Local Reasoning and Information Hiding in SCAP

Xinyu Feng\(^1\) and Zhong Shao\(^2\)

\(^1\) Toyota Technological Institute at Chicago \texttt{feng@tti-c.org}
\(^2\) Yale University \texttt{shao@cs.yale.edu}

Abstract. Separation logic supports state-modular program verification in two aspects: local reasoning by means of the ordinary (first-order) frame rule and information hiding by the hypothetical frame rule (a second-order frame rule). In this paper, we show the support of local reasoning and information hiding in SCAP, a program logic for low-level programs with stack-based control abstractions. We show that, using intuitionistic assertions as preconditions, and using a binary relation over program states to specify program behaviors, SCAP naturally supports local reasoning without requiring a frame rule. Then we show the hypothetical frame rule can be proved as an admissible rule in SCAP, given some natural constraints over SCAP specifications.

1 Introduction

Modularity is very important for scalable program specification, verification, and proof reuse. Recent work on Separation Logic [5, 11, 10] has demonstrated the importance of two kinds of modularity.

- **Local reasoning**: specifications of a module should be independent of the specific context in which it is used, so that the specification and the verified module can be reused in different contexts.
- **Information hiding**: internal data of program modules should be hidden to their clients, so that specification and verification of client code can be much simpler.

Separation logic supports local reasoning by means of the ordinary (first-order) frame rule [5, 11]:

\[
\Gamma \vdash \{p\} C \{q\} \\
\Gamma \vdash \{p \ast r\} C \{q \ast r\}
\]

It says that any property \(r\) over the sub-heap untouched by \(C\) will be preserved at the end of \(C\). Therefore, when we verify \(C\), we can use the small footprint specification \(\{p\}\_\{q\}\), which only specifies the part of the heap accessed in \(C\). When \(C\) is used in different contexts, we can apply the frame rule with different \(r\).

Information hiding is achieved by applying the hypothetical frame rule (a second order frame rule) by O’Hearn, Yang and Reynolds [10]:

\[
\Gamma, \{p_i\} k_i \{q_i\} \text{for } i \leq n \vdash \{p\} C \{q\} \\
\Gamma, \{p_i \ast r\} k_i \{q_i \ast r\} \text{for } i \leq n \vdash \{p \ast r\} C \{q \ast r\}
\]
Here functions $k_i (i \leq n)$ are called by $C$. Each of them uses the sub-heap satisfying $r$ and preserves its well-formedness with respect to $r$. When we verify $C$, if $C$ never directly accesses the sub-heap, we can hide its existence from $C$’s specifications, and pretend that functions $k_i$ do not use $r$ either.

In this paper, we show how these two kinds of modularity can be supported in the SCAP logic [4], a program logic for assembly-level programs. At assembly level, implementations of stack-based control abstractions, such as function calls, exceptions, setjmp/longjmp, and coroutines and threads context switching, all involve indirect jumps. SCAP supports modular verification of stack-based controls without treating all control flows as first-class code pointers following the continuation-passing style [7, 9]. SCAP has been applied to verify low-level libraries and runtime code such as malloc/free [13], garbage collectors [6] and thread context switchings [3].

This paper is based on previous development on SCAP, but makes the following contributions:

– We point out that, using intuitionistic specifications, SCAP can naturally support local reasoning for functions without using the frame rule. Although this fact has been implicitly applied in previous practice using SCAP [13, 6, 3], it has never been related to the local-reasoning supported by Separation Logic.

– We show that, with some natural constraints over SCAP specifications, a similar hypothetical frame rule can be proved as an admissible rule of SCAP, so that information hiding can be supported.

It is also worth noting that we present SCAP in this paper as a stand-alone logic, whose soundness is proved based on the syntactic approach, while previous work [4] presented SCAP as a set of admissible rules in a foundational framework and SCAP’s soundness is delegated to the soundness of the underlying framework.

In the rest of this paper, we first define the machine and the assembly language in Sec. 2 and give an overview of SCAP in Sec. 3. We then show by an example how local reasoning can be achieved in SCAP in Sec. 4. In Sec. 5 we show the hypothetical frame rule can be proved admissible in SCAP with some constraint over SCAP specifications. We compare this work with related work and conclude in Sec. 6.

2 The Machine

In Fig. 1 we show the definition of a MIPS-style target machine (TM). A machine state is called a “Program” ($P$), which consists of a read-only code heap ($C$), an updatable state ($S$), and a program counter (pc). The code heap is a finite partial mapping from code labels to commands. A command $c$ is either a sequential command $\overline{c}$ or a direct or indirect jump command. The sequential command can be arithmetic commands, memory load ($lw$) and store ($sw$), jump and link ($jal$), or conditional branch ($bgtz$) (the last two are not really sequential commands). The state $S$ contains a data heap ($H$) and a register file ($R$). $H$ is a finite partial mapping from memory locations (natural numbers) to word values (integers). $R$ is a total function from registers to word values.
(Prog) \( P ::= (C, S, pc) \)

(CodeHeap) \( C ::= \{ f \sim c \} \)

(State) \( S ::= (H, R) \)

(Heap) \( H ::= \{ l \sim w \} \)

(Register) \( r ::= r_0 | \ldots | r_{31} \)

(Labels) \( f, l ::= \text{n}_n \) (Integers)

(Word) \( w ::= \text{i}_n \)

(SeqCmd) \( \bar{c} ::= \text{addu} r_d, r_s, r_t | \text{addiu} r_d, r_s, w | \text{bgtz} r_s, f | \text{jal} f \\
| \text{lw} r_t, w(r_s) | \text{sw} r_t, w(r_s) \)

(Command) \( c ::= \bar{c} | j f | jr r_s \)

(BasicBlock) \( I ::= \bar{c}; I \mid j f \mid jr r_s \)

Fig. 1. Syntax of Target Machine TM

We define a basic block \( I \) as a list of sequential instructions with a jump instruction at the end. \( C[f] \) extracts a basic block starting from \( f \) in \( C \):

\[
C[f] \triangleq \begin{cases} 
  c & \text{if } c = C(f) \text{ and } c = j f' \text{ or } jr r_s \\
  \bar{c}; I & \text{if } \bar{c} = C(f) \text{ and } I = C[f+1] \\
  \text{undefined} & \text{otherwise}
\end{cases}
\]

The execution of TM programs is modeled as a small-step transition from one program to another, i.e., \( P \xhookrightarrow{*} P' \), as shown in Figure 2. We use the binary relation \( \text{NextS}(pc, c) \) and \( \text{NextPC}(c, S) \) to formalize the effects of command \( c \) over program states and \( pc \)'s. The \( k \)-step transition and the reflexive transitive closure of the transition relation are defined as \( P \xhookrightarrow{*}^k P' \) and \( P \xhookrightarrow{*} P' \), respectively.

3 An Overview of SCAP

In this section, we give an overview of SCAP [4], a Hoare-style program logic for assembly programs. SCAP was originally presented as a set of admissible rules of a foundational verification framework. Here we present it as a stand-alone logic.

Specifications. Instead of defining the syntax of the assertion language, SCAP reuses a mechanized meta-logic (e.g., Coq [12]) as its assertion language. As shown below, a predicate over program states in the meta-logic has the meta-type \( State \rightarrow \text{Prop} \), i.e., a function that takes a state as argument and returns a meta-logical proposition.

\[
\begin{align*}
  (\text{StPred}) \ p & \in \text{State} \rightarrow \text{Prop} \\
  (\text{Spec}) \ \theta & \in \text{State} \rightarrow \text{State} \rightarrow \text{Prop} \\
  (\text{Guarantee}) \ g & \in \text{State} \rightarrow \text{State} \rightarrow \text{Prop} \\
  (\text{CdHpSpec}) \ \Psi & \in \mathcal{P}(\text{Labels} \times \text{Spec}) \\
  (\text{MemPred}) \ m & \in \text{Heap} \rightarrow \text{Prop}
\end{align*}
\]

We use a pair of assertions \( (p, g) \) as a specification \( \theta \) for a basic block. The guarantee \( g \) is a binary relation over states. It specifies the state transition from the specified
The following logical equivalence are true:

(1) \((p \circ g) \circ g' \iff p \circ (g \circ g')\)  
(2) \((p \circ g) \circ g' \iff p \circ (g \circ g')\)  
(3) \((g \circ g') \circ g'' \iff g \circ (g' \circ g'')\)
The inference rules are shown in Figure 4. The judgment $\Psi \vdash \{(p, g)\} f : I$ says that it is safe to execute $I$ if the current state satisfies $p$, and that the transition from the current state to the end of the current function satisfies $g$. $\Psi$ specifies code that might be reached from $I$ via jump, branch or jump-and-link commands. In other words, $\Psi$ specifies imported interfaces for $I$.

If $I$ starts with normal arithmetic instructions, the pure rule requires us to find an intermediate specification $(p', g')$ such that the remaining part of $I$ is well-formed. $(p', g')$ is also used as the post condition of the first command. If the current state satisfies $p$, the state after the transition $NextS(f, c)$ needs to satisfy $p'$. Also, the composition of $NextS(f, c)$ and the remaining behavior $g'$ must fulfill the guarantee $g$. Recall that $NextS(f, c)$ is a binary relation over states, thus an instance of guarantee $g$. The mem rule for load and store commands is similar, with additional requirement that the target address must be in the domain of the heap.

For function call, as shown in the call rule, we first look up the specification $(p', g')$ of the callee from $\Psi$, and find an intermediate specification $(p'', g'')$ for the remaining part of $I$. The precondition $p''$ of the callee must hold after the transition $g_{call}$ made by jal. The next two premises are similar to those in the pure rule, viewing $g_{call} \circ g'$ as an abstract command. Finally, $g'$ must guarantee that $\$ra$ (the alias for the register $r_{al}$) at the end of the callee function contains the same value as the return address passed to the callee at the beginning. This requirement can be relaxed to support more general control flows [4], but the extension is orthogonal with the support of frame rules.

Function return is an indirect jump to the return address saved in $\$ra$. The ret rule simply requires that an identity transition would fulfill the remaining guarantee. The rules for direct jump and branch commands are also simple and not explained here.

Code Heap Rules and Linking. The judgment $\Psi \vdash C : \Psi'$ means that the code heap is well-formed with respect to the imported interface $\Psi$ and exported interface $\Psi'$. The cdhp rule says $C$ is well-formed if, for every exported specification $(f, \theta)$, the corresponding basic block $C[f]$ is well-formed. The link rule says that we can separate a code heap into sub-heaps and certify them separately. We use $f \perp g$ to mean $f$ and $g$ have disjoint domains, i.e., $dom(f) \cap dom(g) = \emptyset$.

Program Invariants The prog rule defines the well-formedness of the whole program $(C, I, pc)$. It requires that $C$ is well-formed, and the imported interface $\Psi$ is a subset of the exported interface $\Psi'$, which means every imported interface has been implemented in $C$ and the implementation has been certified with respect to this interface.
\[\Psi \vdash \emptyset : \text{I} \] (Well-Formed Instr. Seq.)

\[\forall c \in \{\text{addu, addiu, subu}\} \quad \Psi \vdash \{(p', g')\} f + 1: \text{I} \quad \text{p} \triangleright \text{NextS}(f, c) \Rightarrow p' \quad \text{p} \circ \text{NextS}(f, c) \circ g' \Rightarrow g\]

(PURE)

\[c = \text{lw} r_d, w(r_s) \text{ or sw} r_d, w(r_s) \quad \forall S, p S \rightarrow (S, R) + w \in S.H \]

\[\Psi \vdash \{(p', g')\} f + 1: \text{I} \quad \text{p} \triangleright \text{NextS}(f, c) \Rightarrow p' \quad \text{p} \circ \text{NextS}(f, c) \circ g' \Rightarrow g\]

(MEM)

\[(f', (p', g')) \in \Psi \quad \Psi \vdash \{(p'', g'')\} f + 1: \text{I} \quad \text{p} \triangleright \text{gcall} \Rightarrow p' \quad \text{p} \circ \text{gcall} \circ g' \Rightarrow g'' \quad \text{where gcall} = \text{NextS}(f, \text{jal} f') \]

\[\forall S', g', S \rightarrow (S, R) \rightarrow S' \rightarrow (S', R) \rightarrow (S'') \rightarrow S'' \rightarrow (S'', R) \rightarrow (S', R) \rightarrow (S', R) \rightarrow (S, R) \rightarrow \Psi \vdash \{(p, g)\} f: \text{jal} f'; \text{I} \]

(CALL)

\[\Psi \vdash \{(p, g)\} f: \text{jal} f'; \text{I} \quad \text{p} \circ \text{gid} \Rightarrow g \quad \Psi \vdash \{(p, g)\} f: \text{jal} f'; \text{I} \]

(RET)

\[(f', (p', g')) \in \Psi \quad \Psi \vdash \{(p', g')\} f + 1: \text{I} \quad \text{p} \triangleright \text{gcall} \Rightarrow p' \quad \text{p} \circ \text{gcall} \circ g' \Rightarrow g'' \quad \text{where gcall} = \text{NextS}(f, \text{jal} f') \]

\[\forall S', g', S \rightarrow (S, R) \rightarrow S' \rightarrow (S', R) \rightarrow (S', R) \rightarrow (S, R) \rightarrow (S', R) \rightarrow (S', R) \rightarrow (S, R) \rightarrow \Psi \vdash \{(p, g)\} f: \text{jal} f'; \text{I} \]

(T-CALL)

\[\Psi \vdash \{(p', g')\} f + 1: \text{I} \quad (f', (p', g')) \in \Psi \quad \text{p} \triangleright \text{gcall} \Rightarrow p' \quad \text{p} \circ \text{gcall} \circ g' \Rightarrow g'' \quad \text{where gcall} = \text{NextS}(f, \text{jal} f') \]

\[\Psi \vdash \{(p', g')\} f: \text{jal} f'; \text{I} \quad \text{p} \circ \text{gid} \Rightarrow g \quad \Psi \vdash \{(p, g)\} f: \text{jal} f'; \text{I} \]

(BGTZ)

\[\Psi \vdash C; \Psi' \] (Well-Formed Code Heaps)

\[\text{for all } (f, \theta) \in \Psi': \quad \Psi \vdash \{f, \} \ f: C[f] \]

(CDHP)

\[\Psi_1 \vdash C_1; \Psi_1' \quad \Psi_2 \vdash C_2; \Psi_2' \quad C_1 \sqsubseteq C_2 \quad \Psi_1 \cup \Psi_2 \vdash C_1 \cup C_2; \Psi_1' \cup \Psi_2' \]

(LINK)

\[\Psi \vdash P \] (Well-Formed Programs)

\[\Psi \vdash C; \Psi' \quad \Psi \subseteq \Psi' \quad p S \wedge \text{WFST}(n, g, S, \Psi') \quad \Psi \vdash \{(p, g)\} pc: C[pc] \]

\[\Psi \vdash (C, S, pc) \quad \text{(PROG)}

Fig. 4. Inference Rules for SCAP
It also requires the basic block $C[p;c]$ is well-formed with respect to some $(p,g)$, $p$ holds over the current state, and there is a well-formed control stack. The assertion $WFST(n,g,S,\Psi)$ is defined below:

\[
WFST(0,g,S,\Psi) \triangleq \neg \exists S'. g S S' \\
WFST(n+1,g,S,\Psi) \triangleq \forall S'. g S S' \rightarrow \exists p',g'. (\Psi(S',R($ra)) = (p',g')) \\
\land p' S' \land WFST(n,g',S',\Psi)
\]

There is a well-formed stack with depth $n+1$ at the state $S$ if, after the remaining guarantee $g$ has been fulfilled, the return address in the resulting state $S'$ is specified in $\Psi$ with some specification $(p',g')$. $p'$ holds over $S'$, and there is a well-formed stack with depth $n$ in $S'$. If the stack depth is 0, we are at the top-most level and the code is not supposed to return, so we simply require that the remaining guarantee $g$ is not satisfiable.

**Soundness.** The soundness of the program logic can be proved following the syntactic approach to type soundness proof. We prove that if the program $P$ is well formed with respect to $\Psi$ (i.e., $\Psi \vdash P$), then $P$ can execute one step further to reach a new program configuration $P'$ (the progress property), and $P'$ is also well-formed with respect to $\Psi$ (the preservation property).

## 4 Local Reasoning in SCAP

In this section, we use a very simple example to show that the $g$ used in SCAP has natural support of local reasoning without using the frame rule. We first define separation logic connectors and common assertions in Fig. 5.

Figure 6 shows a function `double`. It takes an argument in the register $\$a0$, which points to a memory cell. When the function returns, the value in the memory cell is
Lemma 2. If \( \Psi \vdash \{ (p_1, g_1) \} f + 1 : I \) and \( \{ \text{double} \sim (p_1, g_1) \} \subseteq \Psi \), we have \( \Psi \vdash \{ (p_0, g_0) \} f : \text{jal} \text{ double}; I \).

The Lemma can be proved by applying the \textit{call} rule and proving all the premises of the \textit{call} rule hold.

This example shows SCAP’s support of local reasoning because, in \((p_1, g_1)\), we do not need any knowledge of \(m\) and \(m'\). The specification is polymorphic over the part of memory untouched by the function. Unlike Separation Logic, we are not using small footprint specifications and \(p_1\) is \textit{intuitionistic} in the sense that, if it holds over a state, it holds over all states having a bigger heap [10]. The guarantee \(g_1\) satisfies a similar property. We will formalize these properties in the next Section.

\begin{verbatim}
8  Xinyu Feng and Zhong Shao

0  double: -}\{(p_1, g_1)\}
  1  lw  $t0, 0($a0)  ;; $t0 <- [$a0]
  2  addu $t0, $t0, $t0  ;; $t0 <- $t0 + $t0
  3  sw  $t0, 0($a0)  ;; [$a0] <- $t0
  4  jr  $ra  ;; return

Fig. 6. Example 1: Double Value
\end{verbatim}
enableS(g) \triangleq \lambda S. \exists g' g S S'

enable(p,g) \triangleq p \Rightarrow enableS(g)

monotonic(p) \triangleq \forall H, R, p (H, R) \rightarrow \forall H' \supseteq H. p (H', R)

monotonic(g) \triangleq monotonic(enableS(g))

frame(g) \triangleq \forall H_1, R_1, H_2, R_2, H_3, H_4, g (H_1, R_1) (H_2, R_2) \wedge (H_1 = H_3 \cup H_4) \wedge enableS(g) (H_3', R_1) \rightarrow \exists H_2'. (H_2 = H_3' \cup H_4) \wedge g (H_3', R_1) (H_2', R_2)

precise(m) \triangleq \forall H, H', H''. (H' \subseteq H) \wedge (H'' \subseteq H) \wedge m \Rightarrow H' \wedge m \Rightarrow H'' \Rightarrow H' = H''

LocalSpec(p,g) \triangleq enable(p,g) \wedge monotonic(p) \wedge monotonic(g) \wedge frame(g)

LocalSpec(\Psi) \triangleq \forall (f, \theta) \in \Psi. LocalSpec(\theta)

Fig. 7. Definitions for Frame Rules

5 Information Hiding in SCAP

In this section, we show the support of information hiding in SCAP. We first define some properties of SCAP specifications in Fig. 7. enableS(g) \subseteq holds if and only if the transition g starting from S can go through without getting stuck. enable(p,g) says all states satisfying p would enable g. monotonic(p) means heap extensions preserve the validity of p (i.e., p is intuitionistic). monotonic(g) means heap extensions preserve enableS(g). frame(g) says the transition g satisfies the frame property: if a smaller heap \( H'_1 \) enables g, then the transition g starting from a bigger heap \( H'_1 \) can be tracked back to the transition starting from \( H'_1 \). Here the definitions of monotonic(g) and frame(g) follow Yang and O’Hearn’s definition of safety monotonicity and frame property [14], respectively. Also, following Separation Logic, m is precise (precise(m)) if, given a heap, there is at most one sub-heap satisfying m. LocalSpec(p,g) adds constraints over the (p,g) pair. LocalSpec(\Psi) holds if every specification in \( \Psi \) satisfies LocalSpec. Lemmas 3 and 4 show properties of the enable relation, monotonicity and frame property.

Lemma 3. For any p, m, g and g',

1. if enable(p,g) and monotonic(g), then enable(p \ast m, g);
2. if monotonic(g) and monotonic(g'), then monotonic(g \circ g');
3. if frame(g) and frame(g'), then frame(g \circ g').

Lemma 4. For any f and c, we have monotonic(NextS(f,c)) and frame(NextS(f,c)).

Lemma 5 shows that the following hypothetical frame rule is admissible in SCAP, as long as specifications in \( \Psi_1 \) are local specifications and m is precise:

\[
\begin{align*}
\Psi &= \Psi_1 \cup \Psi_2 \\
\Psi_1 \cup (\Psi_2 \ast m) &\vdash \{(p,g)\} f : I \\
(\text{HYP-FRAME})
\end{align*}
\]
Lemma 5 (Hypothetical Frame Rule).
If $\Psi \vdash \{(p, g)\} f : \I$, $\Psi = \Psi_1 \cup \Psi_2$, $\text{LocalSpec}(\Psi_1)$, and precise($m$), we have

$$\Psi_1 \cup (\Psi_2 + m) \vdash \{(p + m, g + m)\} f : \I.$$ 

Proof sketch. We do induction over the shape of $\I$. By $\Psi \vdash \{(p, g)\} f : \I$ and inversion of the corresponding instruction rule, we know premises of the corresponding instruction rule hold. Based on these premises, we can prove the premises required to derive $\Psi_1 \cup (\Psi_2 + m) \vdash \{(p + m, g + m)\} f : \I$. Lemma 4 and precise($m$) are essential for every commands. We need $\text{LocalSpec}(\Psi_1)$ for jal, j and bgtz commands. □

As shown by O’Hearn et al. [10], the first-order frame rule can be derived from the hypothetical frame rule. Lemma 6 trivially follows Lemma 5 by initializing $\Psi_2$ as $\emptyset$.

Lemma 6 (First-Order Frame Rule).
If $\Psi \vdash \{(p, g)\} f : \I$, $\text{LocalSpec}(\Psi)$, and precise($m$), we have $\Psi \vdash \{(p + m, g + m)\} f : \I$.

Applying the Hypothetical Frame Rule in SCAP. We port the example of memory manager by O’Hearn et al. [10] to SCAP and show how to apply the hypothetical frame rule for information hiding.

To implement memory allocation and free routines, we maintain a linked list of free memory blocks. The free list is pointed to by some global constant $fl$. The well-formedness of the list is specified as $\text{FList}(fl)$. We use its definition here.

The function $\text{alloc}$ takes an argument $n$ in the register $\$a0$, gets a free memory block of size $n$ from the free memory list, and returns the pointer of the memory block in the register $\$v0$ (the register $r_2$). The function preserves the well-formedness ($\text{FList}(fl)$) of the free memory list. We use $(p_1, g_1)$ to specify $\text{alloc}$:

\[
\begin{align*}
p_0 &\triangleq \lambda(H, R). R(\$a0) > 0 \\
g_0 &\triangleq \lambda(H, R). (H', R'). \\
&\quad \exists n > 0. \left( \{ \text{emp} \} \cap R(\$v0) \rightarrow \{ \begin{array}{c} n \\
\cdots \end{array} \} \cap H \right) \wedge R(\$a0) = n \wedge \text{trash(...) R R'} \\
p_1 &\triangleq p_0 \ast \text{FList}(fl) \\
g_1 &\triangleq g_0 \ast \text{FList}(fl)
\end{align*}
\]

Suppose the implementation $C_{\text{alloc}}$ of the function is certified using the interface. We have

$$\Psi_{\text{alloc}} \vdash C_{\text{alloc}} : \Psi'_{\text{alloc}},$$

where $\{\text{alloc} \leadsto (p_1, g_1)\} \subseteq \Psi'$, i.e., $(p_1, g_1)$ is exported in $\Psi'_{\text{alloc}}$.

In some client code, we have the basic block $f : \text{jal alloc; I}$ that calls alloc. To certify the client code, we can use $(p_0, g_0)$ as the interface for $\text{alloc}$. Suppose we have

$$\Psi \cup \{\text{alloc} \leadsto (p_0, g_0)\} \vdash \{(p, g)\} f : \text{jal alloc; I}.$$ (2)

We do not need to specify $\text{FList}(fl)$ in (p, g), because $\text{FList}(fl)$ is hidden in $(p_0, g_0)$ and the client code never directly accesses the free list. By the CDHP rule we can get:
Local Reasoning and Information Hiding in SCAP

$$\Psi \cup \{\text{alloc} \leadsto (p_0, g_0)\} \vdash \{\text{jal} \text{ alloc free}\} : \{f \leadsto (p, g)\}. \quad (3)$$

Next, we may want to apply the \textit{LINK} rule to link the client code and \text{alloc} to get:

$$\Psi_{\text{alloc}} \cup \Psi_{\text{tmp}} \vdash C_{\text{alloc}} \cup C_{\text{client}} : \Psi_{\text{alloc}}' \cup \Psi_{\text{tmp}}', \quad (4)$$

where $\Psi_{\text{tmp}} = \Psi \cup \{\text{alloc} \leadsto (p_0, g_0)\}$, $C_{\text{client}} = \{f \leadsto \text{jal alloc free}\}$, and $\Psi_{\text{alloc}}' = \{f \leadsto (p, g)\}$. However, this would not give us a well-formed whole system because our \textit{PROG} rule requires that, after linking, the imported interface must be consistent with the exported interface (the premise $\Psi \subseteq \Psi'$ in the \textit{PROG} rule). In (4), \text{alloc} has different imported and exported specifications.

To get the fully certified system, we need to apply the hypothetical frame rule over (2), and get:

$$\Psi \cup \{\text{alloc} \leadsto (p_1, g_1)\} \vdash \{(p \ast F\text{List}(f_1), g \ast F\text{List}(f_1))\} : \text{jal alloc free}. \quad (5)$$

Then we can repeat the procedure above by applying the \textit{CDHP} rule and the \textit{LINK} rule to get the certified whole system:

$$\Psi_{\text{alloc}} \cup \Psi_{\text{client}} \vdash C_{\text{alloc}} \cup C_{\text{client}} : \Psi_{\text{alloc}}' \cup \Psi_{\text{client}}', \quad (6)$$

where $\Psi_{\text{client}} = \Psi \cup \{\text{alloc} \leadsto (p_1, g_1)\}$, and $\Psi_{\text{client}}' = \{f \leadsto (p \ast F\text{List}(f_1), g \ast F\text{List}(f_1))\}$.

Note that, although we expose $F\text{List}(f_1)$ in the final specification of \text{alloc} and the client code after linking, this does not break our goal of information hiding because $F\text{List}(f_1)$ is hidden when we derive (2).

6 Related Work and Conclusion

O’Hearn, Yang and Reynolds [10] proposed the hypothetical frame rule and showed that the first-order frame rule is derivable from it. Their frame rules are built-in rules in the logic, instead of admissible rules. To prove the soundness, semantics of primitive commands in the language needs to satisfy safety monotonicity and frame property [14]. The hypothetical frame rule in SCAP is similar to the one by O’Hearn et al. We prove it as an admissible rule of the logic. Since SCAP view functions as an abstract command whose behavior is specified by $g$, we enforce constraints similar to the safety monotonicity and frame property over function specifications.

Calcagno, O’Hearn and Yang [2] proposed an abstract separation logic with a general notion of local actions. They showed the locality requirement is equivalent to the safety monotonicity plus the frame property [14]. Our LocalSpec constraint over $g$ essentially requires $g$ is a local action.

Benton [1] proposed a typed Compositional logic for a stack-based abstract machine. The difference between his work and SCAP has been discussed in our original paper on SCAP [4]. Benton’s logic also has a frame rule, but the hypothetical frame rule is not supported. Nanevski, Govereau and Morrisett also used a binary relation
over states as program specifications in their recent extension of Hoare Type Theory (HTT) for transactions [8]. They also proved the frame rule is admissible in their logic. HTT is a combination of type theory with Hoare logic for higher-level languages, where functions are specified by syntactic types; while SCAP is a pure logic system with no types in the specifications.

In this paper, we show that the guarantee $g$ in SCAP can specify the preservation of untouched heap conveniently. Using intuitionistic specifications, we can support local reasoning in SCAP without using the frame rule. Treating functions as abstract commands, we show that the hypothetical frame rule is admissible in SCAP if specifications for functions satisfy LocalSpec.

References