texttt{ret} \mid \texttt{jmp f} \\
(\textit{InstrSeqs}) & \mathbb{I} ::= \iota;\mathbb{I} \mid \texttt{ret} \mid \texttt{jmp f}
\end{array}$$

Figure 4. Definition of An IR Machine

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

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

Figure 5. Definition of Representations

In Figure 3(c), we see the display $\mathbb{D}$, with $\mathbb{D}[0]$ holding a pointer to the activation record for $main$, the highest (and only) activation record for a function at nesting depth 0. Also, $\mathbb{D}[1]$ holds a pointer to the activation record for $fact$, the highest record at depth 1. From the saved display value in the stack frame of procedure $fact$, we also can keep track of all the former highest display records at nesting depth 1.

B. *Machine Specialization*

In Figure 4, we show our machine in a formal way. The whole machine configuration is called a "World" ($\mathbb{W}$), which consists of a read-only code heap ($\mathbb{C}$), an updatable state ($\mathbb{S}$), and an program counter (pc). The code heap is a finite partial mapping from code labels (f) to instruction sequences ($\mathbb{I}$). The state $\mathbb{S}$ contains a stack ($\mathbb{K}$), a stack frame display ($\mathbb{D}$) and a register file ($\mathbb{R}$). The program counter pc points to the current command in $\mathbb{C}$. We also define the instruction sequence $\mathbb{I}$ as a basic code block, *i.e.*, a list of sequential instructions ending with jump or return commands.

We use the dot notation to represent a component in a tuple, *e.g.*, $\mathbb{S}.\mathbb{K}$ means the stack in state $\mathbb{S}$. More representations are defined in Figure 5. $\mathbb{C}[\mathsf{f}]$ extracts an instruction sequence starting from f in $\mathbb{C}$. Representation $F\{a \rightsquigarrow b\}$ is function updating, and that F is any function.

It should be noticed that, in addition to normal push and pop operations, the instructions such as "lod" and "sto" could do random access to the stack, So the stack $\mathbb{K}$ is modeled as a partial mapping instead of a list. Some register and stack related notations are defined in Figure 6.

C. *Operational Semantics*

In Figure 7, we also define the operational semantics of each instruction in terms of the machine configuration transitions. The relation $\mathsf{NextS}_{(\mathsf{c},\mathsf{pc})}$ shows the transition

$$\begin{array}{lll}
\mathsf{t} \triangleq \mathbb{R}(\mathsf{r}_t) & \mathsf{b} \triangleq \mathbb{R}(\mathsf{r}_b) & \mathsf{sd} \triangleq \mathbb{K}(\mathsf{b}) \\
\mathsf{dl} \triangleq \mathbb{K}(\mathsf{b}+1) & \mathsf{ra} \triangleq \mathbb{K}(\mathsf{b}+2) & \mathsf{nd} \triangleq \mathbb{K}(\mathsf{b}+3)
\end{array}$$

Figure 6. Representations for Register/Stack

$\mathsf{NextS}_{\mathsf{c},\mathsf{pc}}\ \mathbb{S}\ \mathbb{S}'$ where $\mathbb{S} = (\mathbb{K}, \mathbb{D}, \mathbb{R})$

if c =	then $\mathbb{S}'$ =
`int w`	$(\mathbb{K}, \mathbb{D}, \mathbb{R}\{\mathsf{r}_t \rightsquigarrow \mathsf{t} + \mathsf{w}\}$
`lit w`	$(\mathbb{K}\{\mathsf{t} \rightsquigarrow \mathsf{w}\}, \mathbb{D}, \mathbb{R}\{\mathsf{r}_t \rightsquigarrow \mathsf{t} + 1\}$
`lod l w`	$(\mathbb{K}\{\mathsf{t} \rightsquigarrow \mathbb{K}(\mathbb{D}(\mathsf{nd} - \mathsf{l}) + \mathsf{w})\}, \mathbb{D}, \mathbb{R}\{\mathsf{r}_t \rightsquigarrow \mathsf{t} + 1\}$
`sto l w`	$(\mathbb{K}\{(\mathbb{D}(\mathsf{nd} - \mathsf{l}) + \mathsf{w}) \rightsquigarrow \mathbb{K}(\mathsf{t} - 1)\}, \mathbb{D}, \mathbb{R}\{\mathsf{r}_t \rightsquigarrow \mathsf{t} - 1\}$
`opr` 3	$(\mathbb{K}\{(\mathsf{t} - 2) \rightsquigarrow \mathbb{K}(\mathsf{t} - 2)) - \mathbb{K}(\mathsf{t} - 1)\}, \mathbb{D}, \mathbb{R}\{\mathsf{r}_t \rightsquigarrow \mathsf{t} - 1\})$ (-)
`opr` 4	$(\mathbb{K}\{(\mathsf{t} - 2) \rightsquigarrow \mathbb{K}(\mathsf{t} - 2) * \mathbb{K}(\mathsf{t} - 1)\}, \mathbb{D}, \mathbb{R}\{\mathsf{r}_t \rightsquigarrow \mathsf{t} - 1\})$ (*)
`opr` 9	$(\mathbb{K}\{(\mathsf{t} - 2) \rightsquigarrow 0\}, \mathbb{D}, \mathbb{R}\{\mathsf{r}_t \rightsquigarrow \mathsf{t} - 1\})$, if $(\mathbb{K}(\mathsf{t} - 2) \neq \mathbb{K}(\mathsf{t} - 1))$ $(\mathbb{K}\{(\mathsf{t} - 2) \rightsquigarrow 1\}, \mathbb{D}, \mathbb{R}\{\mathsf{r}_t \rightsquigarrow \mathsf{t} - 1\})$, else. (neq)
`jpc f`	$(\mathbb{K}, \mathbb{D}, \mathbb{R}\{\mathsf{r}_t \rightsquigarrow (\mathsf{t} - 1)\})$
`cal l f`	$(\mathbb{K}\{\mathsf{t} \rightsquigarrow \mathbb{D}(\mathsf{nd} - \mathsf{l} + 1), (\mathsf{t} + 1) \rightsquigarrow \mathsf{b},$ $(\mathsf{t} + 2) \rightsquigarrow (\mathsf{pc} + 1), (\mathsf{t} + 3) \rightsquigarrow (\mathsf{nd} - \mathsf{l} + 1)\},$ $\mathbb{D}\{(\mathsf{nd} - \mathsf{l} + 1) \rightsquigarrow \mathsf{t}\}, \mathbb{R}\{\mathsf{r}_b \rightsquigarrow \mathsf{t}\})$
`ret`	$(\mathbb{K}, \mathbb{D}\{\mathsf{nd} \rightsquigarrow \mathbb{K}(\mathsf{b}))\}, \mathbb{R}\{\mathsf{r}_t \rightsquigarrow \mathsf{b}, \mathsf{r}_b \rightsquigarrow \mathsf{dl}\})$
`jmp f`	$(\mathbb{K}, \mathbb{D}, \mathbb{R})$

$\mathsf{NextPC}_{(\mathsf{c},\mathbb{S})}\ \mathsf{pc}\ \mathsf{pc}'$ where $\mathbb{S} = (\mathbb{K}, \mathbb{D}, \mathbb{R})$

if c =	then pc′ =
`jpc f`	f if $\mathbb{K}(\mathsf{t}) = 0$; pc + 1 others
`cal l f`	f
`ret`	$\mathbb{K}(\mathsf{b}+2)$
`jmp f`	f
...	pc+1

$$\frac{\mathsf{c} = \mathbb{C}(\mathsf{pc}) \quad \mathsf{NextS}_{(\mathsf{c},\mathsf{pc})}\ \mathbb{S}\ \mathbb{S}' \quad \mathsf{NextPC}_{(\mathsf{c},\mathbb{S})}\ \mathsf{pc}\ \mathsf{pc}'}{(\mathbb{C}, \mathbb{S}, \mathsf{pc}) \longmapsto (\mathbb{C}, \mathbb{S}', \mathsf{pc}')} \ (\text{PC})$$

Figure 7. operational semantics of PCM

of states by executing c with program counter pc; while $\mathsf{NextPC}_{(\mathsf{c},\mathbb{S})}$ shows how pc changes after c is executed with $\mathbb{K}$, $\mathbb{D}$ and $\mathbb{R}$.

The instruction set captures the most basic and common PCM, 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. $\mathbb{W} \longmapsto \mathbb{W}'$ is made by executing the instruction pointed to by $\mathbb{W}.\mathsf{pc}$.

III. The Program Logic for PCM

$NCBP$, our new program logic for certifying IR programs, follows the invariant-based proof technique: we define a program invariant stronger than the safety property we are interested in. The program invariant guarantees that the program can execute one step, while the instruction rules guarantee that the invariant still holds in the new program state. In this way, we know the program will never get stuck unless hardware problem.

A. *Specification Language*

To specify a program with a code heap $\mathbb{C}$, the programmer should insert specifications at the start points of instruction sequences. We use the mechanized *meta-logic* which is implemented in the Coq proof assistant as our specification language. This logic corresponds to a higher-order predicate logic with inductive definitions. As shown in Figure 8, the specification s is a pair (p, g). The assertion p is a predicate over program state $\mathbb{S}$, while guarantee g is a predicate over two program states. As we can see, the $\mathsf{NextS}_{(\mathsf{c},\mathsf{pc})}$ relation defined in Figure 7 is a special form of g which is over the

$$\begin{array}{rlll}
(\textit{Pred}) & \mathsf{p} & \in & \textit{State} \rightarrow \textit{Prop} \\
(\textit{Guarantee}) & \mathsf{g} & \in & \textit{State} \rightarrow \textit{State} \rightarrow \textit{Prop} \\
(\textit{Spec}) & \mathsf{s} & ::= & (\mathsf{p}, \mathsf{g}) \\
(\textit{CdHpSpec}) & \Psi & ::= & \{(\mathsf{f}_1, \mathsf{s}_1), \ldots, (\mathsf{f}_n, \mathsf{s}_n)\} \\
(\textit{KPred}) & \mathsf{k} & \in & \textit{Stack} \rightarrow \textit{Prop}
\end{array}$$

Figure 8. Specification Constructs for $NCBP$

two adjacent states. We use p to specify the pre-condition over state, and use g to specify the behavior from the specified program point to *current* function return point.

Specification Ψ for code heap $\mathbb{C}$ associates code labels f with corresponding s. Multiple s may be associated with the same f, just as a function may have multiple specified interfaces. We use the predicate k to specify the stack. To enforce the stack partition between different functions, we encode Separation Logic connectors in our specification language (which is also our meta-logic). We also use standard separation logic primitives [11] as assertion operators.

B. Inference Rules

The inference rules are defined as following judgments:

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

The program logic inference rules are shown in Figure 9.

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

- The code heap $\mathbb{C}$ is well-formed following the CDHP rule.
- 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 pc has a specification s in Ψ, thus the current instruction sequence $\mathbb{C}[\mathsf{pc}]$ is well-formed with s.
- Given the exported Ψ', current state $\mathbb{S}$ satisfies the s.

Code Heap Modules. Our logic supports *separate certification* of the program modules. Modules are small code heaps which contain at least one code block. The specification of a module contains specifications of both the code blocks in the current module and the external code blocks in the callee's module. In the CDHP rule, Ψ contains specifications for external code (called by local module $\mathbb{C}$), while Ψ' contains specifications for code blocks in the module $\mathbb{C}$ for other modules. The well-formedness of each individual module is given via the CDHP rule. All non-intersecting well-formed modules can be linked via the LINK rule into a well-formed global code heap.

Sequential Instructions. Like traditional Hoare-logic [12], our logic also uses the pre- and post-condition 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 with ι at the code label pc, given the imported interface in Ψ and a pre-condition (p, g). An intermediate specification $(\mathsf{p}'', \mathsf{g}'')$ with respect to which serves as the post-condition

$$\boxed{\Psi \vdash \{\mathsf{s}\}\, \mathbb{W}} \quad \textbf{(\textit{Well-formed World})}$$

$$\frac{\Psi \vdash \mathbb{C} : \Psi' \qquad \Psi \subseteq \Psi' \qquad \Psi \vdash \{\mathsf{s}\}\, \mathsf{pc} :\ \mathbb{C}[\mathsf{pc}] \qquad \{\mathsf{s}\}\ \Psi'\ \mathbb{S}}{\Psi \vdash \{\mathsf{s}\}\, (\mathbb{C}, \mathbb{S}, \mathsf{pc})} \ (\text{WLD})$$

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

$$\frac{\text{for all } (\mathsf{f}, \mathsf{s}) \in \Psi' : \qquad \Psi \vdash \{\mathsf{s}\}\, \mathsf{f} :\ \mathbb{C}[\mathsf{f}]}{\Psi \vdash \mathbb{C} : \Psi'} \ (\text{CDHP})$$

$$\frac{\Psi_1 \vdash \mathbb{C}_1 : \Psi_1' \qquad \Psi_2 \vdash \mathbb{C}_2 : \Psi_2' \qquad \mathbb{C}_1 \# \mathbb{C}_2}{\Psi_1 \cup \Psi_2 \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1' \cup \Psi_2'} \ (\text{LINK})$$

$$\boxed{\Psi \vdash \{\mathsf{s}\}\, \mathbb{I}} \quad \textbf{(\textit{Well-formed Instr. Sequence})}$$

$$\frac{\begin{array}{lll} \iota \notin \{\mathtt{jpc}, \mathtt{cal}\} & \Psi \vdash \{(\mathsf{p}'', \mathsf{g}'')\}\, \mathsf{pc}{+}1 :\ \mathbb{I} & \\ \mathsf{p} \Rightarrow \mathsf{g}_\iota & (\mathsf{p} \rhd \mathsf{g}_\iota) \Rightarrow \mathsf{p}'' & (\mathsf{p} \circ (\mathsf{g}_\iota \circ \mathsf{g}'')) \Rightarrow \mathsf{g} \end{array}}{\Psi \vdash \{(\mathsf{p}, \mathsf{g})\}\, \mathsf{pc} :\ \iota; \mathbb{I}} \ (\text{SEQ})$$

$$\frac{\begin{array}{ll} (\mathsf{f}', (\mathsf{p}', \mathsf{g}')) \in \Psi & \Psi \vdash \{(\mathsf{p}'', \mathsf{g}'')\}\, \mathsf{pc}{+}1 :\ \mathbb{I} \\ (\mathsf{p} \rhd \mathsf{g}_{\mathtt{jpcT}}) \Rightarrow \mathsf{p}' & (\mathsf{p} \circ (\mathsf{g}_{\mathtt{jpcT}} \circ \mathsf{g}')) \Rightarrow \mathsf{g} \\ (\mathsf{p} \rhd \mathsf{g}_{\mathtt{jpcF}}) \Rightarrow \mathsf{p}'' & (\mathsf{p} \circ (\mathsf{g}_{\mathtt{jpcF}} \circ \mathsf{g}'')) \Rightarrow \mathsf{g} \end{array}}{\Psi \vdash \{(\mathsf{p}, \mathsf{g})\}\, \mathsf{pc} :\ \mathtt{jpc}\ \mathsf{f}'; \mathbb{I}} \ (\text{JPC})$$

$$\frac{\begin{array}{lll} (\mathsf{pc}{+}1, (\mathsf{p}'', \mathsf{g}'')) \in \Psi & \Psi \vdash \{(\mathsf{p}'', \mathsf{g}'')\}\, \mathsf{pc}{+}1 :\ \mathbb{I} & \\ (\mathsf{p} \rhd \mathsf{g}_{\mathtt{cal}}) \Rightarrow \mathsf{p}' & (\mathsf{p} \rhd \mathsf{g}_{\mathtt{fun}}) \Rightarrow \mathsf{p}'' & (\mathsf{p} \circ (\mathsf{g}_{\mathtt{fun}} \circ \mathsf{g}'')) \Rightarrow \mathsf{g} \\ (\mathsf{f}', (\mathsf{p}', \mathsf{g}')) \in \Psi & \mathsf{g}_{\mathtt{fun}} = ((\mathsf{g}_{\mathtt{cal}} \circ \mathsf{g}') \circ \mathsf{g}_{\mathtt{ret}}) & \end{array}}{\Psi \vdash \{(\mathsf{p}, \mathsf{g})\}\, \mathsf{pc} :\ \mathtt{cal}\ \mathsf{l}\ \mathsf{f}'; \mathbb{I}} \ (\text{CAL})$$

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

$$\frac{(\mathsf{f}', (\mathsf{p}', \mathsf{g}')) \in \Psi \qquad (\mathsf{p} \rhd \mathsf{g}_{\mathtt{jmp}}) \Rightarrow \mathsf{p}' \qquad (\mathsf{p} \circ (\mathsf{g}_{\mathtt{jmp}} \circ \mathsf{g}')) \Rightarrow \mathsf{g}}{\Psi \vdash \{(\mathsf{p}, \mathsf{g})\}\, \mathsf{pc} :\ \mathtt{jmp}\ \mathsf{f}'} \ (\text{JMP})$$

Figure 9. $NCBP$ Inference Rules

$$\begin{array}{ll}
\mathsf{p} \Rightarrow \mathsf{g} \triangleq \forall \mathbb{S}. \mathsf{p}\mathbb{S} \rightarrow \exists \mathbb{S}', \mathsf{g}\mathbb{S}\mathbb{S}'; & \mathsf{p} \rhd \mathsf{g} \triangleq \lambda \mathbb{S}. \exists \mathbb{S}_0, \mathsf{p}\mathbb{S}_0 \wedge \mathsf{g}\mathbb{S}_0\mathbb{S} \\
\mathsf{p} \Rightarrow \mathsf{p}' \triangleq \forall \mathbb{S}. \mathsf{p}\mathbb{S} \rightarrow \mathsf{p}'\mathbb{S}; & \mathsf{g} \circ \mathsf{g}' \triangleq \lambda \mathbb{S}, \mathbb{S}''. \exists \mathbb{S}', \mathsf{g}\mathbb{S}\mathbb{S}' \wedge \mathsf{g}'\mathbb{S}'\mathbb{S}'' \\
\mathsf{g} \Rightarrow \mathsf{g}' \triangleq \forall \mathbb{S}, \mathbb{S}'. \mathsf{g}\mathbb{S}\mathbb{S}' \rightarrow \mathsf{g}'\mathbb{S}\mathbb{S}'; & \mathsf{p} \circ \mathsf{g} \triangleq \lambda \mathbb{S}, \mathbb{S}'. \mathsf{p}\mathbb{S} \wedge \mathsf{g}\mathbb{S}\mathbb{S}'
\end{array}$$

Figure 10. Connectors for p and g

for the current instruction ι should be found. At the same time, the remaining instruction sequence is well-formed with intermediate specification $(\mathsf{p}'', \mathsf{g}'')$;

We use g_ι to represent the state transition made by the instruction ι, which is defined in Figure 11 and Figure 7. Since NextS does not depend on the current program counter for these instructions "_" is used to represent arbitrary pc.

The rules use the definitions in Figure 10. The predicate $\mathsf{p} \rhd \mathsf{g}_\iota$ specifies the state resulting from the state transition g_ι, knowing the initial state satisfies p. It is the strongest post condition after g_ι. The composition of two transitions g and g' is represented as $\mathsf{g} \circ \mathsf{g}'$, and $\mathsf{p} \circ \mathsf{g}$ refines g with the knowledge that the initial state satisfies p.

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

$$\begin{array}{llll}
\mathsf{g}_{\mathtt{jpcT}} & \triangleq & \lambda \mathbb{S}, \mathbb{S}'. \mathsf{NextS}_{(\mathtt{jpc},\, \_)}\ \mathbb{S}\ \mathbb{S}' & (\text{where } \mathbb{S}.\mathbb{K}(\mathsf{r}_t) = 0) \\
\mathsf{g}_{\mathtt{jpcF}} & \triangleq & \lambda \mathbb{S}, \mathbb{S}'. \mathsf{NextS}_{(\mathtt{jpc},\, \_)}\ \mathbb{S}\ \mathbb{S}' & (\text{where } \mathbb{S}.\mathbb{K}(\mathsf{r}_t) \neq 0) \\
\mathsf{g}_{\mathtt{c}} & \triangleq & \lambda \mathbb{S}, \mathbb{S}'. \mathsf{NextS}_{(\mathtt{c},\, \_)}\ \mathbb{S}\ \mathbb{S}' & (\text{for all other } \mathtt{c})
\end{array}$$

Figure 11. Local State Transitions

Function Call and Return. Figure 12(a) illustrates the meaning of the specification (p, g) for the procedure `foo`. A pre-condition for an instruction sequence contains a predicate p specifying the current state, and a *guarantee* g describing the relation between the current state and the state at the return point of the current procedure. The guarantee g may cover multiple instruction sequences.

Figure 12(b) shows a procedure call to `bar` (point `B`) from `foo` at point `A` (label `pc` $=$ 5), with the return address `pc` $+$ `1` (point `D`). The specification of `bar` is $(\mathsf{p}_B, \mathsf{g}_B)$. Specifications at `A` and `D` are $(\mathsf{p}_A, \mathsf{g}_A)$ and $(\mathsf{p}_D, \mathsf{g}_D)$, where g_A governs the code segment `A`-`E` and g_D governs `D`-`E`.

To prove the well-formedness of program calling behavior, the following conditions should be satisfied:

- the state should satisfy the pre-condition of callee `bar`,
$$\forall \mathbb{S}, \exists \mathbb{S}'.\mathsf{p}_A\ \mathbb{S} \wedge \mathsf{g}_{\mathtt{cal}}\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathsf{p}_B\ \mathbb{S}';$$
- after `bar` returns, `foo` resumes from `D`,
$$\forall \mathbb{S}, \mathbb{S}'', \mathbb{S}', \mathbb{S}^*.\mathsf{p}_A\ \mathbb{S} \rightarrow \mathsf{g}_{\mathtt{cal}}\mathbb{S}\mathbb{S}' \rightarrow \mathsf{g}_B\mathbb{S}'\mathbb{S}^* \rightarrow \mathsf{g}_{\mathtt{ret}}\mathbb{S}^*\mathbb{S}'' \rightarrow \mathsf{p}_D\ \mathbb{S}'';$$
- the specification in `bar` for `A`-`E` should be satisfied,
$$\forall \mathbb{S}, \mathbb{S}'', \mathbb{S}''', \mathbb{S}', \mathbb{S}^*.\mathsf{p}_A\ \mathbb{S} \rightarrow \mathsf{g}_{\mathtt{cal}}\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathsf{g}_B\ \mathbb{S}'\ \mathbb{S}^* \rightarrow \mathsf{g}_{\mathtt{ret}}\ \mathbb{S}^*\ \mathbb{S}'' \rightarrow \mathsf{g}_D\ \mathbb{S}''\ \mathbb{S}''' \rightarrow \mathsf{g}\ \mathbb{S}\ \mathbb{S}''';$$

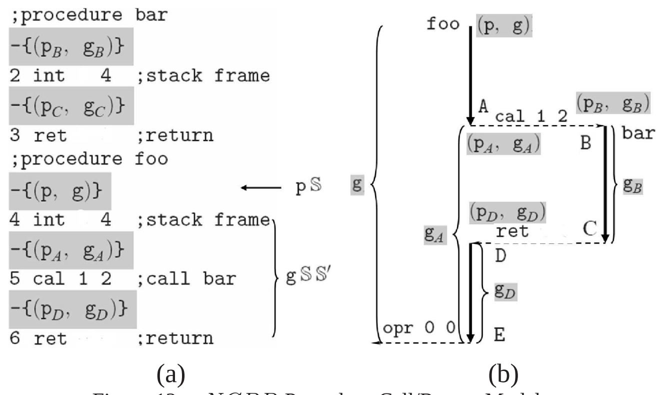
Then we can define guarantee $\mathsf{g}_{\mathtt{fun}}$for callee
$$\mathsf{g}_{\mathtt{fun}} \triangleq \lambda \mathbb{S}, \mathbb{S}''.\exists \mathbb{S}', \exists \mathbb{S}^*, \mathsf{g}_{\mathtt{cal}}\ \mathbb{S}\ \mathbb{S}' \wedge \mathsf{g}_B\ \mathbb{S}'\ \mathbb{S}^* \wedge \mathsf{g}_{\mathtt{ret}}\ \mathbb{S}^*\ \mathbb{S}''.$$
Thus, we can rewrite last two premises as:
$$\forall \mathbb{S}, \mathbb{S}''.\mathsf{p}_A\ \mathbb{S} \rightarrow \mathsf{g}_{\mathtt{fun}}\ \mathbb{S}\ \mathbb{S}'' \rightarrow \mathsf{p}_D\ \mathbb{S}'';$$
and
$$\forall \mathbb{S}, \mathbb{S}'', \mathbb{S}'''.\mathsf{p}_A\ \mathbb{S} \rightarrow \mathsf{g}_{\mathtt{fun}}\ \mathbb{S}\ \mathbb{S}'' \rightarrow \mathsf{g}_D\ \mathbb{S}''\ \mathbb{S}''' \rightarrow \mathsf{g}\ \mathbb{S}\ \mathbb{S}'''.$$

One of the major contributions of our work is the formulation of $\mathsf{g}_{\mathtt{fun}}$ for the callee procedure which hides the complexity of the stack and display. Above conditions are enforced by the CAL rule shown in Figure 9. It can be seen that $\mathsf{g}_{\mathtt{fun}}$ describes the states transition from the function call point to the return point. It should be noticed that we do not require a particular return value but only require that the stack contain a code pointer specified in local Ψ at the return state $\mathbb{S}''$, which is provable based on the knowledge of p and $\mathsf{g}_{\mathtt{fun}}$.

The RET rule simply requires that the function has finished its guaranteed transition at this point. So a state transition $\mathsf{g}_{\mathtt{ret}}$ should satisfy the remaining behavior of the callee function g. In this rule, we do not need to know any information about the return address. So it can be used to support modular certification of any callee function separately.

Stack Invariant. From the procedure call premises, the "well-formed control stack with depth n" can be defined as:
$$\begin{aligned}\mathsf{WFST}(0, \mathsf{g}, \mathbb{S}, \Psi) &\triangleq\ \neg\exists \mathbb{S}'.\ \mathsf{g}\ \mathbb{S}\ \mathbb{S}' \\ \mathsf{WFST}(n, \mathsf{g}, \mathbb{S}, \Psi) &\triangleq \forall \mathbb{S}'.\mathsf{g}\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathbb{S}'.\mathbb{K}(\mathsf{b}+2) \in \mathtt{dom}(\Psi) \\ &\qquad \wedge \mathsf{p}'\ \mathbb{S}' \wedge \mathsf{WFST}(n-1, \mathsf{g}', \mathbb{S}', \Psi) \\ &\text{where } (\mathsf{p}', \mathsf{g}') = \Psi(\mathbb{S}'.\mathbb{K}(\mathsf{b}+2)).\end{aligned}$$
When the stack has depth 0, we are in the outermost function which has no return code pointer. Thus, there exist no $\mathbb{S}'$ at which the function can return.

The stack invariant is defined as that, at each program point with specification (p, g), the program state $\mathbb{S}$ must satisfy p and there exists a well-formed control stack in $\mathbb{S}$:
$$\{(\mathsf{p}, \mathsf{g})\}\ \Psi'\ \mathbb{S} \triangleq \mathsf{p}\ \mathbb{S} \wedge \exists n.\mathsf{WFST}(n, \mathsf{g}, \mathbb{S}, \Psi).$$



```
;procedure bar
-{(p_B, g_B)}
2 int   4   ;stack frame
-{(p_C, g_C)}
3 ret       ;return
;procedure foo
-{(p, g)}
4 int   4   ;stack frame
-{(p_A, g_A)}
5 cal 1 2   ;call bar
-{(p_D, g_D)}
6 ret       ;return
```

Figure 12. $NCBP$ Procedure Call/Return Model

The actual depth of the control 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. As long as properly specified, the callee can update any other parts of the stack. The programmer should prove that this invariant holds at every step of program execution.

Other Instructions. The execution of `jpc` may either fall through or jump to the target code label, depending on whether the condition holds. So in the JPC rule, we use $\mathsf{g}_{\mathtt{jpcT}}$ and $\mathsf{g}_{\mathtt{jpcF}}$ to represent identity transitions with extra knowledge about the stack $\mathbb{K}$. We also need to know that p contains the ownership of the target stack cell. A direct jump is safe (rule JMP) 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 JPC.

Soundness of $NCBP$. Based on the progress and preservation lemmas, the soundness of $NCBP$ guarantees that the complete system never gets stuck as long as the initial state satisfies the program invariant that is defined by the WLD rule. Furthermore, the invariant will be always held during execution, from which we can derive rich properties of programs. The soundness of the program logic is proved following the syntactic approach in Coq proof assistant [13].

Lemma 3.1 (Progress) If $\Psi \vdash \{\mathsf{s}\}\mathbb{W}$, then there exists $\mathbb{W}'$, such that $\mathbb{W} \longmapsto \mathbb{W}'$.

Lemma 3.2 (Preservation) If $\Psi \vdash \{\mathsf{s}\}\mathbb{W}$, and $\mathbb{W} \longmapsto \mathbb{W}'$, then there exists s', $\Psi \vdash \{\mathsf{s}'\}\mathbb{W}'$.

Theorem 3.3 (Soundness of $NCBP$) If $\Psi \vdash \{\mathsf{s}\}\mathbb{W}$, then for all natural number n, there exists a program $\mathbb{W}'$ such that $\mathbb{W} \longmapsto^n \mathbb{W}'$.

IV. Recursive Nested Procedure Examples

In this section we certify a recursive nested factorial procedure implemented in IR to show how $NCBP$ can be used to support non-local and cross procedure access.

A. Callee of Recursive Procedure

Get Instruction Sequences. The source code and the IR program with its specifications for PCM are shown in Figure 1 (Section I).

$$
\begin{array}{ll}
\mathsf{True} & \triangleq \lambda\mathbb{S}.\mathsf{True} \\
\mathsf{validK}(\mathtt{sp},\mathtt{n}) & \triangleq \forall \mathtt{l} \in \{0\ldots n\}.\mathbb{K}(\mathtt{sp}+\mathtt{l}) \mapsto \_ \\
 & \\
\mathtt{NoG} & \triangleq \lambda\mathbb{S}.\lambda\mathbb{S}'.\mathsf{False} \\
\mathsf{Did} & \triangleq \forall \mathtt{l} \in \{0\ldots \mathsf{nd}\}.\mathbb{D}(\mathtt{l}) = \mathbb{D}'(\mathtt{l}) \\
\mathsf{Kid}(\mathtt{ls}) & \triangleq \forall \mathtt{l} \in \{\mathbb{D}(0)\ldots \mathtt{ls}-1\}.\mathbb{K}(\mathtt{l}) = \mathbb{K}'(\mathtt{l}) \\
\mathsf{Frmid}(\mathtt{ls},\mathtt{ls}') & \triangleq \forall \mathtt{l} \in \{0\ldots 3\}.\mathbb{K}(\mathtt{ls}+\mathtt{l}) = \mathbb{K}'(\mathtt{ls}'+\mathtt{l}) \\
\mathsf{Rid} & \triangleq \mathbb{R}(\mathsf{r}_b) = \mathbb{R}'(\mathsf{r}_b) \wedge \mathbb{R}(\mathsf{r}_t) = \mathbb{R}'(\mathsf{r}_t)
\end{array}
$$

Figure 13. Specification Macros for $NCBP$

$$
\begin{array}{ll}
\mathsf{p}_2 & \triangleq\ \mathsf{validK}(\mathsf{b}, 1+4*(\mathbb{K}(\mathsf{b_c}+5)+1)) \wedge (\mathsf{t}=\mathsf{b}) \\
 & \wedge(\mathbb{K}(\mathsf{b_c}+4) >= 1) \wedge (\mathbb{K}(\mathsf{b_c}+5) \in \{0\ldots 5\}) \\
\mathsf{g}_2 & \triangleq\ (\mathsf{t}=\mathsf{t}'-4) \wedge (\mathsf{b}=\mathsf{b}') \wedge \mathsf{Frmid}(\mathsf{b},\mathsf{b}) \wedge \mathsf{Did} \wedge \mathsf{Kid}(\mathsf{b_c}) \\
 & \wedge\mathsf{Frmid}(\mathsf{b_c},\mathsf{b_c}) \wedge (\mathbb{K}'(\mathsf{b_c}+4) = \mathbb{K}(\mathsf{b_c}+4)*(\mathbb{K}(\mathsf{b_c}+5))!) \\
\mathsf{p}_{16} & \triangleq\ \mathsf{validK}(\mathsf{b}, 1+4*(\mathbb{K}(\mathsf{b_c}+5)+1)) \wedge (\mathsf{t}=\mathsf{b}+4) \\
 & \wedge(\mathbb{K}(\mathsf{b_c}+4) = 5!) \wedge (\mathbb{K}(\mathsf{b_c}+5) = 0) \\
\mathsf{g}_{16} & \triangleq\ \mathsf{Rid} \wedge \mathsf{Frmid}(\mathsf{b},\mathsf{b}) \wedge \mathsf{Did} \wedge \mathsf{Kid}(\mathsf{b_c}) \wedge \mathsf{Frmid}(\mathsf{b_c},\mathsf{b_c}) \\
 & \wedge(\mathbb{K}'(\mathsf{b_c}+4) = \mathbb{K}(\mathsf{b_c}+4)*(\mathbb{K}(\mathsf{b_c}+5))!)
\end{array}
$$

where $\mathsf{b_c} \triangleq \mathbb{D}(\mathsf{nd}-1)$, is the stack base point of caller.

Figure 14. Specifications of Callee

Finding the instruction sequence is the first step to certify a program. From the definition in Figure 4, 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 two instruction sequences in callee program. Instructions with label from 2 to 15 form the first sequence. The second one is formed by the return instruction with label 16.

Write Specification for Instruction Sequences. Then the programmer needs to give the specification Ψ of the code heap, which is a finite mapping from code labels f to code specifications s which is a pair (p,g).

$NCBP$ specifications for the code heap are embedded in the code, enclosed by -{} in shadow box, see Figure 1 (Section I). Figure 13 shows definitions of macros used in the code specifications. $\mathsf{validK}(\mathtt{sp},\mathtt{n})$ means that the $\mathtt{n}+1$ stack cells those with addresses from $\mathtt{sp}$ to $\mathtt{sp}+n$ are valid and can be used in current function. Did means all the accessible display elements in $\mathbb{S}$ are not changed. $\mathsf{Kid}(\mathtt{ls})$ means all the stack cells from the bottom of outermost procedure to $\mathtt{ls}$ in $\mathbb{S}$ and $\mathbb{S}'$ are totally equal. $\mathsf{Frmid}(\mathtt{ls},\mathtt{ls}')$ means the stack frame cells those with addresses from $\mathtt{ls}$ to $\mathtt{ls}+3$ are equal to those from $\mathtt{ls}'$ to $\mathtt{ls}'+3$ respectively. Rid means all the registers *except* `pc` in $\mathbb{S}$ are preserved.

Following the inference rules, the code specifications should be given for these points: the head of an instruction sequence, the target labels of function call instruction `cal` and jump instructions (`jmp` and `jpc`), and the function call return address which is just after call instruction `cal`.

Figure 14 shows the specifications of this example. To simplify our presentation, we present the predicate p in the form of a proposition with free variables referring to components of the state $\mathbb{S}$. Also, we use k as shorthand for the proposition $\mathsf{k}\ \mathbb{K}$ when there is no confusion.

The specification of this procedure is given as $(\mathsf{p}_2,\mathsf{g}_2)$.

$$
\begin{array}{ll}
\mathsf{p}_{17} & \triangleq\ \mathsf{validK}(\mathsf{b}, 7+4*6) \wedge (\mathsf{t}=\mathsf{b}) \\
\mathsf{g}_{17} & \triangleq\ \mathsf{Kid}(\mathsf{b}) \wedge \mathsf{Frmid}(\mathsf{b},\mathsf{b}') \wedge \mathsf{Did} \wedge (\mathsf{t}=\mathsf{t}'-6) \wedge (\mathsf{b}=\mathsf{b}') \\
\mathsf{p}_{23} & \triangleq\ \mathsf{validK}(\mathsf{b}, 7+4*5) \wedge (\mathsf{t}=\mathsf{b}+6) \wedge (\mathbb{K}(\mathsf{b}+4) = 5!) \\
 & \wedge(\mathbb{K}(\mathsf{b}+5) = 0) \\
\mathsf{g}_{23} & \triangleq\ \mathsf{Kid}(\mathsf{b}) \wedge \mathsf{Frmid}(\mathsf{b},\mathsf{b}') \wedge \mathsf{Did} \wedge \mathsf{Rid} \\
\mathsf{p}_2 & \triangleq\ ? \qquad\qquad\qquad\qquad \mathsf{g}_2 \ \triangleq\ ?
\end{array}
$$

Figure 15. Specifications of Caller

From p_2, we know that the values of r_t and r_b are equal, there is enough stack space for this recursive procedure to run, and the values of variables `r` and `n` which stored in caller's stack are inside the proper scope. The guarantee g_2 specifies the behavior of the function:

- only the caller stack and local stack are updated;
- the stack frames of caller and callee are not updated;
- the display is not updated;
- the register r_b is not updated while $\mathtt{t} = \mathtt{t}'-4$ which reserves stack cells for local stack frame; and,
- the non-local variables `r` and `n` which are saved in caller's stack fulfill the loop invariant;

$(\mathsf{p}_{16},\mathsf{g}_{16})$ is the specification of the rest of this procedure. The pre-condition p_{16} says that the relationship between the values of r_t and r_b is $(\mathsf{t}=\mathsf{b}+4)$, the stack space is still available and, the results which store in caller's stack must be $\mathtt{r} = 5!$ and $\mathtt{n} = 0$. The guarantee g_{16} looks the same as g_2 except the relation about the value of r_t which reflects the state transition of stack reserving instruction "`lit04`".

Certify and Link Them Together. To check the well-formedness of an instruction sequence beginning with ι, a programmer's task includes applying the appropriate inference rules and finding intermediate assertions such as $(\mathsf{p}',\mathsf{g}')$, which serves both as the post-condition for ι and as the pre-condition 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 links all code heaps into one single well-formed global one.

Support Modular Certification. All the code specifications Ψ used in $NCBP$ rules are the *local* specifications for the current module. Thus, $NCBP$ 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 pre-condition. The RET rule for the instruction "`ret`" does not have any constraint on return value either.

B. Caller of Factorial Procedure

The caller and its specifications for the recursive nested factorial example are shown in Figure 1 (Section I). Figure 15 gives the specification definition of this example.

This procedure just initializes the variables `r` and `n`, and then calls procedure `fact`. The specification at the entry point is $(\mathsf{p}_{17},\mathsf{g}_{17})$. The pre-condition p_{17} simply says that the values of r_t and r_b are equal and there is enough stack space for this procedure to run. The guarantee g_{17} specifies the behavior of the caller procedure:

- the stack cells except the local stack are not updated;
- the callee stack frame is not updated;
- the display is not updated; and,
- the register r_b are not updated while $r_t = {r_t}' - 6$ which reserves cells for local stack frame and local variables.

The specification of the return point is (p_{23}, g_{23}). p_{23} means that there are still enough local stack space, the relation between the values of r_t and r_b is $t = b + 6$, and local variables $\mathtt{r}$ and $\mathtt{n}$ are $5!$ and 0. The guarantee g_{23} also specifies that the non-local stack, the local stack frame, the display and the register file are all the same.

From the CAL rule, we know that (p_2, g_2), the specification of the callee's entry point should be added here to prove caller procedure. The specification (p_2, g_2) in Figure 14 can be used here. Furthermore, the specification of procedure's entry point defines function interface. Caller can use any callees which share the same interface.

V. Implementation and Further Extensions

Implementation with Coq. The program logic presented in this paper has been applied to IR programs for the stack-based virtual machine PCM which supports nested procedure non-local access with display.

We have formalized PCM, its operational semantics, and the program logic in the Coq proof assistant. The syntax of the virtual machine is encoded in Coq using inductive definitions. Operational semantics of PCM and all the inference rules of $NCBP$ are defined as inductive relations. The soundness of the framework itself is formalized and proved in Coq following the syntactic approach. The proof is also formalized and implemented in Coq and is machine-checkable. These examples are implemented directly in IR and are hard to certify using other existing approaches. Manually optimized IR code or code generated by optimizing compilers can also be certified using our systems.

The implementation includes around 300 lines of Coq encoding of PCM including stack and display definition, 850 lines encoding of PCM operational semantics, 230 lines encoding of $NCBP$ rules, and 950 lines of Coq tactics for the soundness proof. We have written several hundred lines of Coq tactics to certify practical examples, including procedure call/return, and recursive nested procedure. The implementation also includes more than 2300 lines of reusable basic facilities, including lemmas and tactics for partial mappings and Separation Logic assertions.

According to our experience, the size of the proof scripts, in terms of the number of lines of Coq tactics, is huge compared with the size of the IR code. However, as observed by [14], the length of proof is probably a poor metric of complexity because of the redundancy of the proof.

Extensions and Future Work. The most important work is building a proof-transforming compiler based on our logic to translate certified IR programs to assembly codes with proofs. This idea is similar to Barthe's certificate translation compiler [15], but we use CiC, the underlying higher-order logic in Coq, as our specification language. We plan to develop a prototype compiler to translate our proof into the proof for machine code.

Component Name	Number of lines
Basic Utility Definitions & Lemmas	2,354
Machine Definition & Lemmas	275
Operational Semantics & Lemmas	854
NCBP-PCM Rules & Lemmas	223
NCBP-PCM & Soundness	966
PCM Stack Related Lemmas	2241
IR Examples Source Code	164
Callee Fact Spec. & Proof	1121
Caller Main Spec. & Proof	522
Total	8720

Figure 16. The Verified Package in Coq

Our logic system does not support concurrency yet. It is actually an easy work to extend the machine to support concurrency. But it is difficult to define a simple logic system to modularly certify concurrent IR programs. We will try it in the near future. Another important and useful extension is the support of the object-oriented features such as objects, references, methods, and inheritance.

VI. Related Work and Conclusion

Logic for Intermediate Representation and Bytecode. Program logic for intermediate representation is the essential component of the certified compilers and the proof-transforming compilers.

Appel and Blazy [16] designed a Separation Logic system for Cminor, the IR of the certified CompCert compiler. The Cminor programs that are proved in Separation Logic can be compiled by the certified compiler.

Bytecode can be used as the IR of a modern compiler. Bytecode Modeling Language (BML) [17] 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 others are focus on the development of the sound proof system. Quigley [7] defined a programming logic for bytecode programs within Isabelle that allows the proof of bytecode programs containing loops. MRG project [18] presented the resource-aware operational semantics of an abstract fragment of JVM Language (named Grail), the program logic, and the proof. Bannwart and Müller [9] presented a program logic for a bytecode language similar to Java bytecode and the .NET CIL that supports lots of object-oriented features such as objects, methods, and inheritance.

Benton [8] proposed a typed, compositional logic for a stack-based abstract machine to certify bytecode programs which are written in an imperative subset of .NET CIL. But none of these systems can be used to verify a program which involves recursive nested procedure.

There have been some efforts [3], [4], [5] on building logic system for low-level (assembly) programs. These results are very useful for us to transform our IR code and the proof into low-level codes and their proof.

Conclusion. This paper presents a Hoare-style framework for modular specifying and certifying IR programs with complex stack-based control abstractions and unstructured control flow, including while loop, procedure call/return, recursive procedure, and even nested procedure. This virtual machine features nested procedure with variables from external scopes via a display. For each control abstraction, we have formalized its invariants and shown how to certify its implementation. Using the producer/consumer model, we studied the stack-based procedure call/return control abstractions. In a natural and general way, we formalize it as the general formula to describe caller's state transition from function call point to return point. It can be used as a general framework for other machines and other calling convention with proper instances of guarantee g for call and return instructions.

This logic system is fully mechanized in the Coq proof assistant. It provides a foundation for reasoning about IR 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 IR programs which run on stack-based virtual machine.

ACKNOWLEDGMENT

The authors thank anonymous referees for suggestions and comments on an earlier version of this paper. We also 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 and SCAP. Yingying Sun and Yunmin Zhu proved some lemmas in our Coq implementation. This work is supported in part by National Natural Science Foundation of China (No. 60573017, No. 90818019 and No. 90816006) and Hi-Tech Research and Development Program of China (No. 2008AA01Z102). 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] C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, and K. Cline, "A certifying compiler for Java," in *Proc. PLDI'00*. ACM Press, 2000, pp. 95–107.

[2] D. von Oheimb, "Hoare logic for Java in Isabelle/HOL," *Concurrency and Computation: Practice and Experience*, vol. 13, no. 13, pp. 1173–1214, 2001.

[3] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni, "Modular verification of assembly code with stack-based control abstractions," in *Proc. PLDI'06*. ACM Press, 2006, pp. 401–414.

[4] A. Saabas and T. Uustalu, "A compositional natural semantics and hoare logic for low-level languages," in *In: Proceedings of the Second Workshop on Structured Operational Semantics*. Elsevier, 2005, pp. 151–168.

[5] G. Tan and A. W. Appel, "A compositional logic for control flow," in *VMCAI'06*, ser. LNCS, vol. 3855. Springer, 2006, pp. 80–94.

[6] N. Wirth, *Algorithms + Data Structures=Programs*. Prentice Hall, 1976.

[7] C. L. Quigley, "A programming logic for java bytecode programs," in *Proc. TPHOLs'03*. Springer-Verlag, 2003, pp. 41–54.

[8] N. Benton, "A typed, compositional logic for a stack-based abstract machine," in *Proc. APLAS'05*. Springer-Verlag, 2005, pp. 364–380.

[9] F. Bannwart and P. Müller, "A program logic for bytecode," in *Proc. Bytecode05*. Elsevier, 2005, pp. 255–273.

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

[11] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proc. LICS'02*. IEEE Computer Society, Jul. 2002, pp. 55–74.

[12] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 26, no. 1, pp. 53–56, Oct. 1969.

[13] Y. Dong, S. Wang, L. Zhang, and P. Yang, "Modular certification of low-level intermediate representation programs," Tsinghua University, Beijing, China, Tech. Rep., Sep. 2008, http://soft.cs.tsinghua.edu.cn/∼dongyuan/verify.

[14] A. McCreight, Z. Shao, C. Lin, and L. Li, "A general framework for certifying garbage collectors and their mutators," in *Proc. PLDI'07*. ACM Press, June 2007, pp. 468–479.

[15] G. Barthe and C. Kunz, "Certificate translation in abstract interpretation," in *Proc. ESOP'08*, ser. LNCS. Springer-Verlag, 2008.

[16] A. W. Appel and S. Blazy, "Separation logic for small-step c minor," in *TPHOLs'07*. Springer-Verlag, 2007.

[17] L. Burdy and M. Pavlova, "Java bytecode specification and verification," in *Proc. SAC06*. ACM Press, 2006.

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