A Simple Model for Certifying Assembly Programs with First-Class Function Pointers

Wei Wang[†] Zhong Shao[‡] Xinyu Jiang[†] Yu Guo[†]

[†]*School of Computer Science and Technology*
University of Science and Technology of China
Hefei, Anhui 230026, China
{wqsh, wewewe}@mail.ustc.edu.cn guoyu@ustc.edu.cn

[‡]*Department of Computer Science*
Yale University
New Haven, CT 06520-8285, U.S.A
zhong.shao@yale.edu

***Abstract*—First-class function pointers are common in low-level assembly languages. Higher-level features such as closures, virtual functions, and call-backs are all compiled down to assembly code with function pointers. Function pointers are, however, hard to reason about. Previous program logics for certifying assembly programs either do not support first-class function pointers, or follow Continuation-Passing-Style reasoning which does not provide the same partial correctness guarantee as in high-level languages. In this paper, we present a simple semantic model for certifying the partial correctness property of assembly programs with first-class function pointers. Our model does not require any complex domain-theoretical construction, instead, it is based on a novel step-indexed, direct-style operational semantics for our assembly language. From the model, we derive a new program logic named ISCAP (or Indexed SCAP). We use an example to demonstrate the power and simplicity of ISCAP. The semantic model, the ISCAP logic, and the soundness proofs have been implemented in the Coq proof assistant.**

I. Introduction

Low-level languages such as C and assembly do not have a rich type system to describe whether a function pointer indeed points to a valid block of code satisfying a desirable specification. Previous work on Foundational Proof-Carrying Code (FPCC) [1]–[3], Typed Assembly Languages (TAL) [4], and Certified Assembly Programming with Embedded Code Pointers (XCAP) [5] can support first-class code pointers but only by rewriting all programs into Continuation-Passing Style (CPS) [6]. Under CPS, function returns are viewed as indirect calls to continuation functions. For example, in the following skeletal assembly program,

```
      movi r,foo          |    bar : -{spec_0}
      jmp bar             |          <* * CodeA * *>
      ...                 |          callr r
foo : -{spec_1}           |          -{spec_2}
      <* * CodeB * *>     |          <* * CodeC * *>
      ret                 |          ret*
CodeA :  the code which saves the return address
CodeB :  the code which requires the condition that r_1 > 0
         and decreases r_1 by one
CodeC :  the code which requires nothing and increments r_1 by one
ret* :   the code after the ret requires r_1 < 2
```

the function pointer `foo` was moved into register r and then invoked inside the `bar` function. Under CPS-based systems, each function, continuation, or program point can be certified using a single state predicate (similar to a pre-condition in Hoare logic [7]); because a call to function or continuation never returns, no post-condition is necessary. The above assembly programs can be specified as follows:

$$\begin{array}{lcl} \text{spec}_0 & \triangleq & \mathsf{cptr}(r, \text{spec}_1) \wedge \mathsf{cptr}(ra, r_1 < 2) \wedge r_1 = 1 \\ \text{spec}_1 & \triangleq & \mathsf{cptr}(ra, r_1 < 1) \wedge r_1 = 1 \wedge \ldots \\ \text{spec}_2 & \triangleq & r_1 < 1 \wedge \mathsf{cptr}(ra, r_1 < 2) \end{array}$$

Here $\mathsf{cptr}(r, p)$ means that r contains a code pointer with pre-condition p; and ra denotes the return address register. The pre-condition of `bar`, i.e., spec_0, needs to worry about the safety of calling function `foo` and all the code after `bar` returns. The pre-condition of `foo` not only ensures that the value in r_1 is positive, but the safety of all the code executed after the function `foo` returns. Note that in spec_1 the information about the return address of `bar` is also needed (though we have omitted it here).

CPS-style reasoning makes the underlying language more uniform, but it obscures high-level program structures. From the specifications above, one cannot tell when a code pointer is a continuation and when a function actually returns. It is difficult to certify large software using CPS-style reasoning.

In this paper, we present a new program logic, named ISCAP (or Indexed SCAP [8]), for reasoning about the partial correctness property of assembly programs with first-class function pointers. We show how to certify unstructured assembly programs without resorting to CPS-based reasoning, so function abstraction is still preserved. We achieve this by combining a novel direct-style mixed-step operational semantics (for assembly programs) with step-indexing [1]. We show how our techniques can be used to build a simple semantic model for first-class function pointers. Under ISCAP, each assembly code block is specified with a pre-condition p and a post-condition g; here p ensures the safe execution from the current program point to the end of the underlying function; and g specifies how the program state changes from the current program point to the end of the current function. Under ISCAP, the specifications for the example above would be as follows:

$$\begin{array}{lcl} \text{spec}_0 & \triangleq & (\mathsf{fptr}(r, \text{spec}_1) \wedge r_1 > 0,\ r_1' = r_1) \\ \text{spec}_1 & \triangleq & (r_1 > 0,\ r_1' = r_1 - 1 \wedge ra' = ra) \\ \text{spec}_2 & \triangleq & (\mathit{True},\ r_1' = r_1 + 1) \end{array}$$

Here, the assertions before the commas are pre-conditions and the ones after the commas are post-conditions. In all post-conditions, r denotes the value in register r of the starting states (the current point) and r' denotes the value in register r of the ending states (the point when the current function returns). The pre-condition of spec_0 says that the function `bar` will execute safely if the value in r_1 is positive and r contains a function pointer with specification spec_1. The post-condition of `bar` ensures that the value in r_1 will not be changed. The specification spec_1 says that `foo` will execute safely if r_1 is positive, and it will decrease r_1 by one and keeps the return address register *ra* unchanged. The specification spec_2 says that the code `CodeC` will execute safely and it will increment r_1 by one. Note that the specifications for `bar` and `foo` do not need to worry about any code executed after the (function) return. The specifications are more intuitive and follow the high-level function abstraction.

Our paper makes the following new contributions:

- We present a new program logic ISCAP that supports modular verification of the partial correctness property for assembly programs with first-class function pointers. Previous work on SCAP [8] also supports direct-style reasoning of function call and return in assembly languages, but it does not support first-class function pointers.
- In Hoare-style logics [7], [9], partial correctness of a program with pre-condition p and post-condition q guarantees that under any state satisfying p, the program can execute safely without fault, and if the program terminates, the ending state will satisfy q. Under CPS-based reasoning [1], [2], [4], [5], the assembly code has no notion of "finishing"; partial correctness here only guarantees that if the "post-condition" (i.e., the pre-condition for the return continuation) specifies a safe return point then the pre-condition also specifies a safe program point. Our system fixes this limitation by introducing direct-style mixed-step operational semantics. More specifically, we give function call a big-step semantics but this semantics is based on the small-step machine-level semantics so the partial correctness property we prove is indeed designed for the assembly language itself, not any high-level extension. We also use a binary state relation—an *action*—as the post-condition which avoids the use of auxiliary variables [8].
- Our new program logic is proven sound using a simple semantic model built on top of our operational semantics. It does not require any domain-theoretical construction [10], [11]. A key technical challenge, which we solve in this paper, is to decide how to apply step-indexing techniques [1] to define the semantics of actions. Each action (or post-condition) specifies two states and it is unclear how many index arguments it should take. We show that only one index (for the ending state) is necessary to build a powerful and sound semantic model.
- We have implemented our ISCAP logic and proved its soundness in the Coq proof assistant [12]. Following previous CAP-like logics [13], we use shallow embedding to define our assertion languages (i.e., both pre- and post-conditions are just Coq predicates). The soundness of ISCAP is proven following a pure semantic approach: all the inference rules for ISCAP are proven as lemmas based on their semantic definitions so the program logic is more extensible.

$$
\begin{array}{llll}
(\mathit{World}) & \mathbb{W} & ::= & (\mathbb{C}, \mathbb{S}, pc) \\
(\mathit{CHeap}) & \mathbb{C} & \in & \mathit{Label} \rightharpoonup_{\text{fin}} \mathit{Instr} \\
(\mathit{Instr}) & c & ::= & sc \mid bc \mid \texttt{jmp}\ f \mid \texttt{ret} \\
(\mathit{SInstr}) & sc & ::= & \texttt{add}\ r, r, r \mid \texttt{multi}\ r, r, z \\
 & & & \mid \texttt{lw}\ r, z(r) \mid \texttt{sw}\ z(r), r \\
 & & & \mid \texttt{mov}\ r, r \mid \texttt{movi}\ r, z \\
 & & & \mid \texttt{call}\ f \mid \texttt{callr}\ r \\
 & & & \mid \texttt{alloc}\ r, n \mid \texttt{free}\ r, n \\
(\mathit{BInstr}) & bc & ::= & \texttt{bgtz}\ r, f \mid \ldots \\
(\mathit{State}) & \mathbb{S} & \in & \mathit{Res} \rightharpoonup_{\text{fin}} \mathit{Value} \\
(\mathit{Res}) & s & ::= & r \mid l \\
(\mathit{Reg}) & r & ::= & \mathit{rv} \mid \mathit{ra} \mid a_i \mid s_i \mid t_i \mid \mathit{rz} \\
(\mathit{Value}) & v & ::= & z(\text{integers}) \\
(\mathit{Addr}) & l & ::= & n(\text{natural numbers}) \\
(\mathit{Label}) & f, pc & ::= & n(\text{natural numbers})
\end{array}
$$

Fig. 1. Syntax of the machine

In the rest of this paper, we first present our machine model and its operational semantics in Sec. II. We define the assertion language, the inference rules of ISCAP, and the underlying semantic model in Sec. III. We show how to apply ISCAP to certify first-class function pointers using an example in Sec. IV. Finally, we describe related work and conclude.

II. The Machine Model

In this section, we first present an assembly-level machine model and its small-step operational semantics, both of which are easily adaptable to the MIPS or X86 architecture. We then extend the semantics with step-indexing and present a direct-style mixed-step operation semantics that can be used to reason about the partial correctness of first-class function pointers.

A. The Raw Machine with its Small-Step Semantics

Fig. 1 provides the syntax of our raw machine. A machine world consists of a code heap, a machine-storage state, and a program counter. The code heap is a partial mapping from code pointers to instructions. Since we do not consider self-modifying code, we abstract the code heap out of the machine storage. The program counter, which is a code label, always points to the current execution point. We have sequential instructions, branch instructions, direct jumps, and function return instructions. The sequential instructions include move, arithmetic, memory allocation and free, and (direct and indirect) function call instructions. The machine storage is modeled as a partial mapping from resources to values, while resources contain registers and memory addresses. We treat registers as resources, following the variables-as-resources approach [14]. For simplicity, we name registers based on how they are used; we will use the following naming conventions in the rest of this paper:

if $\mathbb{C}(pc) =$	$(\mathbb{C}, \mathbb{S}, pc) \longmapsto$	when
`call` f	$(\mathbb{C}, \mathbb{S}\{ra \hookrightarrow pc+1\}, f)$	$ra \in dom(\mathbb{S})$
`callr` r	$(\mathbb{C}, \mathbb{S}\{ra \hookrightarrow pc+1\}, \mathbb{S}(r))$	$\{ra, r\} \in dom(\mathbb{S})$
`jmp` f	$(\mathbb{C}, \mathbb{S}, f)$	
`ret`	$(\mathbb{C}, \mathbb{S}, \mathbb{S}(ra))$	$ra \in dom(\mathbb{S})$
`bgtz` r, f	$(\mathbb{C}, \mathbb{S}, f)$ $(\mathbb{C}, \mathbb{S}, pc+1)$	$\mathbb{S}(r) > 0$ $\mathbb{S}(r) \leq 0$
other sc	$(\mathbb{C}, \mathbb{S}', pc+1)$	$\mathsf{ReqS}_{sc}\ \mathbb{S}$ and $\mathsf{NextS}_{sc}\ \mathbb{S}\ \mathbb{S}'$

where

if $sc =$	$\mathsf{NextS}_{sc}\ \mathbb{S}\ \mathbb{S}'$ where $\mathbb{S}' =$	$\mathsf{ReqS}_{sc}\ \mathbb{S} =$
`multi` r, r_1, z	$\mathbb{S}\{r \hookrightarrow \mathbb{S}(r_1) \times z\}$	$\{r, r_1\} \in dom(\mathbb{S})$
`add` r, r_1, r_2	$\mathbb{S}\{r \hookrightarrow \mathbb{S}(r_1) + \mathbb{S}(r_2)\}$	$\{r, r_1, r_2\} \in dom(\mathbb{S})$
`mov` r, r_1	$\mathbb{S}\{r \hookrightarrow \mathbb{S}(r_1)\}$	$\{r, r_1\} \in dom(\mathbb{S})$
`movi` r, z	$\mathbb{S}\{r \hookrightarrow z\}$	$r \in dom(\mathbb{S})$
`sw` $z(r), r_1$	$\mathbb{S}\{\mathbb{S}(r) + z \hookrightarrow \mathbb{S}(r_1)\}$	$\{r, r_1, \mathbb{S}(r) + z\} \in dom(\mathbb{S})$
`lw` $r, z(r_1)$	$\mathbb{S}\{r \hookrightarrow \mathbb{S}(\mathbb{S}(r_1) + z)\}$	$\{r, r_1, \mathbb{S}(r_1) + z\} \in dom(\mathbb{S})$
`alloc` r, n	$\mathbb{S}\{r \hookrightarrow l\} \uplus \{l \rightsquigarrow \_, \ldots, l+n-1 \rightsquigarrow \_\}$	$r \in dom(\mathbb{S})$
`free` r, n	$\mathbb{S}_0$ when $\mathbb{S} = \mathbb{S}_0 \uplus \{l \rightsquigarrow \_, \ldots, l+n-1 \rightsquigarrow \_\}$	$\{r, l, \ldots, l+n-1\} \in dom(\mathbb{S})$

Fig. 2. Small step semantics

rv	return value	*ra*	return address
$a_0, a_1, \ldots$	arguments	$s_0, s_1, \ldots$	callee saved
$t_0, t_1, \ldots$	caller saved	*rz*	always zero

In Fig. 2 we define a small-step semantics for our machine. Here $dom(\mathbb{S})$ is a set of all the domain resources of $\mathbb{S}$. We use $\mathbb{S}\{s \hookrightarrow v\}$ to denote the new state by remapping s of $\mathbb{S}$ to v while keeping others unchanged. A state $\{s_1 \rightsquigarrow v_1, \ldots, s_n \rightsquigarrow v_n\}$ has domain $\{s_1, \ldots, s_n\}$ and it maps s_i to v_i. We use $\mathbb{S}_1 \perp \mathbb{S}_2$ to mean the domain of $\mathbb{S}_1$ and $\mathbb{S}_2$ are disjoint. We define that $\mathbb{S}_1 \uplus \mathbb{S}_2$ is the disjoint union of $\mathbb{S}_1$ and $\mathbb{S}_2$. It is defined only when the domain of the two states are disjoint.

$$\mathbb{S}_1 \perp \mathbb{S}_2 \triangleq dom(\mathbb{S}_1) \cap dom(\mathbb{S}_2) = \emptyset$$

$$\mathbb{S}_1 \uplus \mathbb{S}_2 = \begin{cases} \mathbb{S}_1 \cup \mathbb{S}_2, & \text{if } \mathbb{S}_1 \perp \mathbb{S}_2 \\ \text{undefined}, & \text{otherwise} \end{cases}$$

For those sequential instructions that are not function calls, we define `NextS` and `ReqS` for each of them. The predicate ReqS_{sc} describes the state requirement for the safe execution of the instruction sc; the action NextS_{sc} defines how sc changes the state.

B. Indexed Machine and Direct-Style Operational Semantics

We borrow the idea of step-indexing [1], [15] and define a step-indexed assembly machine to approximate the behavior of programs that may contain infinite loops. In this paper, to make our semantics clearer, we directly introduce a step index into the machine world. We use $\bar{\mathbb{W}}$ to denote a machine world with an index to create an indexed machine world.

$$\begin{array}{lrcl} (\mathit{IWorld}) & \bar{\mathbb{W}} & ::= & (\mathbb{W}, i) \\ (\mathit{Index}) & i, j & ::= & n (\text{natural numbers}) \end{array}$$

One way to understand the step index is to view it as the number of tokens we have available in order to keep the machine running. Each machine step will cost one token, so the step index is decremented at each step. Since we do not

$$\frac{\mathbb{C}(pc) \notin \{\texttt{call}, \texttt{callr}\} \quad (\mathbb{C}, \mathbb{S}, pc) \longmapsto (\mathbb{C}, \mathbb{S}', pc')}{((\mathbb{C}, \mathbb{S}, pc), i+1) \downarrow ((\mathbb{C}, \mathbb{S}', pc'), i)}$$

$$\frac{\mathbb{C}(pc) = \texttt{call}\ f \quad ((\mathbb{C}, \mathbb{S}\{ra \hookrightarrow pc+1\}, f), i) \Downarrow \bar{\mathbb{W}}}{((\mathbb{C}, \mathbb{S}, pc), i+1) \downarrow \bar{\mathbb{W}}}$$

$$\frac{\mathbb{C}(pc) = \texttt{callr}\ r \quad ((\mathbb{C}, \mathbb{S}\{ra \hookrightarrow pc+1\}, \mathbb{S}(r)), i) \Downarrow \bar{\mathbb{W}}}{((\mathbb{C}, \mathbb{S}, pc), i+1) \downarrow \bar{\mathbb{W}}}$$

$$\frac{\mathbb{C}(pc) \neq \texttt{ret} \quad ((\mathbb{C}, \mathbb{S}, pc), i) \downarrow \bar{\mathbb{W}}' \quad \bar{\mathbb{W}}' \Downarrow \bar{\mathbb{W}}}{((\mathbb{C}, \mathbb{S}, pc), i) \Downarrow \bar{\mathbb{W}}}$$

$$\frac{\mathbb{C}(pc) = \texttt{ret} \quad ((\mathbb{C}, \mathbb{S}, pc), i) \downarrow \bar{\mathbb{W}}}{((\mathbb{C}, \mathbb{S}, pc), i) \Downarrow \bar{\mathbb{W}}}$$

Fig. 3. Direct-style mixed-step operational semantics

$$\frac{}{\mathit{ISafe}(\mathbb{W}, 0)} \qquad \frac{}{\mathit{FSafe}(\mathbb{W}, 0)}$$

$$\frac{\mathbb{C}(pc) \notin \{\texttt{call}, \texttt{callr}\} \quad \exists \bar{\mathbb{W}}.\ ((\mathbb{C}, \mathbb{S}, pc), i) \downarrow \bar{\mathbb{W}}}{\mathit{ISafe}((\mathbb{C}, \mathbb{S}, pc), i)}$$

$$\frac{\begin{array}{c}\mathbb{C}(pc) = \texttt{call}\ f \quad \bar{\mathbb{W}} = ((\mathbb{C}, \mathbb{S}\{ra \hookrightarrow pc+1\}, f), i) \\ \mathit{FSafe}(\bar{\mathbb{W}}) \quad \forall \bar{\mathbb{W}}'.\ \bar{\mathbb{W}} \Downarrow \bar{\mathbb{W}}' \rightarrow \bar{\mathbb{W}}'.pc = pc+1\end{array}}{\mathit{ISafe}((\mathbb{C}, \mathbb{S}, pc), i+1)}$$

$$\frac{\begin{array}{c}\mathbb{C}(pc) = \texttt{callr}\ r \quad \bar{\mathbb{W}} = ((\mathbb{C}, \mathbb{S}\{ra \hookrightarrow pc+1\}, \mathbb{S}(r)), i) \\ \mathit{FSafe}(\bar{\mathbb{W}}) \quad \forall \bar{\mathbb{W}}'.\ \bar{\mathbb{W}} \Downarrow \bar{\mathbb{W}}' \rightarrow \bar{\mathbb{W}}'.pc = pc+1\end{array}}{\mathit{ISafe}((\mathbb{C}, \mathbb{S}, pc), i+1)}$$

$$\frac{\begin{array}{c}\mathbb{C}(pc) \neq \texttt{ret} \quad \mathit{ISafe}((\mathbb{C}, \mathbb{S}, pc), i) \\ \forall \bar{\mathbb{W}}.\ ((\mathbb{C}, \mathbb{S}, pc), i) \downarrow \bar{\mathbb{W}} \rightarrow \mathit{FSafe}(\bar{\mathbb{W}})\end{array}}{\mathit{FSafe}((\mathbb{C}, \mathbb{S}, pc), i)}$$

$$\frac{\mathbb{C}(pc) = \texttt{ret} \quad \mathit{ISafe}((\mathbb{C}, \mathbb{S}, pc), i)}{\mathit{FSafe}((\mathbb{C}, \mathbb{S}, pc), i)}$$

Fig. 4. Safe indexed worlds for sequential instructions or functions

care about how the machine behaves after the index becomes 0, we treat the world with index 0 as a safe world.

Executing a machine-level program is always done on a per-instruction basis. An assembly program, however, does contain some structure. We define a code block $\mathbb{I}$ as a list of instructions ending with a direct jump or a return (implemented as an indirect jump to a specific return address register *ra*).

$$\mathbb{I} \quad ::= \quad sc;\mathbb{I} \mid bc;\mathbb{I} \mid \texttt{jmp}\ f \mid \texttt{ret}$$

Each assembly program is divided into a number of code blocks. Each code block can transfer control to other code blocks by executing a function call, return, branch, or direct jump instruction. Given an address label in the code heap, we can extract a code block starting from that label by using the following definition:

$$\texttt{cb}(\mathbb{C}, f) \quad \triangleq \quad \begin{cases} \mathbb{C}(f) & \text{if } \mathbb{C}(f) \in \{\texttt{jmp},\texttt{ret}\} \\ \mathbb{C}(f); \texttt{cb}(\mathbb{C}, f+1) & \text{otherwise} \end{cases}$$

A function definition in an assembly language often consists of a set of code blocks. Function pointers are no different from regular address labels.

To reason about the partial correctness property of assembly-level functions, we introduce a new direct-style mixed-step operational semantics in Fig. 3. This new semantics is built on top of the small-step semantics of our indexed machine. Here $\bar{\mathbb{W}} \downarrow \bar{\mathbb{W}}'$ denotes that starting from an indexed world $\bar{\mathbb{W}}$, executing the current instruction may result in $\bar{\mathbb{W}}'$. Note, here, executing a function call instruction also includes the complete execution of the called function.

The judgment $\bar{\mathbb{W}} \Downarrow \bar{\mathbb{W}}'$ denotes that starting from an indexed world $\bar{\mathbb{W}}$, finishing the execution of the rest of the current function may lead to $\bar{\mathbb{W}}'$. Other than the return instruction, everything else (including the branch and direct jump instructions) follows the big-step semantics; a function body finishes its execution until it returns.

We define *ISafe*$(\bar{\mathbb{W}})$ to mean that $\bar{\mathbb{W}}$ is safe to execute the current instruction and *FSafe*$(\bar{\mathbb{W}})$ to mean that $\bar{\mathbb{W}}$ is safe to execute the rest of the current function. The formal definition is in Fig. 4. The definitions are defined inductively based on the step index. As assumed, any world with a 0 index is safe. Note that an indexed world is safe to execute a call instruction if it is safe to execute the instruction together with the function that it calls, and furthermore, the called function, if terminates, should return correctly to the program point immediately following the call instruction.

III. The ISCAP Program Logic and Its Model

In this section, we introduce our new ISCAP program logic and its underlying semantic model. The soundness of ISCAP has been proven in the Coq proof assistant [16]. For this paper, we will directly use the calculus of inductive constructions (CiC) [16] as our meta logic. We will use the following syntax to denote terms and predicates in the meta logic :

$$\begin{array}{lll} (\textit{Term})\ A, B & \triangleq & \textit{Set} \mid \textit{Prop} \mid \textit{Type} \\ & & \mid x \mid \lambda x : A.\ B \mid A\ B \mid A \to B \mid \ldots \\ (\textit{Prop})\ P, Q & \triangleq & \textit{True} \mid \textit{False} \mid \neg P \mid P \wedge Q \mid P \vee Q \\ & & \mid P \to Q \mid \forall x : A.\ P \mid \exists x : A.\ P \mid \ldots \end{array}$$

$$p \begin{cases} r > 0 & \triangleq \lambda \mathbb{C}, \mathbb{S}, i.\ \mathbb{S}(r) > 0 \\ s \mapsto v & \triangleq \lambda \mathbb{C}, \mathbb{S}, i.\ \textit{dom}(\mathbb{S}) = \{s\} \wedge \mathbb{S}(s) = v \\ \llcorner \mathsf{ReqS}_{sc} \lrcorner & \triangleq \lambda \mathbb{C}, \mathbb{S}, i.\ \mathsf{ReqS}_{sc}\ \mathbb{S} \\ p_1 \wedge p_2 & \triangleq \lambda \mathbb{C}, \mathbb{S}, i.\ p_1\ \mathbb{C}\ \mathbb{S}\ i \wedge p_2\ \mathbb{C}\ \mathbb{S}\ i \\ p_1 \vee p_2 & \triangleq \lambda \mathbb{C}, \mathbb{S}, i.\ p_1\ \mathbb{C}\ \mathbb{S}\ i \vee p_2\ \mathbb{C}\ \mathbb{S}\ i \\ p \triangleright g & \triangleq \lambda \mathbb{C}, \mathbb{S}, i.\ \exists \mathbb{S}_0.\ p\ \mathbb{C}\ \mathbb{S}_0\ i \wedge g\ \mathbb{C}\ \mathbb{S}_0\ \mathbb{S}\ i \\ p_1 * p_2 & \triangleq \lambda \mathbb{C}, \mathbb{S}, i.\ \exists \mathbb{S}_1, \mathbb{S}_2. \\ & \qquad \mathbb{S} = \mathbb{S}_1 \uplus \mathbb{S}_2 \wedge p_1\ \mathbb{C}\ \mathbb{S}_1\ i \wedge p_2\ \mathbb{C}\ \mathbb{S}_2\ i \end{cases}$$

$$g \begin{cases} \llcorner \mathsf{NextS}_{sc} \lrcorner & \triangleq \lambda \mathbb{C}, \mathbb{S}, \mathbb{S}', i.\ \mathsf{NextS}_{sc}\ \mathbb{S}\ \mathbb{S}' \\ [p] & \triangleq \lambda \mathbb{C}, \mathbb{S}, \mathbb{S}', i.\ \mathbb{S} = \mathbb{S}' \wedge p\ \mathbb{C}\ \mathbb{S}\ i \\ p_1 \bowtie p_2 & \triangleq \lambda \mathbb{C}, \mathbb{S}, \mathbb{S}', i.\ p_1\ \mathbb{C}\ \mathbb{S}\ i \wedge p_2\ \mathbb{C}\ \mathbb{S}'\ i \\ g_1 \wedge g_2 & \triangleq \lambda \mathbb{C}, \mathbb{S}, \mathbb{S}', i.\ g_1\ \mathbb{C}\ \mathbb{S}\ \mathbb{S}'\ i \wedge g_2\ \mathbb{C}\ \mathbb{S}\ \mathbb{S}'\ i \\ g_1 \vee g_2 & \triangleq \lambda \mathbb{C}, \mathbb{S}, \mathbb{S}', i.\ g_1\ \mathbb{C}\ \mathbb{S}\ \mathbb{S}'\ i \vee g_2\ \mathbb{C}\ \mathbb{S}\ \mathbb{S}'\ i \\ g_1 \circ g_2 & \triangleq \lambda \mathbb{C}, \mathbb{S}, \mathbb{S}', i.\ \exists \mathbb{S}_0.\ g_1\ \mathbb{C}\ \mathbb{S}\ \mathbb{S}_0\ i \wedge g_2\ \mathbb{C}\ \mathbb{S}_0\ \mathbb{S}'\ i \\ g_1 * g_2 & \triangleq \lambda \mathbb{C}, \mathbb{S}, \mathbb{S}', i.\ \exists \mathbb{S}_1, \mathbb{S}_2, \mathbb{S}_1', \mathbb{S}_2'. \\ & \qquad \mathbb{S} = \mathbb{S}_1 \uplus \mathbb{S}_2 \wedge \mathbb{S}' = \mathbb{S}_1' \uplus \mathbb{S}_2' \\ & \qquad \wedge g_1\ \mathbb{C}\ \mathbb{S}_1\ \mathbb{S}_1'\ i \wedge g_2\ \mathbb{C}\ \mathbb{S}_2\ \mathbb{S}_2'\ i \end{cases}$$

$$P \begin{cases} p_1 \Rightarrow p_2 \triangleq \forall \mathbb{C}, \mathbb{S}, i.\ p_1\ \mathbb{C}\ \mathbb{S}\ i \to p_2\ \mathbb{C}\ \mathbb{S}\ i \\ g_1 \Rightarrow g_2 \triangleq \forall \mathbb{C}, \mathbb{S}, \mathbb{S}', i.\ g_1\ \mathbb{C}\ \mathbb{S}\ \mathbb{S}'\ i \to g_2\ \mathbb{C}\ \mathbb{S}\ \mathbb{S}'\ i \end{cases}$$

Fig. 5. Assertions, operators and relations

A. Assertions

We first define the two types of assertions used in our logic: *predicates* and *actions*. A predicate is used to specify a machine state and an action is used to specify the relationship between two machine states. Both kinds of assertions are dependent on the code heap and the step index, which are important for specifying first-class code pointers.

$$\begin{array}{llll} (\textit{Pred}) & p, q & \in & \textit{CHeap} \to \textit{State} \to \textit{Index} \to \textit{Prop} \\ (\textit{Act}) & g & \in & \textit{CHeap} \to \textit{State} \to \textit{State} \to \textit{Index} \to \textit{Prop} \end{array}$$

The first state argument of an action denotes the starting state; the second one its ending state. *Note that in each action, we specify two states but only one index.* This choice is critical to our semantic model. Intuitively, each such action specifies a postcondition, so it does not require any step index for its starting state.

In Fig. 5, we define some special assertions, assertion operators, and assertion relations. The first set of definitions are predicates. The following set of definitions are all actions. The last two are assertion relations. For ease of presentation, we often use r to refer to the value stored in register r in many assertion definitions; for example, $r > 0$ means that the state stores a positive value in register r. We can similarly define other boolean expression predicates. We lift $\wedge$ and $\vee$ from meta logic to assertions. $p \triangleright g$ specifies the ending states of action g starting from states satisfying p, with the same code heap and index. We also define the separation conjunction for predicates and actions. $[p]$ shows an action with the same starting and ending states, which satisfy p. The action $g_1 \circ g_2$ composes two actions into one. The last two relations are implication of assertions.

$\boxed{\Psi \models \{p\}\, sc\, \{g\}}$ ***(Well-Specified Sequential Instructions)***

$$\langle p\rangle \triangleq p * (ra \mapsto \_\,) \qquad \langle g\rangle \triangleq g * (ra \mapsto \_ \bowtie ra \mapsto \_\,) \qquad \langle (p,g)\rangle \triangleq (p * (ra \mapsto \_\,), g * [ra \mapsto \_\,])$$

$$\frac{\Psi(f) = \langle (p,g)\rangle}{\Psi \models \{\langle p\rangle\}\, \texttt{call}\ f\, \{\langle g\rangle\}}\ (\text{CALL}) \qquad \frac{}{\Psi \models \{\langle p\rangle \wedge \mathsf{fptr}(r, \langle (p,g)\rangle)\}\, \texttt{callr}\ r\, \{\langle g\rangle\}}\ (\text{CALLR}) \qquad \frac{sc \notin \{\texttt{call}, \texttt{callr}\}}{\Psi \models \{\llcorner \mathsf{ReqS}_{sc} \lrcorner\}\, sc\, \{\llcorner \mathsf{NextS}_{sc} \lrcorner\}}\ (\text{SC})$$

$$\frac{\Psi \models \{p_1\}\, sc\, \{g_1\} \quad p \Rightarrow p_1 \quad g_1 \Rightarrow g}{\Psi \models \{p\}\, sc\, \{g\}}\ (\text{WEAK-I}) \qquad \frac{\Psi \models \{p\}\, sc\, \{g\}}{\Psi \models \{p * q\}\, sc\, \{g * [q]\}}\ (\text{FRAME-I}) \qquad \frac{\Psi \models \{\mathsf{fptr}(f, \Psi(f)) \wedge p\}\, sc\, \{g\}}{\Psi \models \{p\}\, sc\, \{g\}}\ (\text{FP-I})$$

$\boxed{\Psi \models \{p\}\, \mathbb{I}\, \{g\}}$ ***(Well-Specified Functions)***

$$\frac{\Psi \models \{p\}\, sc\, \{g\} \quad \Psi \models \{p_1\}\, \mathbb{I}\, \{g_1\} \quad p \triangleright g \Rightarrow p_1}{\Psi \models \{p\}\, sc; \mathbb{I}\, \{g \circ g_1\}}\ (\text{SEQ}) \qquad \frac{\Psi \models \{p_1\}\, \mathbb{I}\, \{g_1\} \quad p \Rightarrow p_1 \quad g_1 \Rightarrow g}{\Psi \models \{p\}\, \mathbb{I}\, \{g\}}\ (\text{WEAK-F})$$

$$\frac{\Psi \models \{p\}\, \mathbb{I}\, \{g\} \quad \Psi(f) = (p_1, g_1)}{\Psi \models \{(r \le 0 \wedge p) \vee (r > 0 \wedge p_1)\}\, \texttt{bgtz}\ r, f; \mathbb{I}\, \{g \vee g_1\}}\ (\text{BGTZ}) \qquad \frac{}{\Psi \models \{ra \mapsto \_\,\}\, \texttt{ret}\, \{[ra \mapsto \_\,]\}}\ (\text{RET})$$

$$\frac{\Psi(f) = (p,g)}{\Psi \models \{p\}\, \texttt{jmp}\ f\, \{g\}}\ (\text{JMP}) \qquad \frac{\Psi \models \{\mathsf{fptr}(f, \Psi(f)) \wedge p\}\, \mathbb{I}\, \{g\}}{\Psi \models \{p\}\, \mathbb{I}\, \{g\}}\ (\text{FP-F}) \qquad \frac{\Psi \models \{p\}\, \mathbb{I}\, \{g\}}{\Psi \models \{p * q\}\, \mathbb{I}\, \{g * [q]\}}\ (\text{FRAME-F})$$

$\boxed{\Psi \models \mathbb{C} : \Psi_1}$ ***(Well-Specified Code Heaps)***

$$\frac{\Psi \models \{p\}\, \mathtt{cb}(\mathbb{C}, f)\, \{g\}}{\Psi \models \mathbb{C} : \{f \rightsquigarrow (p,g)\}}\ (\text{SGLTON}) \qquad \frac{\Psi_1 \models \mathbb{C}_1 : \Psi_3 \quad \Psi_2 \models \mathbb{C}_2 : \Psi_4}{\Psi_1 \oplus \Psi_2 \models \mathbb{C}_1 \oplus \mathbb{C}_2 : \Psi_3 \oplus \Psi_4}\ (\text{LINK})$$

$\boxed{\models \{p\}\, (\mathbb{C}, f)\, \{g\}}$ ***(Well-Specified Program)***

$$\frac{\Psi \models \mathbb{C} : \Psi \quad \Psi \models \{p\}\, \mathtt{cb}(\mathbb{C}, f)\, \{g\}}{\models \{p\}\, (\mathbb{C}, f)\, \{g\}}\ (\text{PROG})$$

Fig. 6. ISCAP Inference rules

B. Assertion Pairs as Specifications

We use a predicate as the pre-condition of a program point and an action to specify how a program will behave. If p can guarantee the safety of a sequential instruction sc and the behavior of sc satisfies g, we say (p, g) covers the sequential instruction sc.

Definition 1 *The formal definition of* (p, g) *covers sc under the code heap* $\mathbb{C}$ *for* i *steps* $((p, g) \propto_{(\mathbb{C},i)} sc)$*:*

$$\forall j < i.\ \forall \mathbb{S}, pc.\ p\ \mathbb{C}\ \mathbb{S}\ j \wedge \mathbb{C}(pc) = sc \rightarrow \mathit{ISafe}((\mathbb{C}, \mathbb{S}, j), pc)$$
$$\wedge (\forall \mathbb{S}', j', pc'.\ ((\mathbb{C}, \mathbb{S}, j), pc) \downarrow ((\mathbb{C}, \mathbb{S}', j'), pc') \rightarrow g\ \mathbb{C}\ \mathbb{S}\ \mathbb{S}'\ j')$$

For example, $(\llcorner \mathsf{ReqS}_{sc} \lrcorner, \llcorner \mathsf{NextS}_{sc} \lrcorner)$ covers the instruction sc if it is not a function call.

We want to define that an assertion pair covers a function call in the same way. However, we cannot define structurally a function from the unstructured assembly programs, so we use structure $\mathbb{I}$ to denote the function body from block $\mathbb{I}$ to the end(s) of the current function. This is different from traditional systems, which use code blocks $\mathbb{I}$ to stand for the whole continuation starting from block $\mathbb{I}$. From now on, when we say an assertion pair covers a code block, we really mean it covers its underlying function.

Definition 2 *The formal definition of* (p, g) *covers* $\mathbb{I}$ *under the code heap* $\mathbb{C}$ *for* i *steps* $((p, g) \propto_{(\mathbb{C},i)} \mathbb{I})$*:*

$$\forall j < i.\ \forall \mathbb{S}, pc.\ p\ \mathbb{C}\ \mathbb{S}\ j \wedge \mathtt{cb}(\mathbb{C}, pc) = \mathbb{I} \rightarrow \mathit{FSafe}((\mathbb{C}, \mathbb{S}, j), pc)$$
$$\wedge (\forall \mathbb{S}', j', pc'.\ ((\mathbb{C}, \mathbb{S}, j), pc) \Downarrow ((\mathbb{C}, \mathbb{S}', j'), pc') \rightarrow g\ \mathbb{C}\ \mathbb{S}\ \mathbb{S}'\ j')$$

C. Function Pointer Assertions and Well-Formed Assertions

In our machine model, function pointers may be stored in any part of a state. We use the following assertion to specify a first-class function pointer:

$$\mathsf{fptr}(f, (p, g)) \triangleq \lambda \mathbb{C}, \mathbb{S}, i.\ (p, g) \propto_{(\mathbb{C},i)} \mathtt{cb}(\mathbb{C}, f)$$
$$\mathsf{fptr}(r, (p, g)) \triangleq \lambda \mathbb{C}, \mathbb{S}, i.\ (p, g) \propto_{(\mathbb{C},i)} \mathtt{cb}(\mathbb{C}, \mathbb{S}(r))$$

It means that (p, g) can cover the function pointed by f (or r of state $\mathbb{S}$) for i steps under code heap $\mathbb{C}$. Only well-formed assertions (defined below) are allowed in our program logic.

Definition 3 *Well-formed assertions are monotonic on the index:*

$$\mathtt{wf}(p) \triangleq \forall \mathbb{C}, \mathbb{S}, i \le j.\ p\ \mathbb{C}\ \mathbb{S}\ j \rightarrow p\ \mathbb{C}\ \mathbb{S}\ i$$
$$\mathtt{wf}(g) \triangleq \forall \mathbb{C}, \mathbb{S}, \mathbb{S}', i \le j.\ g\ \mathbb{C}\ \mathbb{S}\ \mathbb{S}'\ j \rightarrow g\ \mathbb{C}\ \mathbb{S}\ \mathbb{S}'\ i$$

Ill-formed assertions like $\lambda \mathbb{C}, \mathbb{S}, i.\ i > 3$, are meaningless for reasoning about the safety properties of programs.

Lemma 1 *About well-formed assertions:*

- *those special assertions defined in Fig. 5 are well-formed;*
- *function pointer assertions are well-formed;*
- *well-formedness is preserved by the operators defined in Fig. 5.*

D. ISCAP Inference Rules

Fig. 6 shows the inference rules of our new program logic ISCAP. We have four sets of rules for four kinds of rule judgments. Ψ is the specification for code heap. It is a partial mapping from code pointers to assertion pairs.

$$(\mathit{CHSpec}) \quad \Psi \quad \in \quad \mathit{Label} \rightharpoonup_{\text{fin}} \mathit{Pred} \times \mathit{Act}$$

We use code heap specifications to cover code heaps. A code heap specification Ψ covers a code heap $\mathbb{C}$ for i steps under another code heap $\mathbb{C}_0$ (denoted as $\Psi \propto_{(\mathbb{C}_0,i)} \mathbb{C}$) if and only if for any code pointer f in domain of Ψ, $\Psi(f)$ covers $\mathtt{cb}(\mathbb{C}, f)$ for i steps under the code heap $\mathbb{C}_0$.

The first set contains rules for sequential instructions. Judgement $\Psi \models \{p\}\, sc\, \{g\}$ means that if for any code heap $\mathbb{C}$ and index i, Ψ covers $\mathbb{C}$ for i steps under $\mathbb{C}$, then (p, g) covers sc for $i + 1$ steps under $\mathbb{C}$.

$$\forall \mathbb{C}, i.\ \Psi \propto_{(\mathbb{C},i)} \mathbb{C} \rightarrow (p, g) \propto_{(\mathbb{C},i+1)} sc$$

Rule CALL is for direct function calls and Rule CALLR is for indirect function calls. Rule SC is for other sequential instructions. We also have weakening rule WEAK-I and ordinary frame rule FRAME-I for well-specified sequential instructions. FP-I is used to introduce function pointer assertions when we know the exact values of the function pointers.

The following set contains rules for well-specified functions. Judgement $\Psi \models \{p\}\, \mathbb{I}\, \{g\}$ means that if for any code heap $\mathbb{C}$ and index i, Ψ covers $\mathbb{C}$ for i steps under $\mathbb{C}$, then (p, g) covers $\mathbb{I}$ for $i + 1$ steps under $\mathbb{C}$.

$$\forall \mathbb{C}, i.\ \Psi \propto_{(\mathbb{C},i)} \mathbb{C} \rightarrow (p, g) \propto_{(\mathbb{C},i+1)} \mathbb{I}$$

Rules SEQ, BGTZ, JMP and RET follow the syntax of the code blocks. Because an assertion only covers a program to the return point (of the current function), the pre-condition for the return instruction only need to guarantee that the return address register is in domain of the state, ignoring the future execution after the function return. We also have weakening rule WEAK-F, ordinary frame rule FRAME-F and introduction rule of function pointer assertion FP-F.

The third set contains rules for well-specified code heaps. Judgment $\Psi \models \mathbb{C} : \Psi_0$ means that for any global code heap $\mathbb{C}_0$ and index i, if Ψ covers $\mathbb{C}_0$ for i steps under $\mathbb{C}_0$, then Ψ_0 covers the local code heap $\mathbb{C}$ for $i+1$ steps under $\mathbb{C}_0$. Global code heap contains all the function code which the local code heap may call.

$$\forall \mathbb{C}_0, i.\ \Psi \propto_{(\mathbb{C}_0,i)} \mathbb{C}_0 \rightarrow \Psi \propto_{(\mathbb{C}_0,i+1)} \mathbb{C}$$

Rule SGLTON is used for generating the specification for the singleton local code heap. Rule LINK is for linking two local code heaps. $\oplus$ is the harmonious merge operator for partial mappings. The result partial mapping is only defined when the two small mappings do not conflict with each other. $\mathbb{C}_1 \oplus \mathbb{C}_2$ is formally defined as:

$$\lambda f.\ \begin{cases} c, & \text{if } \mathbb{C}_1(f) = \mathbb{C}_2(f) = c \\ \mathbb{C}_1(f), & \text{if } f \in \mathit{dom}(\mathbb{C}_1) \wedge f \notin \mathit{dom}(\mathbb{C}_2) \\ \mathbb{C}_2(f), & \text{if } f \notin \mathit{dom}(\mathbb{C}_1) \wedge f \in \mathit{dom}(\mathbb{C}_2) \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

$\Psi_1 \oplus \Psi_2$ is defined similarly.

We can show that when the local code heap specification is equal to the global code heap specification, we get a global code heap specification which can cover the global code heap for any steps. This is the key lemma of the whole system. It shows how the indexed semantics works and how the code pointer circularity is broken. The proof is omitted here but can be found in our Coq implementation [12].

Lemma 2 *If $\Psi \models \mathbb{C} : \Psi$, then $\forall i.\ \Psi \propto_{(\mathbb{C},i)} \mathbb{C}$. (With the implicit assumption that all assertions in Ψ are well-formed)*

The last rule is the top rule. Judgement $\models \{p\}\, (\mathbb{C}, f)\, \{g\}$ is the final goal we want to prove. It means that (p, g) can cover function pointed by f for any steps under code heap $\mathbb{C}$.

$$\forall i.\ (p, g) \propto_{(\mathbb{C},i)} \mathtt{cb}(\mathbb{C}, f)$$

Theorem 1 (Soundness) *All rules in Fig. 6 are valid based on their semantic definitions.*

Now let's show what we get by the top rule with a corollary.

Corollary 1 (Partial Correctness) *If $\models \{p\}\, (\mathbb{C}, f)\, \{g\}$, then for any state $\mathbb{S}$ satisfying $\forall i.\ p\ \mathbb{C}\ \mathbb{S}\ i$, it is safe to execute the function pointed by f and when the function returns, the ending state $\mathbb{S}'$ should satisfy $\forall i.\ g\ \mathbb{C}\ \mathbb{S}\ \mathbb{S}'\ i$.*

It is easy to see that this property is the traditional partial correctness based on functions. It is stronger than the simple safety properties proven in the existing logic systems for certification of assembly programs [1]–[5].

IV. An Example

Fig. 7 shows a function testing whether virtual functions in C++ are called correctly. Its assembly implementation is in Fig. 8. We use identifiers to denote different code labels for clarity. Each object of a class contains a series of memory cells. The first cell contains the starting address of its virtual function list. The rest of the memory cells contain its member variables. All member functions of an object have an implicit argument which points to the object itself (which is the starting address of its series of memory cells). The function `pcirc` is the virtual function `circ` of the class `Polygon` and `scirc` of the class `Square`. The constructor for class `Polygon` is `pcons` which stores the virtual function pointer `pcirc` in the first memory cell of the new object. The constructor for class `Square` is `scons` which first calls the constructor of its parent class. Then `scons` stores the virtual function pointer `scirc` in the first memory cell of the new object and initializes the variable member `e` which is stored in the second memory cell. In the `test` function, it creates an object of class `Square` and initializes its edge by its argument. Then it calls the `circ` function of that object by first finding the virtual list and then getting the corresponding function pointer from the virtual list. Since the constructor function and the `circ` function for class `Triangle` is not used here, we do not list them in the figure.

```
class Polygon{
public:
  virtual int circ() {return 0;}
};

class Triangle : public Polygon {
  int e1,e2,e3;
public:
  Triangle(int a1, int a2, int a3)
    {e1 = a1; e2 = a2; e3 = a3;}
  int circ() {return (e1 + e2 + e3);}
};

class Square : public Polygon {
  int e;
public:
  Square(int a) {e = a;}
  int circ() {return (4 * e);}
};

int test(int x){
  Polygon *p = new Square(x);
  return (p->circ());
}
```

Fig. 7. Virtual functions in C++

The specifications are given in Fig. 9. Note that in order to show how the function pointer is used, we also show the assertion pair (p_4, g_4) at the point where the `circ` function is called in the function `test`. (p_4, g_4) covers the function body with three instructions: an indirect call to the "circ", the move instruction and the return instruction, which can be proven easily following the rules given in Fig. 6. The predicate p_4 contains a function pointer assertion since the actual value of register t_0 is unknown. The pre-condition p for the whole `test` function does not contain any function pointer assertion, since the exact pointer value `scirc` is known from the post condition of function `scons`. From p to p_4, the function pointer assertion is introduced using the rule FP-F.

We can prove the following theorem.

Theorem 2 *The function* `test` *is well-specified by* (p, g)*:*

$$\models \{p\}\,(\mathbb{C}, \mathit{test})\,\{g\}$$

in which p, g and $\mathbb{C}$ *are shown in Fig. 8 and Fig. 9.*

Note that the code heap and the index are not referred in p or g, which means the traditional partial correctness of function `test` is proven here using our program logic.

V. Related Work and Conclusions

The original SCAP program logic [8] was designed to support modular verification of assembly programs with stack-based control abstractions. Unlike the CPS-based XCAP logic [5], SCAP does not treat return address as first-class continuation pointers. Instead, it maintains an implicit stack invariant, ensuring that each enclosing function has a safe return point. SCAP does not support first-class function pointers. Combining SCAP and XCAP would lead to a program logic that supports embedded code pointers [17], but it is unclear

```
pcirc :   −{(p0, g0)}         ; circ() of class Polygon
          ret
pcons :   −{(p1, g1)}         ; constructor of Polygon
          movi t0, pvlist     ; virtual list of Polygon
          sw 0(a0), t0
          ret
scirc :   −{(p2, g2)}         ; circ() of class Square
          lw rv, 1(a0)        ; load length of edge e
          multi rv, rv, 4     ; 4 × e
          ret
scons :   −{(p3, g3)}         ; constructor of Square
          mov s1, ra          ; save return address
          call pcons          ; call pcons first
          movi t0, svlist     ; virtual list of Square
          sw 0(a0), t0
          sw 1(a0), a1        ; assign length of edge e
          mov ra, s1          ;resume return address
          ret
test :    −{(p, g)}
          mov s0, ra          ; save return address
          movi a1, a0         ; initial edge length
          alloc a0, 2         ; memory for new object
          call scons          ; create Square object
          lw t0, 0(a0)        ; load the address of vlist
          lw t0, 0(t0)        ; dynamically address circ()
          −{(p4, g4)}
          callr t0            ; call circ()
          mov ra, s0          ; resume return address
          ret
pvlist :  pcirc               ; vlist of Polygon
svlist :  scirc               ; vlist of Square
```

Fig. 8. Assembly implementation for the example

$$
\begin{array}{lll}
p_0 & \triangleq & \mathit{ra} \mapsto \_ \\
g_0 & \triangleq & [\mathit{ra} \mapsto \_\,] \\
p_1 & \triangleq & \exists\, l.\ \mathit{ra} \mapsto \_ \ * t_0 \mapsto \_ \ * (a_0 \mapsto l) * (l \mapsto \_\,) \\
g_1 & \triangleq & \exists\, l.\ [(a_0 \mapsto l) * (\mathit{ra} \mapsto \_\,)] \\
 & & \quad *((l \mapsto \_ \ * t_0 \mapsto \_\,) \bowtie (l \mapsto \texttt{pvlist} * t_0 \mapsto \_\,)) \\
p_2 & \triangleq & \exists\, l.\ \mathit{ra} \mapsto \_ \ * \mathit{rv} \mapsto \_ \ * a_0 \mapsto l * (l+1 \mapsto \_\,) \\
g_2 & \triangleq & \exists\, l, v.\ [\mathit{ra} \mapsto \_ \ * a_0 \mapsto l * (l+1 \mapsto v)] \\
 & & \quad *(\mathit{rv} \mapsto \_ \ \bowtie (\mathit{rv} \mapsto 4 \times v)) \\
p_3 & \triangleq & \exists\, l.\ a_1 \mapsto \_ \ * \mathit{ra} \mapsto \_ \ * t_0 \mapsto \_ \ * s_1 \mapsto \_ \ * a_0 \mapsto l \\
 & & \quad * l \mapsto \_ \ * (l+1 \mapsto \_\,) \\
g_3 & \triangleq & \exists\, l, v.\ [\mathit{ra} \mapsto \_ \ * a_0 \mapsto l * a_1 \mapsto v] \\
 & & \quad *(l \mapsto \_ \ * (l+1 \mapsto \_\,) \bowtie l \mapsto \texttt{svlist} * (l+1 \mapsto v)) \\
 & & \quad *(t_0 \mapsto \_ \ * s_1 \mapsto \_ \ \bowtie t_0 \mapsto \_ \ * s_1 \mapsto \_\,) \\
p_4 & \triangleq & \exists\, l.\ \mathit{rv} \mapsto \_ \ * \mathit{ra} \mapsto \_ \ * t_0 \mapsto \_ \ * a_0 \mapsto l \\
 & & \quad *(l+1 \mapsto \_\,) \wedge \mathsf{fptr}(t_0, (p_2, g_2)) \\
g_4 & \triangleq & \exists\, l, v, f.\ [a_0 \mapsto l * s_0 \mapsto f * (l+1 \mapsto v)] \\
 & & \quad *(\mathit{ra} \mapsto \_ \ * \mathit{rv} \mapsto \_ \ \bowtie \mathit{ra} \mapsto f * (\mathit{rv} \mapsto 4 \times v)) \\
p & \triangleq & \mathit{ra} \mapsto \_ \ * t_0 \mapsto \_ \ * a_0 \mapsto \_ \ * a_1 \mapsto \_ \ * \mathit{rv} \mapsto \_ \ * s_0 \mapsto \_ \\
 & & \quad * s_1 \mapsto \_ \ * \texttt{pvlist} \mapsto \texttt{pcirc} * \texttt{svlist} \mapsto \texttt{scirc} \\
g & \triangleq & \exists\, v.\ [\mathit{ra} \mapsto \_ \ * \texttt{pvlist} \mapsto \texttt{pcirc} * \texttt{svlist} \mapsto \texttt{scirc}] \\
 & & \quad *(t_0 \mapsto \_ \ * a_1 \mapsto \_ \ \bowtie t_0 \mapsto \_ \ * a_1 \mapsto \_\,) \\
 & & \quad *(s_0 \mapsto \_ \ * s_1 \mapsto \_ \ \bowtie s_0 \mapsto \_ \ * s_1 \mapsto \_\,) \\
 & & \quad *(a_0 \mapsto v * \mathit{rv} \mapsto \_ \ \bowtie a_0 \mapsto \_ \ * (\mathit{rv} \mapsto 4 \times v))
\end{array}
$$

$$
\begin{array}{ll}
\Psi : & \{\texttt{pcirc} \rightsquigarrow (p_0, g_0), \texttt{pcons} \rightsquigarrow (p_1, g_1), \\
 & \ \texttt{scirc} \rightsquigarrow (p_2, g_2), \texttt{scons} \rightsquigarrow (p_3, g_3)\}
\end{array}
$$

Fig. 9. The specifications for the example

whether it can prove the same partial correctness properties as we have done using ISCAP.

Appel *et al.* [1], [3], [15] pioneered the step-indexing technique and have shown how to use it to build semantic models for a rich type system with ML-style polymorphism, high-order functions, recursive types, and references. They use the index not just to count execution steps, but also to construct a dependently typed global heap type (for references). Our use of step-indexing is similar to theirs, except that their work is for building semantic models for types, while we use indices to modify the original definition of Hoare triples to prove the partial correctness property for programs with higher-order features. Appel *et al.* also built a Separation Logic for CMinor [18] using an index-based semantic model; it can support nested Hoare triples like our fptr assertion language construct. However, their model is based on a higher-level language and their soundness theorem only guarantees the safety property, not the same partial correctness property we are proving in this paper.

Reus *et al.* [11], [19], [20] presented an extension of separation logic with support for high-order store and nested Hoare triples. They can also prove the similar partial correctness properties as we do but the soundness of their program logics requires a more complicated semantic model based on solving recursive domain equations. Our semantic model, in contrast, requires elementary construction only and it is based on an intuitive operational semantics which we believe is more suitable for low-level assembly languages.

Honda *et al.* proposed a compositional logic for a high-level functional language [21]. They used auxiliary variables called anchors in judgments to represent the eventual values of specified terms and showed some non-trivial examples. However, anchors are special for functional languages and can not be applied to assembly languages.

Previous program logics for assembly code verification follow CPS-based reasoning and cannot be used to certify the same partial correctness property as done in Hoare-style logics. In this paper, we have proposed a new program logic named ISCAP for certifying the partial correctness properties of assembly programs with first-class function pointers. We have proved the soundness of ISCAP by building a simple semantic model based on a direct-style, mixed-step operational semantics for the assembly language. In the future, we plan to extend our semantic model to support other stack-based control structures [8], concurrency, recursive data structures, references, and higher-order frame rules.

Acknowledgment

We thank Alex Vaynberg, Xinyu Feng, and anonymous referees for their suggestions and comments on an earlier version of this paper. This research is based on work supported in part by the National Natural Science Foundation of China under Grants No. 60928004, 61073040, and 61003043 and Natural Science Foundation of Jiangsu Province, China (No. BK2008181). Part of Wei Wang's work is done during his visit to Yale University in 2008-2010 and is also supported in part by China Scholarship Council and by National Science Foundation (of USA) under Grant CCR-0915888. Zhong Shao is supported in part by DARPA CRASH grant FA8750-10-2-0254 and NSF grants CNS-0915888, CNS-0910670, and CCF-0811665. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] A. W. Appel and D. McAllester, "An indexed model of recursive types for foundational proof-carrying code," *ACM Transactions on Programming Languages and Systems*, pp. 657–683, Sep. 2001.

[2] G. Tan, A. W. Appel, K. N. Swadi, and D. Wu, "Construction of a semantic model for a typed assembly language," in *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2004.

[3] A. W. Appel, P. Mellies, C. D. Richards, and J. Vouillon, "A very modal model of a modern, major, general type system," in *Proc. 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL07)*, Jan. 2007, pp. 109–122.

[4] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From system F to typed assembly language," *ACM Trans. on Programming Languages and Systems*, 1999.

[5] Z. Ni and Z. Shao, "Certified assembly programming with embedded code pointers," in *Proc. 33rd ACM Symp. on Principles of Prog. Lang.*, Jan. 2006, pp. 320–333.

[6] A. W. Appel, "Compiling with continuations," *Cambridge University Press*, 1992.

[7] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, pp. 576–580, Oct. 1969.

[8] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni, "Modular verification of assembly code with stack-based control abstractions," in *Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI'06)*, Ottawa Canada, Jun. 2006, pp. 401–414.

[9] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proc. 17th IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 2002, pp. 55–74.

[10] P. America and J. Rutten, "Solving reflexive domain equations in a category of complete metric spaces," in *J. Comput. Syst. Sci.*, 1989, pp. 343–375.

[11] J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang, "Nested Hoare triples and frame rules for higher-order store," in *Conference on Computer Science Logic*, 2009, pp. 440–454.

[12] W. Wang, "Coq implementation and soundness proof." [Online]. Available: http://kyhcs.ustcsz.edu.cn/~wwang/pgiscap_soundness.tar

[13] D. Yu, N. A. Hamid, and Z. Shao, "Building certified libraries for PCC: Dynamic storage allocation," in *Proc. 2003 European Symposium on Programming (ESOP'03)*, Warsaw Poland, Apr. 2003, pp. 402–416.

[14] M. J. Parkinson, R. Bornat, and C. Calcagno, "Variables as resource in Hoare logics," in *Proc. 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, Aug. 2006, pp. 137–146.

[15] A. J. Ahmed, "Semantics of types for mutable state," Ph.D. dissertation, Princeton University, 2004.

[16] The Coq Development Team, *The Coq Proof Assistant Reference Manual*, Oct. 2004-2006.

[17] X. Feng, Z. Ni, Z. Shao, and Y. Guo, "An open framework for foundational proof-carrying code," in *Proc. 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, Jan. 2007, pp. 67–78.

[18] A. W. Appel and S. Blazy, "Separation logic for small-step Cminor," in *Theorem Proving in Higher Order Logics, 20TH INT. CONF. TPHOLS 2007*, vol. 4732 of LNCS. Springer, 2007, pp. 5–21.

[19] B. Reus and J. Schwinghammer, "Separation logic for higher-order store," in *CSL*, 2006, pp. 575–590.

[20] L. Birkedal, B. Reus, J. Schwinghammer, and H. Yang, "A simple model of separation logic for higher-order store," in *ICALP (2)*, 2008, pp. 348–360.

[21] K. Honda, N. Yoshida, and M. Berger, "An observationally complete program logic for imperative higher-order frame rules," in *Proc. 20st Annual IEEE Symposium on Logic in Computer Science*, 2005, pp. 270–279.