Modular Verification of Assembly Code with Stack-Based Control Abstractions

Xinyu Feng[†] Zhong Shao[†] Alexander Vaynberg[†] Sen Xiang[‡] Zhaozhong Ni[†]

[†]Department of Computer Science
Yale University
New Haven, CT 06520-8285, U.S.A.
{feng, shao, alv, ni-zhaozhong}@cs.yale.edu

[‡]Department of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui 230026, China
xiangsen@ustc.edu

Abstract

Runtime stacks are critical components of any modern software—they are used to implement powerful control structures such as function call/return, stack cutting and unwinding, coroutines, and thread context switch. Stack operations, however, are very hard to reason about: there are no known formal specifications for certifying C-style `setjmp/longjmp`, stack cutting and unwinding, or weak continuations (in C--). In many proof-carrying code (PCC) systems, return code pointers and exception handlers are treated as general first-class functions (as in continuation-passing style) even though both should have more limited scopes.

In this paper we show that stack-based control abstractions follow a much simpler pattern than general first-class code pointers. We present a simple but flexible Hoare-style framework for modular verification of assembly code with all kinds of stack-based control abstractions, including function call/return, tail call, `setjmp/longjmp`, weak continuation, stack cutting, stack unwinding, multi-return function call, coroutines, and thread context switch. Instead of presenting a specific logic for each control structure, we develop all reasoning systems as instances of a generic framework. This allows program modules and their proofs developed in different PCC systems to be linked together. Our system is fully mechanized. We give the complete soundness proof and a full verification of several examples in the Coq proof assistant.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.2.4 [*Software Engineering*]: Software/Program Verification — correctness proofs, formal methods

General Terms Languages, Verification

Keywords Assembly Code Verification, Modularity, Stack-Based Control Abstractions, Proof-Carrying Code

1. Introduction

Runtime stacks are critical components of any modern software—they are used to implement powerful control structures such as procedure call/return, tail call [34, 8], C-style `setjmp/longjmp` [20], stack cutting and unwinding (for handling exceptions) [7, 12, 30], coroutines [10], and thread context switch [15]. Correct implementation of these constructs is of utmost importance to the safety and reliability of many software systems.

```
0    f:
1         addiu  $sp, $sp, -32    ;allocate stack frame
2         sw     $fp, 32($sp)     ;save old $fp
3         addiu  $fp, $sp, 32     ;set new $fp
4         sw     $ra, -4($fp)     ;save $ra
5         jal    h                ;call h
6    ct:  lw     $ra, -4($fp)     ;restore $ra
7         lw     $fp, 0($fp)      ;restore $fp
8         addiu  $sp, $sp, 32     ;deallocate frame
9         jr     $ra              ;return
10   h:
11        jr     $ra              ;return
```

Figure 1. Stack-Based Function Call/Return

Stack-based controls, however, can be unsafe and error-prone. For example, both stack cutting and `longjmp` allow cutting across a chain of stack frames and returning immediately from a deeply nested function call. If not done carefully, it can invoke an obsolete `longjmp` or a dead *weak continuation* [30]). Neither C nor C-- [30] provides any formal specifications for certifying `setjmp/longjmp`, stack cutting and unwinding, or weak continuations. In Java virtual machine and Microsoft's .NET IL, operations on native C stacks are not *managed* so they must be trusted.

Stack operations are very hard to reason about because they involve subtle low-level invariants: both return code pointers and exception handlers should have restricted scopes, yet they are often stored in memory or passed in registers—making it difficult to track their lifetime. For instance, the following C program is compiled into the MIPS assembly code shown in Figure 1:

```
void f(){          |        void h(){
    h();           |            return;
    return;        |        }
}                  |
```

Before calling function `h`, the caller `f` first saves its return code pointer (in $ra) on the stack; the instruction jal `h` loads the return address (the label `ct`) in $ra, and jumps to the label `h`; when `h` returns, the control jumps back to the label `ct`, where `f` restores its return code pointer and stack pointers and jumps back to its caller's code. The challenge is to formalize and capture the invariant that `ct` does not outlive `f` even though it can escape into other functions.

Many proof-carrying code (PCC) systems [3, 14, 29] support stack-based controls by using continuation-passing style (CPS) [2]. CPS treats return addresses or exception handlers as first-class code pointers. Under CPS, the code following `ct` (lines 6-9) is treated not as a part of function `f` but as a separate new function; when `h` is called, the continuation function `ct` is passed as an extra argument in $ra which is then called at the end of function `h`. CPS makes typechecking easier but it is still hard to describe the above invariant about `f` and `ct`. Indeed, none of the existing PCC systems [26, 27, 9, 3, 14] have successfully certified `setjmp/longjmp`, weak

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.

Stack-Based Control Abstraction	Reasoning System	Definition & Formalization
function call/return	SCAP	SEC 4
tail call optimization [34, 8]	SCAP	SEC 4.3
exception: stack unwinding [30]	SCAP-I EUCAP	SEC 5.1 TR [13]
exception: stack cutting [30]	SCAP-II ECAP	SEC 5.2 TR [13]
multi-return function call [32]	SCAP-II	SEC 5.2
weak continuation [30]	SCAP-II	SEC 5.2
`setjmp/longjmp` [20]	SCAP-II	SEC 5.3
coroutines [10]	CAP-CR	SEC 6.1
coroutines + function call [10]	SCAP-CR	SEC 6.2
threads [15]	FCCAP	TR [13]

Table 1. A Summary of Supported Control Abstractions

continuations, and general stack cutting and unwinding (see Sec 8 and Sec 2.1 for an in-depth discussion about the related work).

In this paper we describe a formal system that can expose and validate the invariants of stack-based control abstractions. We show that return pointers (or exception handlers) are much more disciplined than general first-class code pointers. A return pointer is always associated with some *logical* control stack whose validity can be established statically. A function can cut to any return pointer if it can establish the validity of its associated logical control stack.

More specifically, we present a simple but flexible Hoare-style framework for modular verification of assembly code with all kinds of stack-based control abstractions (see Table 1). Instead of presenting a specific logic for each construct, we develop all reasoning systems as instances of a generic framework. This allows program modules and their proofs developed in different PCC systems to be linked together. Our system is fully mechanized. We give the complete soundness proof and a full verification of several examples in the Coq proof assistant [35]. Our paper builds upon previous work on program verification but makes the following new contributions:

- As far as we know, our paper is the first to successfully formalize and verify sophisticated stack operations such as `setjmp/longjmp`, weak continuations, and general stack cutting. We verify raw assembly implementation so there is no loss of efficiency or additional runtime check. Our interface is simple, general, yet modular (so a library only needs to be verified once). Our framework is sound: a program certified using our system is free of *unchecked* runtime errors [20, 30].
- We have also done a thorough study of common stack-based control abstractions in the literatures (see Table 1; due to the space limit, several constructs are treated in a companion technical report [13]). For each construct, we formalize its invariants and show how to certify its implementation. As an important advantage, all these systems are instances of a generic framework; in fact, the inference rules for each system are just derived lemmas in the base framework, so programs certified in different PCC systems can be linked together [13].
- Our SCAP system (Sec 4) is interesting and novel in its own right. Instead of treating return pointers as first-class code pointers (which require "impredicative types" [23, 29]), SCAP specifies the invariant at each program point using a pair of a precondition and a "local" guarantee (which states the obligation that the current function must fulfill before it can return or throw an exception). These guarantees, when chained together, is used to specify the logical control stack. SCAP is also orthogonal to the recent work on XCAP [29]: it can apply the same syntactic technique [29] to certify general first-class code pointers.
- Our certified framework is also very flexible. A logical control stack specifies a chain of valid return pointers, but it imposes no restriction on where we store these pointers. Because all invariants are specified as state predicates or state relations, we can support any physical stack layout and calling conventions.

In the rest of this paper, we first review common stack-based controls and summarize our main approach (Sec 2). We then define our machine platform and a generic Hoare-style framework (Sec 3). We present our SCAP system for certifying function call/return and show how to extend it to support different control abstractions in Table 1 (Sec 4–7 and the companion TR [13]). Finally we discuss implementation and related work, and then conclude.

2. Background and Related Work

Before giving an overview of our approach, we first survey common stack-based control abstractions in the literatures:

- *Function call/return* follow a strict "last-in, first-out" pattern: the callee always returns to the point where it was most recently called. Similar concepts include the JVM *subroutines* [22], which are used to compile the "`try-finally`" block in Java.
- The *tail call optimization* is commonly used in compiler implementation: if a function call occurs at the end of the current function, the callee will reuse the current stack frame and return directly to the caller of the current function.
- *Exceptions, stack unwinding, and stack cutting.* When an exception is raised, the control flow is transferred to the point at which the exception is handled. There are mainly two strategies for implementing exceptions (on stacks) [30]. *Stack unwinding* walks the stack one frame at a time until the handler is reached; intermediate frames contain a default handler that restores values of callee-save registers and re-raises the exception; a function always returns to the activation of its immediate caller. *Stack cutting* sets the stack pointer and the program counter directly to the handler which may be contained in a frame deep on the stack; intermediate frames are skipped over.
- *Weak continuations and setjmp/longjmp*. C-- uses weak continuations [30] to support different implementation strategies for exceptions. A weak continuation is similar to the first-class continuation except that it can only be defined inside a procedure and cannot outlive the activation of the enclosing procedure. C uses `setjmp/longjmp` library functions [20] to enable an immediate return from a deeply nested function call, the semantics of which is similar to weak-continuations (while the implementation may be more heavyweight). Especially, the function containing the `setjmp` must not have terminated when a `longjmp` is launched. Both C-- and C make no effort to prohibit invocation of a dead weak continuation or an obsolete `longjmp`.
- *Multi-return function call*. Shivers and Fisher [32] proposed MRLC to allow functions to have multiple return points, whose expressiveness sits between general CPS and first-order functions. The mechanism is similar to weak continuations, but proposed at a higher abstract level. Multi-return function call supports pure stack-based implementations.
- *Coroutines and threads* involve multiple execution contexts that exist concurrently. Control can be transferred from one execution context to another. Implementation of context switch does not follow the regular function calling convention: it fetches the return code pointer from the stack of the target coroutine (thread) and returns to the target instead of its caller.

2.1 Reasoning about Control Abstractions

Traditional Hoare-logic [16] uses the pre- and postcondition as specifications for programs. Most work on Hoare-logic [4] reasons

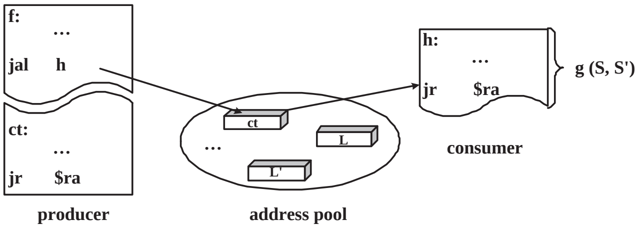


Figure 2. The Model for Code Pointers

about control structures in higher-level languages and does not directly reason about return code pointers in their semantics. To apply traditional Hoare-logic to generate mechanized proofs for low-level code, we need to first formalize auxiliary variables and the Invariance rule, which is a non-trivial issue and complicates the formalization, as shown in pervious work [37, 5]; next, we need to relate the entry point with the exit point of a function and show the validity of return code pointers—this is hard at the assembly level due to the lack of abstractions.

Stata and Abadi [33] also observed two similar challenges for typechecking Java byte code subroutines. They propose a Hoare-style type system to reason about subroutine calls ("jsr L") and returns ("ret x"). To ensure the return address used by a subroutine is the one that is most recently pushed onto the stack, they have to disallow recursive function calls, and require labeling of code to relate the entry point with the return point of subroutines.

Necula used Hoare triples to specify functions in SAL [26]. He needs a history $\mathcal{H}$ of states, which contains copies of the register file and the whole memory at the moment of function invocations. At the return point, the last state is popped up from $\mathcal{H}$ and the relation between that state and the current state is checked. Not a model of physical stacks, $\mathcal{H}$ is used purely for reasoning about function calls; it complicates the operational semantics of SAL significantly. Also, SAL uses a very restrictive physical stack model where only contiguous stack is supported and general pointer arguments (which may point into the stack) are not allowed.

To overcome the lack of structures in low-level code, many PCC systems have also used CPS to reason about regular control abstractions, which treats return code pointers (and exception handlers) as first-class code pointers. CPS is a general semantic model to support all the control abstractions above, but it is hard to use CPS to characterize the invariants of control stacks for specific control abstractions (*e.g.,* `setjmp/longjmp` and weak continuation). CPS-based reasoning also requires specification of continuation pointers using "impredicative types" [23, 29]), which makes the program specification complex and hard to understand. Another issue with CPS-based reasoning is the difficulty to specify first-class code pointers modularly in logic: because of the circular references between code pointers and data heap (which may in turn contains code pointers), it is not clear how to apply existing approaches [25, 3, 29] to model sophisticated stack-based invariants.

2.2 Our Approach

In this paper we will show that we can support modular reasoning of stack-based control abstractions without treating them as first-class code pointers. In our model, when a control transfer occurs, the pointer for the continuation code is deposited into an abstract "address pool" (which may be physically stored in memory or the register file). The code that saves the continuation is called a "producer", and the code that uses the continuation later is called a "consumer". In case of function calls, as shown in Figure 2, the caller is the "producer" and the callee is the "consumer", while the return address is the continuation pointer.

$$\begin{array}{rll}
(\mathit{Program}) & \mathbb{P} & ::= (\mathbb{C},\mathbb{S},\mathbb{I}) \\
(\mathit{CodeHeap}) & \mathbb{C} & ::= \{\mathtt{f} \rightsquigarrow \mathbb{I}\}^* \\
(\mathit{State}) & \mathbb{S} & ::= (\mathbb{H},\mathbb{R}) \\
(\mathit{Heap}) & \mathbb{H} & ::= \{\mathtt{l} \rightsquigarrow \mathtt{w}\}^* \\
(\mathit{RegFile}) & \mathbb{R} & ::= \{\mathtt{r} \rightsquigarrow \mathtt{w}\}^* \\
(\mathit{Register}) & \mathtt{r} & ::= \{\mathtt{r}_k\}^{k\in\{0\ldots31\}} \\
(\mathit{Labels}) & \mathtt{f},\mathtt{l} & ::= i \;\; (\mathit{nat\ nums}) \\
(\mathit{Word}) & \mathtt{w} & ::= n \;\; (\mathit{integers}) \\
(\mathit{InstrSeq}) & \mathbb{I} & ::= \mathtt{c};\mathbb{I} \mid \mathsf{j}\ \mathtt{f} \mid \mathsf{jal}\ \mathtt{f},\mathtt{f}_{ret} \mid \mathsf{jr}\ \mathtt{r}_s \\
(\mathit{Command}) & \mathtt{c} & ::= \mathsf{addu}\ \mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t \mid \mathsf{addiu}\ \mathtt{r}_d,\mathtt{r}_s,\mathtt{w} \\
 & & \quad \mid \mathsf{beq}\ \mathtt{r}_s,\mathtt{r}_t,\mathtt{f} \mid \mathsf{bgtz}\ \mathtt{r}_s,\mathtt{f} \mid \mathsf{lw}\ \mathtt{r}_t,\mathtt{w}(\mathtt{r}_s) \\
 & & \quad \mid \mathsf{subu}\ \mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t \mid \mathsf{sw}\ \mathtt{r}_t,\mathtt{w}(\mathtt{r}_s)
\end{array}$$

Figure 3. Syntax of Target Machine TM

$zero	$\mathtt{r}_0$	always zero
$at	$\mathtt{r}_1$	assembler temporary
$v0 − $v1	$\mathtt{r}_2-\mathtt{r}_3$	return values
$a0 − $a3	$\mathtt{r}_4-\mathtt{r}_7$	arguments
$t0 − $t9	$\mathtt{r}_8-\mathtt{r}_{15},\mathtt{r}_{24}-\mathtt{r}_{25}$	temporary (caller saved)
$s0 − $s7	$\mathtt{r}_{16}-\mathtt{r}_{23}$	callee saved
$k0 − $k1	$\mathtt{r}_{26}-\mathtt{r}_{27}$	kernel
$gp	$\mathtt{r}_{28}$	global pointer
$sp	$\mathtt{r}_{29}$	stack pointer
$fp	$\mathtt{r}_{30}$	frame pointer
$ra	$\mathtt{r}_{31}$	return address

Figure 4. Register Aliases and Usage

The producer is responsible for ensuring that each code pointer it deposits is a valid one and depositing the code pointer does not break the *invariant* of the address pool. The consumer ensures that the invariant established at its entry point still holds when it fetches the code pointer from the pool and makes an indirect jump. The validity of the code pointer is guaranteed by the invariant. To overcome the lack of abstraction at the assembly level, we use a guarantee g—a relation over a pair of states—to bridge the gap between the entry and exit points of the consumer. This approach avoids maintaining any state history or labeling of code.

The address pool itself is structureless, with each control abstraction molding the pool into the needed shape. For functions, exceptions, weak continuations, *etc.*, the pool takes the form of a stack; for coroutines and threads it takes the form of a queue or a queue of stacks (each stack corresponding to a coroutine/thread). The invariant specified by a control abstraction also restricts how the pool is used. Function call, for example, restricts the (stack-shaped) pool to a strict "last-in, first-out" pattern, and makes sure that all addresses remain constant until they are fetched.

In the rest of this paper, we will describe the invariant for each control abstraction. We also present a set of lemmas that allow programmers to verify structureless assembly code with higher-level abstractions. Before we define these systems, we first present our generic CAP0 framework. All the systems for specific control abstractions will be presented as a set of lemmas in CAP0.

3. The CAP0 Framework

In this section, we first present a MIPS-style "untyped" target machine language (TM) and its operational semantics. Then we propose a general logic, CAP0, for verifying TM programs. The generic CAP0 framework will serve as the common basis for the interoperability of different logics.

3.1 The Target Machine

In Figure 3 we show the definition of a MIPS-style target machine (TM). A machine state is called a "Program" ($\mathbb{P}$), which consists of

if $\mathbb{I} =$	then $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}) \longmapsto$
j $\mathtt{f}$	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(\mathtt{f}))$ when $\mathtt{f} \in dom(\mathbb{C})$
jal $\mathtt{f}, \mathtt{f}_{ret}$	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}\{\mathtt{r}_{31} \leadsto \mathtt{f}_{ret}\}), \mathbb{C}(\mathtt{f}))$ when $\mathtt{f} \in dom(\mathbb{C})$
jr $\mathtt{r}_s$	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(\mathbb{R}(\mathtt{r}_s)))$ when $\mathbb{R}(\mathtt{r}_s) \in dom(\mathbb{C})$
beq $\mathtt{r}_s, \mathtt{r}_t, \mathtt{f}; \mathbb{I}'$	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}')$ when $\mathbb{R}(\mathtt{r}_s) \neq \mathbb{R}(\mathtt{r}_t)$; $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(\mathtt{f}))$ when $\mathbb{R}(\mathtt{r}_s) = \mathbb{R}(\mathtt{r}_t)$, $\mathtt{f} \in dom(\mathbb{C})$
bgtz $\mathtt{r}_s, \mathtt{f}; \mathbb{I}'$	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}')$ when $\mathbb{R}(\mathtt{r}_s) \leq 0$; $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(\mathtt{f}))$ when $\mathbb{R}(\mathtt{r}_s) > 0$, $\mathtt{f} \in dom(\mathbb{C})$
c; $\mathbb{I}'$	$(\mathbb{C}, \mathtt{Next}_{\mathtt{c}}(\mathbb{H}, \mathbb{R}), \mathbb{I}')$

where

if c $=$	then $\mathtt{Next}_{\mathtt{c}}(\mathbb{H}, \mathbb{R}) =$
addu $\mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{R}(\mathtt{r}_s) + \mathbb{R}(\mathtt{r}_t)\})$
addiu $\mathtt{r}_d, \mathtt{r}_s, \mathtt{w}$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{R}(\mathtt{r}_s) + \mathtt{w}\})$
lw $\mathtt{r}_t, \mathtt{w}(\mathtt{r}_s)$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_t \leadsto \mathbb{H}(\mathbb{R}(\mathtt{r}_s) + \mathtt{w})\})$ when $\mathbb{R}(\mathtt{r}_s) + \mathtt{w} \in dom(\mathbb{H})$
subu $\mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{R}(\mathtt{r}_s) - \mathbb{R}(\mathtt{r}_t)\})$
sw $\mathtt{r}_t, \mathtt{w}(\mathtt{r}_s)$	$(\mathbb{H}\{\mathbb{R}(\mathtt{r}_s) + \mathtt{w} \leadsto \mathbb{R}(\mathtt{r}_t)\}, \mathbb{R})$ when $\mathbb{R}(\mathtt{r}_s) + \mathtt{w} \in dom(\mathbb{H})$

Figure 5. Operational Semantics of TM

a read-only code heap ($\mathbb{C}$), an updatable state ($\mathbb{S}$), and an instruction sequence ($\mathbb{I}$). The code heap is a finite partial mapping from code labels to instruction sequences. The state $\mathbb{S}$ contains a data heap ($\mathbb{H}$) and a register file ($\mathbb{R}$). Each instruction sequence is a basic code block, *i.e.,* a list of instructions ending with a jump-instruction. We use an instruction sequence $\mathbb{I}$ in $\mathbb{P}$ (rather than a program counter) to represent the basic block that is being executed.

The target machine has 32 registers. Following the MIPS convention, Figure 4 shows the register aliases and usage. All the assembly code shown in the rest of the paper follows this convention.

The instruction set captures the most basic and common MIPS instructions. Since we do not have a program counter, we change the syntax of the jal instruction and require that the return address be explicitly given. The execution of TM programs is modeled as a small-step transition from one program to another, *i.e.,* $\mathbb{P} \longmapsto \mathbb{P}'$. Figure 5 defines the program transition function. The semantics of most instructions are the same with corresponding MIPS instructions, except that code labels in jump-instructions (*e.g.,* j $\mathtt{f}$, jr $\mathtt{r}$) and branch-instructions (*e.g.,* beq $\mathtt{r}_s, \mathtt{r}_t, \mathtt{f}$) are treated as absolute addresses instead of relative addresses.

3.2 The CAP0 Framework

CAP0 generalizes our previous work on CAP systems [39, 29]. It leaves the program specification unspecified, which can be customized to embed different logics into the framework. The soundness of CAP0 is independent of specific forms of program specifications. The framework supports separate verification of program modules using different verification logics.

3.2.1 Program Specifications

The verification constructs are defined as follows.

$$
\begin{array}{rccl}
(\mathit{CdHpSpec}) & \Psi & ::= & \{\mathtt{f} \leadsto \theta\}^* \\
(\mathit{CdSpec}) & \theta & ::= & \ldots \\
(\mathit{Interp.}) & \mathtt{a}, \llbracket \theta \rrbracket, \langle \mathtt{a} \rangle_{\Psi} & \in & \mathit{CdHpSpec} \rightarrow \mathit{State} \rightarrow \mathit{Prop}
\end{array}
$$

To verify a program, the programmer needs to first give a specification Ψ of the code heap, which is a finite mapping from code labels to code specifications θ. To support different verification methodology, the CAP0 framework does not enforce the form of θ. Instead, it requires the programmer to provide an interpretation function $\llbracket \_ \rrbracket$ which maps θ to predicates ($\mathtt{a}$) over the code heap specification and the program state. CAP0 uses the interpretation of code specifications as its assertion language.

$\boxed{\Psi \vdash \{\mathtt{a}\}\, \mathbb{P}}$ ***(Well-formed Program)***

$$
\frac{\Psi_{\mathrm{G}} \vdash \mathbb{C} : \Psi_{\mathrm{G}} \quad (\mathtt{a}\, \Psi_{\mathrm{G}}\, \mathbb{S}) \quad \vdash \{\mathtt{a}\}\, \mathbb{I}}{\Psi_{\mathrm{G}} \vdash \{\mathtt{a}\}\, (\mathbb{C}, \mathbb{S}, \mathbb{I})} \qquad \text{(PROG)}
$$

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

$$
\frac{\mathtt{a} = \llbracket \theta \rrbracket \quad \vdash \{\langle \mathtt{a} \rangle_{\Psi_{\mathrm{L}}}\}\, \mathbb{I}}{\Psi_{\mathrm{L}} \vdash \{\mathtt{f} \leadsto \mathbb{I}\} : \{\mathtt{f} \leadsto \theta\}} \qquad \text{(CDHP)}
$$

$$
\frac{\begin{array}{l} \Psi_1 \vdash \mathbb{C}_1 : \Psi_1' \quad \Psi_2 \vdash \mathbb{C}_2 : \Psi_2' \qquad dom(\mathbb{C}_1) \cap dom(\mathbb{C}_2) = \emptyset \\ \qquad \forall \mathtt{f} \in dom(\Psi_1) \cap dom(\Psi_2).\ \Psi_1(\mathtt{f}) = \Psi_2(\mathtt{f}) \end{array}}{\Psi_1 \cup \Psi_2 \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1' \cup \Psi_2'} \qquad \text{(LINK)}
$$

$\boxed{\vdash \{\mathtt{a}\}\, \mathbb{I}}$ ***(Well-formed Instruction Sequence)***

$$
\frac{\forall \Psi, \mathbb{S}.\ \mathtt{a}\, \Psi\, \mathbb{S} \rightarrow \llbracket \Psi(\mathtt{f}) \rrbracket\, \Psi\, \mathbb{S}}{\vdash \{\mathtt{a}\}\, \mathsf{j}\ \mathtt{f}} \qquad \text{(J)}
$$

$$
\frac{\forall \Psi, \mathbb{H}, \mathbb{R}.\ \mathtt{a}\, \Psi\, (\mathbb{H}, \mathbb{R}) \rightarrow \llbracket \Psi(\mathtt{f}) \rrbracket\, \Psi\, (\mathbb{H}, \mathbb{R}\{\mathsf{ra} \leadsto \mathtt{f}_{ret}\})}{\vdash \{\mathtt{a}\}\, \mathsf{jal}\ \mathtt{f}, \mathtt{f}_{ret}} \qquad \text{(JAL)}
$$

$$
\frac{\forall \Psi, \mathbb{S}.\ \mathtt{a}\, \Psi\, \mathbb{S} \rightarrow \llbracket \Psi(\mathbb{S}.\mathbb{R}(\mathtt{r}_s)) \rrbracket\, \Psi\, \mathbb{S}}{\vdash \{\mathtt{a}\}\, \mathsf{jr}\ \mathtt{r}_s} \qquad \text{(JR)}
$$

$$
\frac{\begin{array}{l} \vdash \{\mathtt{a}'\}\, \mathbb{I} \\ \forall \Psi, \mathbb{S}.\ \mathtt{a}\, \Psi\, \mathbb{S} \rightarrow ((\mathbb{S}.\mathbb{R}(\mathtt{r}_s) \leq 0 \rightarrow \mathtt{a}'\, \Psi\, \mathbb{S}) \wedge \\ \qquad\qquad\qquad (\mathbb{S}.\mathbb{R}(\mathtt{r}_s) > 0 \rightarrow \llbracket \Psi(\mathtt{f}) \rrbracket\, \Psi\, \mathbb{S})) \end{array}}{\vdash \{\mathtt{a}\}\, \mathsf{bgtz}\ \mathtt{r}_s, \mathtt{f}; \mathbb{I}} \qquad \text{(BGTZ)}
$$

$$
\frac{\begin{array}{l} \mathtt{c} \in \{\mathsf{addu}, \mathsf{addiu}, \mathsf{lw}, \mathsf{subu}, \mathsf{sw}\} \\ \forall \Psi, \mathbb{S}.\ \mathtt{a}\, \Psi\, \mathbb{S} \rightarrow \mathtt{a}'\, \Psi\, (\mathtt{Next}_{\mathtt{c}}(\mathbb{S})) \qquad \vdash \{\mathtt{a}'\}\, \mathbb{I} \end{array}}{\vdash \{\mathtt{a}\}\, \mathtt{c}; \mathbb{I}} \qquad \text{(SEQ)}
$$

Figure 6. Inference Rules for CAP0

To support separate verification of modules, we add an extra constraint on the arguments of $\mathtt{a}$ using the lifting function $\langle \_ \rangle_{\Psi}$, which says that the specification Ψ of the *local module* is the smallest set of code specifications we need to know to verify this module. The lifting function is defined as follows:

$$
\langle \mathtt{a} \rangle_{\Psi} \triangleq \lambda \Psi'.\lambda \mathbb{S}.(\Psi \subseteq \Psi') \wedge \mathtt{a}\, \Psi'\, \mathbb{S}.
$$

We will give a detailed explanation of CAP0's support of modularity in the next section.

3.2.2 Inference Rules and Soundness

We use the following judgments to define inference rules:

$$
\begin{array}{rl}
\Psi \vdash \{\mathtt{a}\}\, \mathbb{P} & \text{(well-formed program)} \\
\Psi \vdash \mathbb{C} : \Psi' & \text{(well-formed code heap)} \\
\vdash \{\mathtt{a}\}\, \mathbb{I} & \text{(well-formed instruction sequence)}
\end{array}
$$

Figure 6 shows the inference rules of CAP0.

A program is well-formed (the PROG rule) if there exists a *global* code heap specification Ψ_{G} and an assertion $\mathtt{a}$ such that:

- the *global* code heap $\mathbb{C}$ is well-formed with respect to Ψ_{G};
- given Ψ_{G}, the current state $\mathbb{S}$ satisfies the assertion $\mathtt{a}$; and
- the current instruction sequence $\mathbb{I}$ is well-formed.

The CAP0 framework supports *separate verification* of program modules. Modules are modeled as small code heaps which contain at least one code block. The specification of a module contains not only specifications of the code blocks in the current module, but also specifications of external code blocks which will be called by the module. In the judgment $\Psi \vdash \mathbb{C} : \Psi'$, Ψ contains specifications for imported external code and for code within the module $\mathbb{C}$ (to support recursive functions), while Ψ' contains specifications for exported interfaces for other modules. Programmers are required to first establish the well-formedness of each individual module via the CDHP rule. Two non-intersecting well-formed modules can then be linked together via the LINK rule. The PROG rule requires that all modules be linked into a well-formed global code heap.

In the CDHP rule, the user specification θ (for $\mathbb{I}$) is first mapped to a predicate over the code heap specification and the program state, and then lifted by the lifting function parameterized by the *local specification* Ψ_{L} of this module. Later on, we will see that none of the instruction rules (*e.g.,* J and JAL) refer to the global program specification Ψ_{G}. Instead, a universally quantified Ψ is used with the constraint that it must be a superset of Ψ_{L}. Such a constraint is enforced by the lifting function $\langle \_ \rangle_{\Psi_{\mathrm{L}}}$.

The well-formedness of instruction sequences ensures that it is safe to execute $\mathbb{I}$ in a machine state satisfying the assertion $\mathtt{a}$. An instruction sequence beginning with $\mathtt{c}$ is safe (rule SEQ) if we can find an assertion $\mathtt{a}'$ which serves both as the postcondition of $\mathtt{c}$ (that is, $\mathtt{a}'$ holds on the updated machine state after executing $\mathtt{c}$, as captured by the implication) and as the precondition of the tail instruction sequence. A direct jump is safe (rule J) if the current assertion can imply the assertion of the target code block as specified in Ψ. Rules for other jump and branch instructions are similar to the J rule. When proving the well-formedness of an instruction sequence, a programmer's task includes applying the appropriate inference rules and finding intermediate assertions such as $\mathtt{a}'$.

Soundness The soundness of CAP0 inference rules with respect to the operational semantics of TM is established following the syntactic approach [38] to prove type soundness. We do not require the specific form of code specifications θ to prove the soundness.

Lemma 3.1 (Progress) If $\Psi \vdash \{\mathtt{a}\}\,\mathbb{P}$, then there exists a program $\mathbb{P}'$, such that $\mathbb{P} \longmapsto \mathbb{P}'$.

Lemma 3.2 (Preservation) If $\Psi \vdash \{\mathtt{a}\}\,\mathbb{P}$, and $\mathbb{P} \longmapsto \mathbb{P}'$, then there exists $\mathtt{a}'$, $\Psi \vdash \{\mathtt{a}'\}\,\mathbb{P}'$.

Theorem 3.3 (Soundness) If $\Psi \vdash \{\mathtt{a}\}\,\mathbb{P}$, then for all natural number *n*, there exists a program $\mathbb{P}'$ such that $\mathbb{P} \longmapsto^{n} \mathbb{P}'$.

The soundness proof [13] has been formally encoded in Coq.

CAP0 and Previous CAP systems. The CAP0 framework is a generalization of our previous work on CAP systems [39, 29]. The original CAP [39] does not support separate verification of program modules. The idea of letting assertions be parameterized by Ψ and using universally-quantified Ψ in the CAP0 inference rules, is borrowed from Ni and Shao's work on XCAP [29]. XCAP is proposed to reason about general first-class code pointers, where a special form of assertions (with type *State* → *PropX*) is used for program specifications.

CAP0 generalizes XCAP and leaves the form of program specifications unspecified. The interpretation function in CAP0, which is different from the one in XCAP, maps different forms of specifications to a general form. It is trivial to embed the original CAP in CAP0 by the following customization and interpretation.

$$\begin{array}{rrcl} (\mathit{Assertion}) & \mathtt{p} & \in & \mathit{State} \rightarrow \mathit{Prop} \\ (\mathit{CdSpec}) & \theta & ::= & \mathtt{p} \\ (\mathit{Interp.}) & [\![\mathtt{p}]\!] & \triangleq & \lambda\Psi.\lambda\mathbb{S}.\mathtt{p}\,\mathbb{S} \end{array}$$

XCAP and its extension [28] for weak memory update can be embedded into CAP0 too if we use formulae of type (*State* → *PropX*) to customize the θ in CAP0, and use the interpretation in XCAP as our interpretation function. TAL [24] may also be embedded in CAP0 indirectly through XCAP, as shown by Ni and Shao [28].

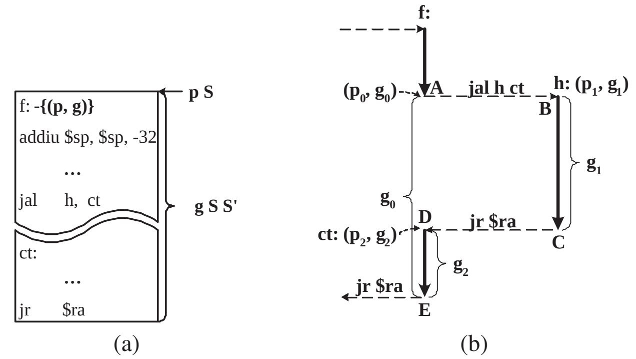


Figure 7. The Model for Function Call/Return in SCAP

4. SCAP for Function Call/Return

4.1 Stack-Based Reasoning for Function Call

We present SCAP as an instance of the CAP0 framework. The code specification θ in CAP0 is instantiated with the SCAP specification, which is defined as:

$$\begin{array}{rrcl} (\mathit{Assertion}) & \mathtt{p} & \in & \mathit{State} \rightarrow \mathit{Prop} \\ (\mathit{Guarantee}) & \mathtt{g} & \in & \mathit{State} \rightarrow \mathit{State} \rightarrow \mathit{Prop} \\ (\mathit{CdSpec}) & \theta & ::= & (\mathtt{p}, \mathtt{g}) \end{array}$$

A precondition for an instruction sequence contains a predicate $\mathtt{p}$ specifying the current state, and a *guarantee* $\mathtt{g}$ describing the relation between the current state and the state at the return point of the current function (if the function ever returns). Figure 7(a) shows the meaning of the specification $(\mathtt{p}, \mathtt{g})$ for the function `f` defined in Figure 1 (Section 1). Note that $\mathtt{g}$ may cover multiple instruction sequences. If a function has multiple return points, $\mathtt{g}$ governs all the traces from the current program point to any return point.

Figure 7(b) illustrates a function call to `h` from `f` at point `A`, with the return address `ct`. The specification of `h` is $(\mathtt{p}_1, \mathtt{g}_1)$. Specifications at `A` and `D` are $(\mathtt{p}_0, \mathtt{g}_0)$ and $(\mathtt{p}_2, \mathtt{g}_2)$ respectively, where $\mathtt{g}_0$ governs the code segment `A-E` and $\mathtt{g}_2$ governs `D-E`.

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

- the precondition of function `h` can be satisfied, *i.e.,*
$$\forall \mathbb{H}, \mathbb{R}. \mathtt{p}_0\;(\mathbb{H}, \mathbb{R}) \rightarrow \mathtt{p}_1\;(\mathbb{H}, \mathbb{R}\{\$\mathsf{ra} \rightsquigarrow \mathtt{ct}\});$$
- after `h` returns, `f` can resume its execution from point `D`, *i.e.,*
$$\forall \mathbb{H}, \mathbb{R}, \mathbb{S}'. \mathtt{p}_0\;(\mathbb{H}, \mathbb{R}) \rightarrow \mathtt{g}_1\;(\mathbb{H}, \mathbb{R}\{\$\mathsf{ra} \rightsquigarrow \mathtt{ct}\})\;\mathbb{S}' \rightarrow \mathtt{p}_2\;\mathbb{S}';$$
- if the function `h` and the code segment `D-E` satisfy their specifications, the specification for `A-E` is satisfied, *i.e.,*
$$\begin{array}{l} \forall \mathbb{H}, \mathbb{R}, \mathbb{S}', \mathbb{S}''. \mathtt{p}_0\;(\mathbb{H}, \mathbb{R}) \rightarrow \\ \quad \mathtt{g}_1\;(\mathbb{H}, \mathbb{R}\{\$\mathsf{ra} \rightsquigarrow \mathtt{ct}\})\;\mathbb{S}' \rightarrow \mathtt{g}_2\;\mathbb{S}'\;\mathbb{S}'' \rightarrow \mathtt{g}_0\;(\mathbb{H}, \mathbb{R})\;\mathbb{S}''; \end{array}$$
- the function `h` must reinstate the return code pointer when it returns, *i.e.,* $\forall \mathbb{S}, \mathbb{S}'. \mathtt{g}_1\;\mathbb{S}\;\mathbb{S}' \rightarrow \mathbb{S}.\mathbb{R}(\$\mathsf{ra}) = \mathbb{S}'.\mathbb{R}(\$\mathsf{ra})$.

Above conditions are enforced by the CALL rule shown in Figure 8 (ignore the meaning of $[\![(\mathtt{p}, \mathtt{g})]\!]$ for the time being, which will be defined later).

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

- the remaining instruction sequence is well-formed with regard to the intermediate specification;
- $\mathtt{p}'$ is satisfied by the resulting state of $\mathtt{c}$; and

$$\frac{\begin{array}{l}\mathtt{f},\mathtt{f}_{ret}\in dom(\Psi_{\mathrm{L}}) \quad (\mathrm{p}',\mathrm{g}')=\Psi_{\mathrm{L}}(\mathtt{f}) \quad (\mathrm{p}'',\mathrm{g}'')=\Psi_{\mathrm{L}}(\mathtt{f}_{ret})\\ \forall\mathbb{H},\mathbb{R}.\mathrm{p}\ (\mathbb{H},\mathbb{R})\rightarrow\mathrm{p}'\ (\mathbb{H},\mathbb{R}\{\$\mathsf{ra}\leadsto\mathtt{f}_{ret}\})\\ \forall\mathbb{H},\mathbb{R},\mathbb{S}'.\ \mathrm{p}\ (\mathbb{H},\mathbb{R})\rightarrow\mathrm{g}'\ (\mathbb{H},\mathbb{R}\{\$\mathsf{ra}\leadsto\mathtt{f}_{ret}\})\ \mathbb{S}'\rightarrow\\ \qquad\qquad(\mathrm{p}''\ \mathbb{S}'\wedge(\forall\mathbb{S}''.\ \mathrm{g}''\ \mathbb{S}'\ \mathbb{S}''\rightarrow\mathrm{g}\ (\mathbb{H},\mathbb{R})\ \mathbb{S}''))\\ \forall\mathbb{S},\mathbb{S}'.\mathrm{g}'\ \mathbb{S}\ \mathbb{S}'\rightarrow\mathbb{S}.\mathbb{R}(\$\mathsf{ra})=\mathbb{S}'.\mathbb{R}(\$\mathsf{ra})\end{array}}{\vdash\{\langle\llbracket(\mathrm{p},\mathrm{g})\rrbracket\rangle_{\Psi_{\mathrm{L}}}\}\,\mathsf{jal}\ \mathtt{f},\mathtt{f}_{ret}}\ (\textsc{call})$$

$$\frac{\begin{array}{l}\mathsf{c}\in\{\mathsf{addu},\mathsf{addiu},\mathsf{lw},\mathsf{subu},\mathsf{sw}\}\\ \vdash\{\langle\llbracket(\mathrm{p}',\mathrm{g}')\rrbracket\rangle_{\Psi_{\mathrm{L}}}\}\,\mathbb{I} \qquad \forall\mathbb{S}.\mathrm{p}\ \mathbb{S}\rightarrow\mathrm{p}'\ (\mathtt{Next}_{\mathsf{c}}(\mathbb{S}))\\ \forall\mathbb{S},\mathbb{S}'.\mathrm{p}\ \mathbb{S}\rightarrow\mathrm{g}'\ (\mathtt{Next}_{\mathsf{c}}(\mathbb{S}))\ \mathbb{S}'\rightarrow\mathrm{g}\ \mathbb{S}\ \mathbb{S}'\end{array}}{\vdash\{\langle\llbracket(\mathrm{p},\mathrm{g})\rrbracket\rangle_{\Psi_{\mathrm{L}}}\}\,\mathsf{c};\mathbb{I}}\ (\textsc{scap-seq})$$

$$\frac{\forall\mathbb{S}.\mathrm{p}\ \mathbb{S}\rightarrow\mathrm{g}\ \mathbb{S}\ \mathbb{S}}{\vdash\{\langle\llbracket(\mathrm{p},\mathrm{g})\rrbracket\rangle_{\Psi_{\mathrm{L}}}\}\,\mathsf{jr}\ \$\mathsf{ra}}\ (\textsc{ret})$$

$$\frac{\begin{array}{l}\mathtt{f}\in dom(\Psi_{\mathrm{L}}) \quad (\mathrm{p}',\mathrm{g}')=\Psi_{\mathrm{L}}(\mathtt{f})\\ \forall\mathbb{S}.\mathrm{p}\ \mathbb{S}\rightarrow\mathrm{p}'\ \mathbb{S} \quad \forall\mathbb{S},\mathbb{S}'.\mathrm{p}\ \mathbb{S}\rightarrow\mathrm{g}'\ \mathbb{S}\ \mathbb{S}'\rightarrow\mathrm{g}\ \mathbb{S}\ \mathbb{S}'\end{array}}{\vdash\{\langle\llbracket(\mathrm{p},\mathrm{g})\rrbracket\rangle_{\Psi_{\mathrm{L}}}\}\,\mathsf{j}\ \mathtt{f}}\ (\textsc{t-call})$$

$$\frac{\begin{array}{l}\mathtt{f}\in dom(\Psi_{\mathrm{L}}) \quad (\mathrm{p}'',\mathrm{g}'')=\Psi_{\mathrm{L}}(\mathtt{f}) \qquad \vdash\{\langle\llbracket(\mathrm{p}',\mathrm{g}')\rrbracket\rangle_{\Psi_{\mathrm{L}}}\}\,\mathbb{I}\\ \forall\mathbb{S}.\mathrm{p}\ \mathbb{S}\rightarrow\mathbb{S}.\mathbb{R}(\mathtt{r}_s)\leq 0\rightarrow(\mathrm{p}'\ \mathbb{S}\wedge(\forall\mathbb{S}'.\mathrm{g}'\ \mathbb{S}\ \mathbb{S}'\rightarrow\mathrm{g}\ \mathbb{S}\ \mathbb{S}'))\\ \forall\mathbb{S}.\mathrm{p}\ \mathbb{S}\rightarrow\mathbb{S}.\mathbb{R}(\mathtt{r}_s)>0\rightarrow(\mathrm{p}''\ \mathbb{S}\wedge(\forall\mathbb{S}'.\mathrm{g}''\ \mathbb{S}\ \mathbb{S}'\rightarrow\mathrm{g}\ \mathbb{S}\ \mathbb{S}'))\end{array}}{\vdash\{\langle\llbracket(\mathrm{p},\mathrm{g})\rrbracket\rangle_{\Psi_{\mathrm{L}}}\}\,\mathsf{bgtz}\ \mathtt{r}_s,\mathtt{f};\mathbb{I}}\ (\textsc{scap-bgtz})$$

Figure 8. SCAP Inference Rules

- if the remaining instruction sequence satisfies its guarantee g', the original instruction sequence satisfies g.

Suppose the state transition sequence made by the function is $(\mathbb{S}_0,\ldots,\mathbb{S}_n)$. To show that the function satisfies its guarantee g (*i.e.,* $\mathrm{g}\ \mathbb{S}_0\ \mathbb{S}_n$), we enforce the following chain of implication relations:

$$\mathrm{g}_n\ \mathbb{S}_{n-1}\ \mathbb{S}_n\rightarrow\mathrm{g}_{n-1}\ \mathbb{S}_{n-2}\ \mathbb{S}_n\rightarrow\ldots\rightarrow\mathrm{g}\ \mathbb{S}_0\ \mathbb{S}_n,$$

where each g_i is the intermediate specification used at each verification step. Each arrow on the chain is enforced by rules such as SCAP-SEQ. The head of the chain (*i.e.,* $\mathrm{g}_n\ \mathbb{S}_{n-1}\ \mathbb{S}_n$) is enforced by the RET rule (where $\mathbb{S}_{n-1}$ is the same with $\mathbb{S}_n$ since the jump instruction does not change the state), therefore we can finally reach the conclusion of $\mathrm{g}\ \mathbb{S}_0\ \mathbb{S}_n$.

SCAP also supports tail function call, where the callee reuses the caller's stack frame and the return code pointer. To make a tail function call, the caller just directly jumps to the callee's code. As shown in the T-CALL rule, we need to check that the guarantee of the callee matches the guarantee that remains to be fulfilled by the caller function.

Rules for branch instructions are straightforward. The SCAP-BGTZ rule is like a combination of the SCAP-SEQ rule and the T-CALL rule, since the execution may either fall through or jump to the target code label, depending on whether the condition holds.

Notice that all the code specifications Ψ_{L} used in SCAP rules are the *local* specifications for the current module. SCAP 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 function, we do not need any knowledge about the return address $ra in its precondition p. The RET rule for the instruction "jr $ra" does not have any constraint on $ra either. Examples in Section 4.3 illustrate how to write program specifications in SCAP.

4.2 The Stack Invariant

Figure 9 shows a snapshot of the stack of return continuations: the specification of the current function is $(\mathrm{p}_0,\mathrm{g}_0)$, which will return to its caller at the end; and the caller will return to the caller's caller... The return continuations in the dashed box compose a logical control stack.

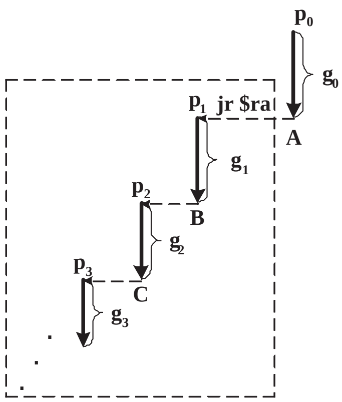


Figure 9. The Logical Control Stack

To establish the soundness of the SCAP inference rules, we need to ensure that when the current function returns at A, $ra contains a valid code pointer with the specification $(\mathrm{p}_1,\mathrm{g}_1)$, and p_1 is satisfied. Similarly we need to ensure that, at return points B and C, $ra contains valid code pointers with specifications $(\mathrm{p}_2,\mathrm{g}_2)$ and $(\mathrm{p}_3,\mathrm{g}_3)$ respectively, and that p_2 and p_3 are satisfied by then. Suppose the current state is $\mathbb{S}_0$ which satisfies p_0, above safety requirement can be formalized as follows:

$$\begin{array}{l}\mathrm{g}_0\ \mathbb{S}_0\ \mathbb{S}_1\rightarrow\\ \qquad\mathbb{S}_1.\mathbb{R}(\$\mathsf{ra})\in dom(\Psi)\wedge\Psi(\mathbb{S}_1.\mathbb{R}(\$\mathsf{ra}))=(\mathrm{p}_1,\mathrm{g}_1)\wedge\mathrm{p}_1\ \mathbb{S}_1;\\[1ex] \mathrm{g}_0\ \mathbb{S}_0\ \mathbb{S}_1\rightarrow\mathrm{g}_1\ \mathbb{S}_1\ \mathbb{S}_2\rightarrow\\ \qquad\mathbb{S}_2.\mathbb{R}(\$\mathsf{ra})\in dom(\Psi)\wedge\Psi(\mathbb{S}_2.\mathbb{R}(\$\mathsf{ra}))=(\mathrm{p}_2,\mathrm{g}_2)\wedge\mathrm{p}_2\ \mathbb{S}_2;\\[1ex] \mathrm{g}_0\ \mathbb{S}_0\ \mathbb{S}_1\rightarrow\mathrm{g}_1\ \mathbb{S}_1\ \mathbb{S}_2\rightarrow\mathrm{g}_2\ \mathbb{S}_2\ \mathbb{S}_3\rightarrow\\ \qquad\mathbb{S}_3.\mathbb{R}(\$\mathsf{ra})\in dom(\Psi)\wedge\Psi(\mathbb{S}_3.\mathbb{R}(\$\mathsf{ra}))=(\mathrm{p}_3,\mathrm{g}_3)\wedge\mathrm{p}_3\ \mathbb{S}_3;\\[1ex] \ldots\end{array}$$

where Ψ is the program specification, and each $\mathbb{S}_i$ is implicitly quantified by universal quantification.

Generalizing above safety requirement, we recursively define the "well-formed control stack with depth n" as follows:

$$\begin{array}{l}\mathsf{WFST}(0,\mathrm{g},\mathbb{S},\Psi)\triangleq\neg\exists\mathbb{S}'.\ \mathrm{g}\ \mathbb{S}\ \mathbb{S}'\\ \mathsf{WFST}(n,\mathrm{g},\mathbb{S},\Psi)\triangleq\\ \quad\forall\mathbb{S}'.\mathrm{g}\ \mathbb{S}\ \mathbb{S}'\rightarrow\mathbb{S}'.\mathbb{R}(\$\mathsf{ra})\in dom(\Psi)\wedge\mathrm{p}'\ \mathbb{S}'\wedge\mathsf{WFST}(n-1,\mathrm{g}',\mathbb{S}',\Psi)\\ \text{where }(\mathrm{p}',\mathrm{g}')=\Psi(\mathbb{S}'.\mathbb{R}(\$\mathsf{ra})).\end{array}$$

When the stack has depth 0, we are in the outermost function which has no return code pointer (the program either "halts" or enters an infinite loop). In this case, we simply require that there exist no $\mathbb{S}'$ at which the function can return, *i.e.,* $\neg\exists\mathbb{S}'.\ \mathrm{g}\ \mathbb{S}\ \mathbb{S}'$.

Then the stack invariant we need to enforce is 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* $\mathbb{S}$. The invariant is formally defined as:

$$\mathrm{p}\ \mathbb{S}\wedge\exists n.\mathsf{WFST}(n,\mathrm{g},\mathbb{S},\Psi).$$

With the stack invariant, we can "typecheck" the function return ("jr $ra") using the very simple RET rule without requiring that $ra contain a valid code pointer.

SCAP in the CAP Framework. We prove the soundness of SCAP by showing that SCAP inference rules are provable from the corresponding CAP0 rules, given a proper interpretation function for the SCAP specifications.

```
unsigned fact(unsigned n){
  return n ? n * fact(n - 1) : 1;
}

(a) regular recursive function

void fact(unsigned *r, unsigned n){
  if (n == 0) return;
  *r = *r * n;
  fact(r, n - 1);
}

(b) tail recursion with pointer arguments
```

Figure 10. Factorial Functions in C

$$\begin{array}{llll}
\mathsf{TRUE} & \triangleq \lambda\mathbb{S}.\mathsf{True} & \mathsf{NoG} & \triangleq \lambda\mathbb{S}.\lambda\mathbb{S}'.\mathsf{False} \\
\mathsf{Hnid}(\mathtt{ls}) & \triangleq \forall \mathtt{l} \notin \mathtt{ls}.[l] = [l]' & \mathsf{Rid}(\mathtt{rs}) & \triangleq \forall \mathtt{r} \in \mathtt{rs}, [\mathtt{r}] = [\mathtt{r}]' \\
\mathsf{Frm}[i] & \triangleq [[\$\mathsf{fp}] - i] & \mathsf{Frm}'[i] & \triangleq [[\$\mathsf{fp}] - i]'
\end{array}$$

$$\begin{array}{ll}
\mathrm{g}_{\mathrm{frm}} \triangleq & [\$\mathsf{sp}]' = [\$\mathsf{sp}] + 3 \wedge [\$\mathsf{fp}]' = \mathsf{Frm}[0] \\
& \wedge [\$\mathsf{ra}]' = \mathsf{Frm}[1] \wedge [\$\mathtt{s0}]' = \mathsf{Frm}[2]
\end{array}$$

Figure 11. Macros for SCAP Examples

In Section 4.1 we instantiated the CAP0 code specification θ with (p, g) in SCAP, without giving the interpretation function. Having defined the stack invariant, the interpretation of (p, g) is simply defined as the invariant:

$$[\![(\mathrm{p}, \mathrm{g})]\!] \triangleq \lambda\Psi.\lambda\mathbb{S}.\ \mathrm{p}\ \mathbb{S} \wedge \exists n.\mathsf{WFST}(n, \mathrm{g}, \mathbb{S}, \Psi).$$

The proof of SCAP inference rules as lemmas in CAP0 are presented in Appendix A and encoded in Coq [13].

4.3 Examples

In this section we show how SCAP can be used to support callee-save registers, optimizations for tail-recursions, and general pointer arguments in C.

Figure 10 shows two versions of the factorial function implemented in C. The first one is a regular recursive function, while the second one saves the intermediate result in the address passed as argument and makes a tail-recursive call.

The compiled assembly code of these two functions is shown in Figure 12 and 13. In both programs, the label `entry` points to the initial code segment where the function `fact` is called. SCAP specifications for the code heap are embedded in the code, enclosed by $-\{\}$. Figure 11 shows definitions of macros used in the code specifications. To simplify the presentation, we use $[\mathtt{r}]$ and $[\mathtt{l}]$ to represent values contained in the register `r` and memory location `l`. We also use primed representations $[\mathtt{r}]'$ and $[\mathtt{l}]'$ to represent values in the resulting state (the second argument) of a guarantee g. $\mathsf{Rid}(\mathtt{rs})$ means all the registers in `rs` are preserved by the function. $\mathsf{Hnid}(\mathtt{ls})$ means all memory cells *except* those with addresses in `ls` are preserved. $\mathsf{Frm}[i]$ represents the i^{th} word on the stack frame.

The specification at the entrance point (labeled by `prolog`) of the first function is given as $(\mathsf{TRUE}, \mathrm{g}_0)$ in Figure 12. The precondition defines no constraint on the value of $\$\mathsf{ra}$. The guarantee g_0 specifies the behavior of the function:

- the return value $[\$\mathtt{v0}]$ is the factorial of the argument $[\$\mathtt{a0}]$;
- callee-save registers are not updated; and
- the memory, other than the stack frames, are not updated.

If we use pre-/post-conditions in traditional Hoare-Logic to specify the function, we have to use auxiliary variables to specify the first point, and apply the Invariance Rule for the last two points. Using the guarantee g_0 they can be easily expressed.

$$\begin{array}{l}
\mathrm{g}_0 \triangleq [\$\mathtt{v0}]' = [\$\mathtt{a0}]! \wedge \mathsf{Rid}(\{\$\mathsf{gp}, \$\mathsf{sp}, \$\mathsf{fp}, \$\mathsf{ra}, \$\mathtt{s0}, \ldots, \$\mathtt{s7}\}) \\
\qquad \wedge \mathsf{Hnid}(\{([\$\mathsf{sp}] - 3 * [\$\mathtt{a0}] - 2), \ldots, [\$\mathsf{sp}]\}) \\
\mathrm{g}_1 \triangleq [\$\mathtt{v0}]' = [\$\mathtt{a0}]! \wedge \mathsf{Rid}(\{\$\mathsf{gp}, \$\mathtt{s1}, \ldots, \$\mathtt{s7}\}) \wedge \mathrm{g}_{\mathrm{frm}} \\
\qquad \wedge \mathsf{Hnid}(\{([\$\mathsf{sp}] - 3 * [\$\mathtt{a0}] + 1), \ldots, [\$\mathsf{sp}]\}) \\
\mathrm{g}_3 \triangleq ([\$\mathtt{v0}]' = [\$\mathtt{v0}] * [\$\mathtt{s0}]) \wedge \mathsf{Rid}(\{\$\mathsf{gp}, \$\mathtt{s1}, \ldots, \$\mathtt{s7}\}) \wedge \mathrm{g}_{\mathrm{frm}} \wedge \mathsf{Hnid}(\emptyset) \\
\mathrm{g}_4 \triangleq \mathsf{Rid}(\{\$\mathsf{gp}, \$\mathtt{v0}, \$\mathtt{s1}, \ldots, \$\mathtt{s7}\}) \wedge \mathrm{g}_{\mathrm{frm}} \wedge \mathsf{Hnid}(\emptyset)
\end{array}$$

```
prolog:   -{(TRUE, g0)}
          addiu  $sp, $sp, -3          ;allocate frame
          sw     $fp, 3($sp)           ;save old $fp
          addiu  $fp, $sp, 3           ;new $fp
          sw     $ra, -1($fp)          ;save return addr
          sw     $s0, -2($fp)          ;callee-save reg
          j      fact

fact:     -{(TRUE, g1)}
          bgtz   $a0, nonzero          ;n == 0
          addiu  $v0, $zero, 1         ;return 1
          j      epilog

nonzero:  -{([$a0] > 0, g1)}
          addiu  $s0, $a0, 0           ;save n
          addiu  $a0, $a0, -1          ;n--
          jal    prolog, cont          ;fact(n)

cont:     -{([$v0] = ([$s0] - 1)!, g3)}
          multu  $v0, $s0, $v0         ;return n*(n-1)!
          j      epilog

epilog:   -{(TRUE, g4)}
          lw     $s0, -2($fp)          ;restore $s0
          lw     $ra, -1($fp)          ;restore $ra
          lw     $fp, 0($fp)           ;restore $fp
          addiu  $sp, $sp, 3           ;restore $sp
          jr     $ra                   ;return

halt:     -{(TRUE, NoG)}
          j      halt

entry:    -{(TRUE, NoG)}
          addiu  $a0, $zero, 6         ;$a0 = 6
          jal    prolog, halt
```

Figure 12. SCAP Factorial Example

In the second implementation (in Figure 13), the caller passes the address of a *stack variable* to the function `fact`. The tail recursion is optimized by reusing the stack frame and making a direct jump. The precondition p_0 requires that stack variable be initialized to 1 and not be allocated on the unused stack space. The guarantee g_0 is similar to the one for the first version.

Malicious functions cannot be called. It is also interesting to see how malicious functions are rejected in SCAP. The following code shows a malicious function which disguises a function call of the virus code as a return (the more deceptive x86 version is "`push virus; ret`").

```
ld_vir: -{(p, g)}
        addiu $ra, $zero, virus   ;fake the ret addr
        jr    $ra                 ;disguised func. call
```

The function `ld_vir` can be verified in SCAP with a proper specification of (p, g) (*e.g.,* $(\mathsf{TRUE}, \lambda\mathbb{S}, \mathbb{S}'.\mathsf{True})$), because the SCAP RET rule does not check the return address in $\$\mathsf{ra}$. However, SCAP will reject any code trying to call `ld_vir`, because the `g` cannot satisfy the premises of the CALL rule.

5. Generalizations of SCAP

The methodology for SCAP scales well to multi-return function calls and weak continuations. In this section, we will generalize the SCAP system in two steps. By a simple relaxation of the CALL

$$
\begin{array}{l}
\mathrm{p}_0 \triangleq [[\$\mathsf{a0}]] = 1 \wedge [\$\mathsf{a0}] \notin \{([\$\mathsf{sp}]-2),\ldots,[\$\mathsf{sp}]\}\\
\mathrm{g}_0 \triangleq [[\$\mathsf{a0}]]' = [\$\mathtt{a1}]! \ \wedge\ \mathsf{Rid}(\{\$\mathsf{gp},\$\mathsf{sp},\$\mathsf{fp},\$\mathsf{ra},\$\mathsf{a0},\$\mathtt{s0},\ldots,\$\mathtt{s7}\})\\
\qquad \wedge \mathsf{Hnid}(\{[\$\mathsf{sp}]-2,\ldots,[\$\mathsf{sp}],[\$\mathsf{a0}]\})\\
\mathrm{p}_1 \triangleq [\$\mathsf{a0}] \notin \{([\$\mathsf{sp}]+1),\ldots,([\$\mathsf{sp}]+3)\}\\
\mathrm{g}_1 \triangleq ([[\$\mathsf{a0}]]' = [[\$\mathsf{a0}]] * [\$\mathtt{a1}]!) \wedge \mathsf{Rid}(\{\$\mathsf{gp},\$\mathsf{a0},\$\mathtt{s1},\ldots,\$\mathtt{s7}\})\\
\qquad \wedge \mathrm{g}_{\mathrm{frm}} \wedge \mathsf{Hnid}(\{[\$\mathsf{a0}]\})\\
\mathrm{g}_3 \triangleq \mathsf{Rid}(\{\$\mathsf{gp},\$\mathsf{a0},\$\mathtt{s1},\ldots,\$\mathtt{s7}\}) \wedge \mathrm{g}_{\mathrm{frm}} \wedge \mathsf{Hnid}(\emptyset)
\end{array}
$$

```
prolog:  -{(p0, g0)}
         addiu  $sp, $sp, -3         ;allocate frame
         sw     $fp, 3($sp)          ;save old $fp
         addiu  $fp, $sp, 3          ;new $fp
         sw     $ra, -1($fp)         ;save return addr
         sw     $s0, -2($fp)         ;callee-save reg
         j      fact

fact:    -{(p1, g1)}
         bgtz   $a1, nonzero         ;if n == 0 continue
         j      epilog

nonzero: -{(p1 ∧ [$a1] > 0, g1)}
         lw     $s0, 0($a0)          ;intermediate result
         multu  $s0, $s0, $a1        ;*r * n
         sw     $s0, 0($a0)          ;*r = *r * n
         addiu  $a1, $a1, -1         ;n--
         j      fact                 ;tail call

epilog:  -{(TRUE, g3)}
         lw     $s0, -2($fp)         ;restore $s0
         lw     $ra, -1($fp)         ;restore $ra
         lw     $fp, 0($fp)          ;restore $fp
         addiu  $sp, $sp, 3          ;restore $sp
         jr     $ra                  ;return

halt:    -{(TRUE, NoG)}
         j      halt

entry    -{(TRUE, NoG)}
         addiu  $sp, $sp, -1         ;allocate a slot
         addiu  $a0, $sp, 1          ;
         addiu  $s0, $zero, 1        ;$s0 = 1
         sw     $s0, 0($a0)          ;initialize
         addiu  $a1, $zero, 6        ;$a1 = 6
         jal    prolog, halt
```

Figure 13. SCAP Implementation of Tail Recursion

rule, we get system SCAP-I to support function calls with multiple return addresses (with the restriction that a function must return to its immediate caller). We can use SCAP-I to certify the stack-unwinding-based implementation for exceptions. We then combine the relaxed call rule with the support for tail function call and get a more general system, namely SCAP-II. SCAP-II can certify weak continuations, `setjmp/longjmp` and the full-blown MRLC [32].

5.1 SCAP-I

In SCAP, a function call is a $\mathsf{jal}\ \mathtt{f},\mathtt{f}_{ret}$ instruction (equivalent to $\mathsf{addiu}\ \$\mathsf{ra},\$\mathsf{zero},\mathtt{f}_{ret};\ \mathsf{j}\ \mathtt{f}$). The callee can only return to $\mathtt{f}_{ret}$, forced by the constraint $\forall \mathbb{S},\mathbb{S}'.\ \mathrm{g}'\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathbb{S}.\mathbb{R}(\$\mathsf{ra}) = \mathbb{S}'.\mathbb{R}(\$\mathsf{ra})$ in the CALL rule. To allow the callee to return to multiple locations, we simply remove that constraint. Also, since we no longer force a single return address, there is no need to set $ra at the call site, reducing the calling instruction to $\mathsf{j}\ \mathtt{f}$. The resulting rule becomes

$$
\frac{\begin{array}{l}
\forall \mathbb{S}.\ \mathrm{p}\ \mathbb{S} \rightarrow \mathrm{p}'\ \mathbb{S} \qquad\qquad \mathtt{f} \in dom(\Psi_{\mathrm{L}}) \quad (\mathrm{p}',\mathrm{g}') = \Psi_{\mathrm{L}}(\mathtt{f})\\
\ \forall \mathbb{S},\mathbb{S}'.\ \mathrm{p}\ \mathbb{S} \rightarrow \mathrm{g}'\ \mathbb{S}\ \mathbb{S}' \rightarrow\\
\qquad \mathbb{S}'.\mathbb{R}(\$\mathsf{ra}) \in dom(\Psi_{\mathrm{L}}) \wedge \mathrm{p}''\ \mathbb{S}' \wedge (\forall \mathbb{S}''.\ \mathrm{g}''\ \mathbb{S}'\ \mathbb{S}'' \rightarrow \mathrm{g}\ \mathbb{S}\ \mathbb{S}'')\\
\text{where } (\mathrm{p}'',\mathrm{g}'') = \Psi_{\mathrm{L}}(\mathbb{S}'.\mathbb{R}(\$\mathsf{ra}))
\end{array}}{\vdash \{\langle [\![(\mathrm{p},\mathrm{g})]\!] \rangle_{\Psi_{\mathrm{L}}}\}\, \mathsf{j}\ \mathtt{f}}
$$

(CALL-I)

This rules does not specify how the return address is going to be passed into the function. Instead, we only require that $ra contain a code pointer specified in Ψ_{L} at the return state $\mathbb{S}'$, which is provable based on the knowledge of p and g'. This allows SCAP-I to certify any convention for multi-return function call.

The CALL-I rule is also a lemma provable from the J rule of CAP0, using the same interpretation as the one for SCAP. The rest of SCAP-I inference rules are the same with those in SCAP. For instance, we can also use the T-CALL rule when we use "j f" to make a tail call.

5.2 SCAP-II for Weak Continuations

The weak continuation construct in C-- allows a function to return to any activation on the control stack. Since we use the guarantee g to represent the behavior of a function, we need to understand what happens to the intermediate activations on the stack that are "skipped": are their g's discarded or fulfilled?

In SCAP-II, we enforce that the callee must fulfill the remaining behavior of its caller before it can "skip" its caller and return to an activation deeper on the control stack. From the caller's point of view, it made a *tail call* to the callee.

$$
\frac{\begin{array}{l}
\forall \mathbb{S}.\ \mathrm{p}\ \mathbb{S} \rightarrow \mathrm{p}'\ \mathbb{S} \qquad\qquad \mathtt{f} \in dom(\Psi_{\mathrm{L}}) \quad (\mathrm{p}',\mathrm{g}') = \Psi_{\mathrm{L}}(\mathtt{f})\\
\forall \mathbb{S},\mathbb{S}'.\ \mathrm{p}\ \mathbb{S} \rightarrow \mathrm{g}'\ \mathbb{S}\ \mathbb{S}' \rightarrow\\
\qquad (\mathrm{g}\ \mathbb{S}\ \mathbb{S}' \vee\\
\qquad\ \ \mathbb{S}'.\mathbb{R}(\$\mathsf{ra}) \in dom(\Psi_{\mathrm{L}}) \wedge \mathrm{p}''\ \mathbb{S}' \wedge (\forall \mathbb{S}''.\ \mathrm{g}''\ \mathbb{S}'\ \mathbb{S}'' \rightarrow \mathrm{g}\ \mathbb{S}\ \mathbb{S}''))\\
\text{where } (\mathrm{p}'',\mathrm{g}'') = \Psi_{\mathrm{L}}(\mathbb{S}'.\mathbb{R}(\$\mathsf{ra}))
\end{array}}{\vdash \{\langle [\![(\mathrm{p},\mathrm{g})]\!] \rangle_{\Psi_{\mathrm{L}}}\}\, \mathsf{j}\ \mathtt{f}}
$$

(CALL-II)

In the CALL-II rule, we further relax the second premise of the CALL-I rule and provide an option of either returning to the return point of the caller or satisfying the caller's remaining g and therefore being able to return to the caller's caller. This requirement automatically forms arbitrary length chains that allow the return to go arbitrarily far in the stack. Also notice that the CALL-II rule is simply a combination of the CALL-I rule and the T-CALL in SCAP for tail call.

We also relax SCAP's definition of "well-formed stack" and allow dismissal of multiple stack frames at the return point. Using the new predicate WFST' defined below in the interpretation function for (p,g), we can derive the CALL-II rule as a lemma.

$$
\begin{array}{l}
\mathsf{WFST}'(0,\mathrm{g},\mathbb{S},\Psi) \triangleq\ \neg\exists \mathbb{S}'.\ \mathrm{g}\ \mathbb{S}\ \mathbb{S}'\\
\mathsf{WFST}'(n,\mathrm{g},\mathbb{S},\Psi) \triangleq\\
\quad \forall \mathbb{S}'.\mathrm{g}\ \mathbb{S}\ \mathbb{S}' \rightarrow\\
\qquad\quad \mathbb{S}'.\mathbb{R}(\$\mathsf{ra}) \in dom(\Psi) \wedge \mathrm{p}'\ \mathbb{S}' \wedge \exists m < n.\ \mathsf{WFST}'(m,\mathrm{g}',\mathbb{S}',\Psi)\\
\text{where } (\mathrm{p}',\mathrm{g}') = \Psi(\mathbb{S}'.\mathbb{R}(\$\mathsf{ra})).
\end{array}
$$

$$
[\![(\mathrm{p},\mathrm{g})]\!] \triangleq \lambda\Psi.\lambda\mathbb{S}.\mathrm{p}\ \mathbb{S}\ \wedge \exists n.\mathsf{WFST}'(n,\mathrm{g},\mathbb{S},\Psi)
$$

The rest of SCAP-II inference rules are the same with those in SCAP, which are all provable based on the new interpretation.

In the next section, we show how to use SCAP-II to reason about `setjmp/longjmp`. More examples with stack unwinding and stack cutting are presented in [13].

5.3 Example: setjmp/longjmp

`setjmp` and `longjmp` are two functions in the C library that are used to perform non-local jumps. They are used as follows: a `setjmp` is called to save the current state of the program into a data structure (*i.e.,* `jmp_buf`). That state contains the current stack pointer, all callee-save registers, the code pointer to the next instruction, and everything else prescribed by the architecture. Then when called with such a structure, `longjmp` restores every part of the saved state, and then jumps to the stored code pointer.

These functions in C are not considered safe. `setjmp` does not save closures, and thus the behavior of `longjmp` is undefined if the function calling the corresponding `setjmp` has returned. The

$$
\begin{aligned}
\mathsf{p_{buf}}(x) &\triangleq \{x \mapsto \_, \ldots, x+11 \mapsto \_\}\\
\mathsf{g_{buf}}(x) &\triangleq ([x]' = [\$\mathsf{s0}]) \wedge \ldots \wedge ([x+7]' = [\$\mathsf{s7}]) \wedge ([x+8]' = [\$\mathsf{fp}]) \wedge\\
&\qquad ([x+9]' = [\$\mathsf{sp}]) \wedge ([x+10]' = [\$\mathsf{gp}]) \wedge ([x+11]' = [\$\mathsf{ra}])\\
\mathsf{g'_{buf}}(x) &\triangleq ([\$\mathsf{s0}]' = [x]) \wedge \ldots \wedge ([\$\mathsf{s7}]' = [x+7]) \wedge ([\$\mathsf{fp}]' = [x+8]) \wedge\\
&\qquad ([\$\mathsf{sp}]' = [x+9]) \wedge ([\$\mathsf{gp}]' = [x+10]) \wedge ([\$\mathsf{ra}]' = [x+11])\\
\mathsf{p}_0 &\triangleq \mathsf{p_{buf}}([\$\mathsf{a0}]) * \mathsf{TRUE}\\
\mathsf{g}_0 &\triangleq ([\$\mathsf{v0}]' = 0) \wedge \mathsf{Rid}(\{\$\mathsf{ra}, \$\mathsf{sp}, \$\mathsf{fp}, \$\mathsf{gp}, \$\mathsf{a0}, \$\mathsf{s0}, \ldots, \$\mathsf{s7}\})\\
&\qquad \wedge \mathsf{g_{buf}}([\$\mathsf{a0}]) \wedge \mathsf{Hnid}(\{[\$\mathsf{a0}], \ldots, [\$\mathsf{a0}]+11\})\\
\mathsf{p}_1 &\triangleq (\mathsf{p_{buf}}([\$\mathsf{a0}]) * \mathsf{TRUE}) \wedge [\$\mathsf{a1}] \neq 0\\
\mathsf{g}_1 &\triangleq ([\$\mathsf{v0}]' = [\$\mathsf{a1}]) \wedge \mathsf{g'_{buf}}([\$\mathsf{a0}]) \wedge \mathsf{Hnid}(\emptyset)
\end{aligned}
$$

```
setjmp:   -{(p0, g0)}
          sw     $s0, 0($a0)      ;save callee-saves
          ...
          sw     $s7, 7($a0)
          sw     $fp, 8($a0)      ;frame pointer
          sw     $sp, 9($a0)      ;stack pointer
          sw     $gp, 10($a0)     ;global pointer
          sw     $ra, 11($a0)     ;old $ra
          addiu  $v0, $zero, 0    ;return value
          jr     $ra

longjmp:  -{(p1, g1)}
          lw     $s0, 0($a0)      ;restore callee-saves
          ...
          lw     $s7, 7($a0)
          lw     $fp, 8($a0)      ;restore $fp
          lw     $sp, 9($a0)      ;restore $sp
          lw     $gp, 10($a0)     ;restore $gp
          lw     $ra, 11($a0)     ;restore $ra
          addu   $v0, $zero, $a1  ;return value
          jr     $ra              ;jump to restored $ra
```

Figure 14. Implementation for `setjmp/longjmp`

```
jmp_buf env;    /* env is a global variable */

int rev(int x){                  void cmp0(int x){
  if (setjmp(env) == 0){           cmp1(x);
    cmp0(x);                     }
    return 0;
  }else{                         void cmp1(int x){
    return 1;                      if (x == 0)
  }                                  longjmp(env, 1);
}                                }
```

Figure 15. C Program Using `setjmp/longjmp`

```
rev:    -{(p0, g0)}
        addiu  $sp, $sp, -3          ;allocate frame
        sw     $fp, 3($sp)           ;save old $fp
        addiu  $fp, $sp, 3           ;new $fp
        sw     $ra, -1($fp)          ;save $ra
        sw     $a0, -2($fp)          ;save argument
        addiu  $a0, $zero, env       ;argument for setjmp
        addiu  $ra, $zero, ct1       ;set ret addr
        j      setjmp                ;setjmp(env)

ct1:    -{(p1, g1)}
        beq    $v0, $zero, ct2       ;if $v0 = 0 goto ct2
        addiu  $v0, $zero, 1
        j      epilog                ;return 1

ct2:    -{(p2, g2)}
        lw     $a0, -2($fp)          ;$a0 = x
        addiu  $ra, $zero, ct3       ;set ret addr
        j      cmp0                  ;cmp0(x)

ct3:    -{(p3, g3)}
        addiu  $v0, $zero, 0
        j      epilog                ;return 0

cmp0:   -{(p4, g4)}
        addiu  $sp, $sp, -3          ;allocate frame
        sw     $fp, 3($sp)           ;save old $fp
        addiu  $fp, $sp, 3           ;new $fp
        sw     $ra, -1($fp)          ;save $ra
        addiu  $ra, $zero, epilog    ;set ret addr
        j      cmp1                  ;cmp1(x)

cmp1:   -{(p5, g5)}
        beq    $a0, $zero, cutto     ;if ($a0==0) longjmp
        jr     $ra                   ;else return

cutto:  -{(p6, g6)}
        addiu  $a0, $zero, env       ;$a0 = env
        addiu  $a1, $zero, 1         ;$a1 = 1
        j      longjmp               ;longjmp(env, 1)

epilog: -{(p7, g7)}
        lw     $ra, -1($fp)          ;restore $ra
        lw     $fp, 0($fp)           ;restore $fp
        addiu  $sp, $sp, 3           ;restore $sp
        jr     $ra                   ;return
```

Figure 16. TM Code Using `setjmp/longjmp`

control flow abstraction provided by `setjmp/longjmp` is very similar to weak continuations and can be reasoned using SCAP-II.

The code in Figure 14 shows a simple implementation of `setjmp/longjmp` functions and their specifications. Here we borrow the separation logic [31] notation, where $\{l \mapsto n\}$ means the memory cell at address l contains value n, while $P * Q$ specifies two parts of memory which have disjoint domains and satisfy P and Q respectively. As shown in [39], separation logic primitives can be encoded in Coq and embedded in general predicates.

The precondition p_0 of `setjmp` simply requires that the argument `$a0` point to a `jmp_buf`. It guarantees (g_0) that the return value is 0; values of callee save registers, return code pointers and some other registers are not changed and they are saved in the `jmp_buf`; and data heap except the `jmp_buf` is not changed.

Precondition p_1 for `longjmp` is similar to p_0, with extra requirement that the second argument `$a1`, which will be the return value, cannot be 0. The guarantee g_1 says the function returns `$a1`, recovers register values saved in `jmp_buf` (including return code pointers and stack pointers), and does not change any part of the memory.

In Figure 15 we use a simple C program to illustrate the use of `setjmp/longjmp`. The function `rev` calls `setjmp` before it calls function `cmp0`, which in turn calls `cmp1`. If the argument is 0, the function `cmp1` skips its caller and jumps to the "`else`" branch of `rev` directly, otherwise it returns to its caller. So the behavior of `rev` is to return 1 if the argument is 0, and to return 0 otherwise.

Based on our specification of `setjmp/longjmp`, the compiled code of the C program can be certified using SCAP-II. The assembly code and specifications are presented in Figures 16 and 17. Here we reuse some macros defined previously in Figures 14 and 11.

The precondition p_0 for function `rev` requires that `env` point to a block of memory for the `jmp_buf`, and that there be disjoint memory space for stack frames; while the guarantee g_0 specifies the relationship between the argument $[\$\mathtt{a0}]$ and the return value $[\$\mathtt{v0}]'$, and the preservation of callee-save registers and memory (except for the space for stack frames). Specifications for function `cmp0` and `cmp1` are $(\mathsf{p}_4, \mathsf{g}_4)$ and $(\mathsf{p}_5, \mathsf{g}_5)$, respectively. Two different conditions are considered in g_4 and g_5, *i.e.,* conditions under which the functions return normally or cut the stack. Also, it is tricky to specify the code labeled by `ct1`, which may be reached after the return from either `setjmp` or `longjmp`. We need to consider both cases in the specification $(\mathsf{p}_1, \mathsf{g}_1)$.

$$
\begin{array}{lcl}
\mathsf{blk}(x,y) & \triangleq & \{x\mapsto \_, x+1\mapsto \_, \ldots, y\mapsto \_\} \\
\mathsf{p}'_{\mathrm{buf}}(x) & \triangleq & \{x \mapsto [\$\mathtt{s0}], \ldots, x+11 \mapsto \mathtt{ct1}\} \\
\mathsf{g}_{\mathrm{frm}} & \triangleq & ([\$\mathsf{sp}]' = [\$\mathsf{sp}]+3) \wedge ([\$\mathsf{fp}]' = \mathsf{Frm}[0]) \wedge ([\$\mathsf{ra}]' = \mathsf{Frm}[1]) \\
\mathsf{g}_{\mathrm{epi}} & \triangleq & \mathsf{Rid}(\{\$\mathsf{gp}, \$\mathtt{s0}, \ldots, \$\mathtt{s7}\}) \wedge \mathsf{g}_{\mathrm{frm}} \wedge \mathsf{Hnid}(\emptyset) \\
\mathsf{p}_0 & \triangleq & \mathsf{p}_{\mathrm{buf}}(\mathtt{env}) * \mathsf{blk}([\$\mathsf{sp}]-5, [\$\mathsf{sp}]) * \mathsf{TRUE} \\
\mathsf{g}_0 & \triangleq & ([\$\mathtt{a0}] = 0 \rightarrow [\$\mathtt{v0}]' = 1) \wedge ([\$\mathtt{a0}] \neq 0 \rightarrow [\$\mathtt{v0}]' = 0) \wedge \\
 & & \quad \wedge \mathsf{Rid}(\{\$\mathsf{gp}, \$\mathsf{sp}, \$\mathsf{fp}, \$\mathsf{ra}, \$\mathtt{s0}, \ldots, \$\mathtt{s7}\}) \\
 & & \quad \wedge \mathsf{Hnid}(\{[\$\mathsf{sp}]-5, \ldots, [\$\mathsf{sp}], \mathtt{env}, \ldots, \mathtt{env}+11\}) \\
\mathsf{p}_1 & \triangleq & \mathsf{p}'_{\mathrm{buf}}(\mathtt{env}) * \mathsf{blk}([\$\mathsf{sp}]-2, [\$\mathsf{sp}]) * \mathsf{TRUE} \\
\mathsf{g}_1 & \triangleq & ([\$\mathtt{v0}] = 0 \rightarrow \mathsf{g}_2) \wedge ([\$\mathtt{v0}] \neq 0 \rightarrow ([\$\mathtt{v0}]' = 1) \wedge \mathsf{g}_{\mathrm{epi}}) \\
\mathsf{p}_2 & \triangleq & \mathsf{p}_1 \\
\mathsf{g}_2 & \triangleq & (\mathsf{Frm}[2] = 0 \rightarrow [\$\mathtt{v0}]' = 1) \wedge (\mathsf{Frm}[2] \neq 0 \rightarrow [\$\mathtt{v0}]' = 0) \\
 & & \wedge \mathsf{g}'_{\mathrm{buf}}(\mathtt{env}) \wedge \mathsf{g}_{\mathrm{frm}} \wedge \mathsf{Hnid}(\{[\$\mathsf{sp}]-2, \ldots, [\$\mathsf{sp}]\}) \\
\mathsf{p}_3 & \triangleq & \mathsf{TRUE} \\
\mathsf{g}_3 & \triangleq & ([\$\mathtt{v0}]' = 0) \wedge \mathsf{g}_{\mathrm{epi}} \\
\mathsf{p}_4 & \triangleq & \mathsf{p}_{\mathrm{buf}}(\mathtt{env}) * \mathsf{blk}([\$\mathsf{sp}]-2, [\$\mathsf{sp}]) * \mathsf{TRUE} \\
\mathsf{g}_4 & \triangleq & ([\$\mathtt{a0}] = 0 \rightarrow \mathsf{g}'_{\mathrm{buf}}(\mathtt{env}) \wedge [\$\mathtt{v0}]' \neq 0) \\
 & & \wedge ([\$\mathtt{a0}] \neq 0 \rightarrow \mathsf{Rid}(\{\$\mathsf{gp}, \$\mathsf{sp}, \$\mathsf{fp}, \$\mathsf{ra}, \$\mathtt{s0}, \ldots, \$\mathtt{s7}\})) \\
 & & \wedge \mathsf{Hnid}(\{[\$\mathsf{sp}]-2, \ldots, [\$\mathsf{sp}]\}) \\
\mathsf{p}_5 & \triangleq & \mathsf{p}_{\mathrm{buf}}(\mathtt{env}) * \mathsf{TRUE} \\
\mathsf{g}_5 & \triangleq & ([\$\mathtt{a0}] = 0 \rightarrow \mathsf{g}'_{\mathrm{buf}}(\mathtt{env}) \wedge [\$\mathtt{v0}]' = 1) \\
 & & \wedge ([\$\mathtt{a0}] \neq 0 \rightarrow \mathsf{Rid}(\{\$\mathsf{gp}, \$\mathsf{sp}, \$\mathsf{fp}, \$\mathsf{ra}, \$\mathtt{s0}, \ldots, \$\mathtt{s7}\})) \wedge \mathsf{Hnid}(\emptyset) \\
\mathsf{p}_6 & \triangleq & \mathsf{p}_{\mathrm{buf}}(\mathtt{env}) * \mathsf{TRUE} \\
\mathsf{g}_6 & \triangleq & \mathsf{g}'_{\mathrm{buf}}(\mathtt{env}) \wedge [\$\mathtt{v0}]' = 1 \wedge \mathsf{Hnid}(\emptyset) \\
\mathsf{p}_7 & \triangleq & \mathsf{TRUE} \\
\mathsf{g}_7 & \triangleq & ([\$\mathtt{v0}]' = [\$\mathtt{v0}]) \wedge \mathsf{g}_{\mathrm{epi}}
\end{array}
$$

Figure 17. Specifications for Code in Figure 16

```
void f() {                    void h() {
  int i;                        int j;
  while(true) {                 while(true) {
    i++;                          j++;
    switch;                       switch;
  }                             }
}                             }
```

Figure 18. Higher-level Coroutine Pseudo code

6. Reasoning about Coroutines

Figure 18 shows a trivial higher-level program that uses coroutines. The purpose behind coroutines is to create code that actually consists of two mostly independent code executions that are sequential, with precisely defined switch points. Examples of such programs include producer/consumer programs and simple deterministic (round-robin) threads.

In this section, we present variations of SCAP to reason about coroutines. The system CAP-CR supports separate verification of coroutines without functions, while SCAP-CR can reason about arbitrary interleaving of coroutine switching and function call/return. Like SCAP, both systems can be embedded in CAP0.

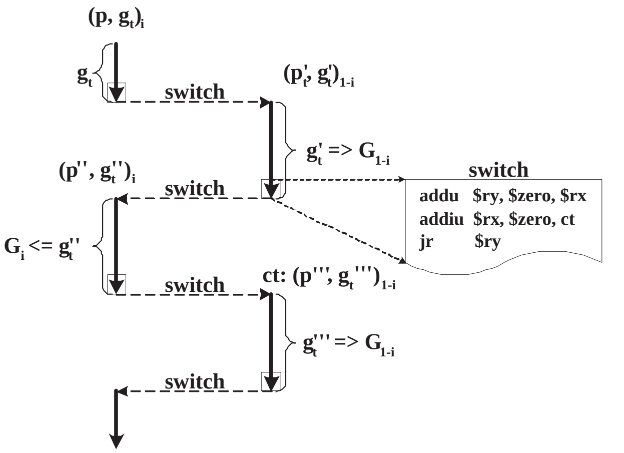
6.1 Coroutines without Function Call

We first work on a simplified model of coroutines, in which a coroutine does not make a function call. Figure 19 illustrates the execution of coroutines. To implement the switch from one routine to the other, we use two special registers ($rx and $ry) to hold the code pointers. `switch` is implemented as follows:

```
addu   $ry, $zero, $rx  ;set the target switch addr
addiu  $rx, $zero, ct   ;save the return addr
jr     $ry              ;jump to target address
```

where `ct` is the code label for the return continuation, as shown in Figure 19. In concrete implementations, $rx and $ry can be any two designated registers or even two memory cells.

Specifications θ for coroutine code are defined as follows:



$(\mathbf{p}, \mathbf{g}_t)_i$
$\mathbf{g}_t$
switch
$(\mathbf{p}'_t, \mathbf{g}'_t)_{1-i}$
$\mathbf{g}'_t \Rightarrow \mathbf{G}_{1-i}$
$(\mathbf{p}'', \mathbf{g}''_t)_i$
switch
$\mathbf{G}_i \Leftarrow \mathbf{g}''_t$
switch
ct: $(\mathbf{p}''', \mathbf{g}'''_t)_{1-i}$
$\mathbf{g}'''_t \Rightarrow \mathbf{G}_{1-i}$
switch
switch

```
addu   $ry, $zero, $rx
addiu  $rx, $zero, ct
jr     $ry
```

Figure 19. A Simplified Model for Coroutines

$$
\begin{array}{rrcl}
(\mathit{Assertion}) & \mathtt{p} & \in & \mathit{State} \rightarrow \mathit{Prop} \\
(\mathit{Guarantee}) & \mathsf{g}_t & \in & \mathit{State} \rightarrow \mathit{State} \rightarrow \mathit{Prop} \\
(\mathit{CdSpec}) & \theta & ::= & (\mathtt{p}, \mathsf{g}_t)_i
\end{array}
$$

where p specifies the current state, g_t describes the behavior of the code segment from the current program point to the switching point, and the index i ($0 \leq i \leq 1$) represent the i^{th} coroutine.

Different than function call, there is a new challenge for separate verification of coroutine code. Since the `switch` is done by an indirect jump, we do not know to which code segment of the target coroutine we are jumping to. However, we still need to ensure that the target coroutine will switch back to the right place with the expected state. To solve this problem, we use the rely-guarantee method [18] and assign a *global guarantee* $\mathbb{G}_i$ for the i^{th} coroutine. As shown in Figure 19, although we do not know whether we are jumping to the code segment with guarantee g'_t or the one with guarantee g'''_t, we can require that all code segments between two switch points in coroutine $1-i$ must satisfy $\mathbb{G}_{1-i}$, that is we require $\mathsf{g}'_t \Rightarrow \mathbb{G}$ and $\mathsf{g}'''_t \Rightarrow \mathbb{G}$. Here we use the short hand $\mathsf{g}_t \Rightarrow \mathbb{G}$ for $\forall \mathbb{S}, \mathbb{S}'.\ \mathsf{g}_t\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathbb{G}\ \mathbb{S}\ \mathbb{S}'$.

In summary, the specifications of coroutine i consist of $(\mathtt{p}, \mathsf{g}_t)_i$ pairs for each code segment, and a global guarantee $\mathbb{G}_i$ that specifies the common behavior for code segments between two consecutive switch points.

We use the following SWITCH rule to type check the indirect jump for switching.

$$
\frac{\begin{array}{l}
\forall \mathbb{S}. \mathsf{p}\ \mathbb{S} \rightarrow \mathsf{g}_t\ \mathbb{S}\ \mathbb{S} \\
\forall \mathbb{S}.\ \mathsf{p}\ \mathbb{S} \rightarrow \\
\qquad (\mathbb{S}.\mathbb{R}(\$\mathsf{rx}) \in \mathit{dom}(\Psi_{\mathrm{L}}) \wedge (\mathsf{g}'_t \Rightarrow \mathbb{G}_i) \wedge \\
\qquad\ (\forall \mathbb{S}'.\ \mathbb{G}_{1-i}\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathbb{S}'.\mathbb{R}(\$\mathsf{ry}) = \mathbb{S}.\mathbb{R}(\$\mathsf{rx}) \wedge \mathsf{p}'\ \mathbb{S}')) \\
\text{where } (\mathsf{p}', \mathsf{g}'_t)_i = \Psi_{\mathrm{L}}(\mathbb{S}.\mathbb{R}(\$\mathsf{rx}))
\end{array}}{\vdash \{\langle [\![(\mathsf{p}, \mathsf{g}_t)_i]\!] \rangle_{\Psi_{\mathrm{L}}}\} \mathsf{jr}\ \$\mathsf{ry}} \quad (\textsc{switch})
$$

The SWITCH rule is like a combination of the CALL rule and the RET rule of SCAP, because from coroutine i's point of view, the switch is like a function call, while for coroutine $(1-i)$ it is like a return. The first premise requires that the coroutine i must finish its guaranteed behavior before it switches to the coroutine $(1-i)$. The second premise requires that:

- $rx contain the return code pointer at the switch point, and the behavior starting from the return code pointer satisfy the global guarantee $\mathbb{G}_i$;
- at state $\mathbb{S}'$, the coroutine $(1-i)$ switch back to the expected place, *i.e.,* $\mathbb{S}'.\mathbb{R}(\$\mathsf{ry}) = \mathbb{S}.\mathbb{R}(\$\mathsf{rx})$; and

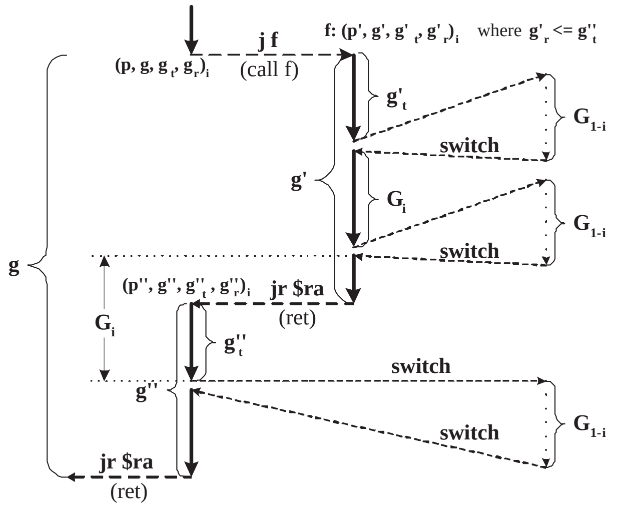


Figure 20. Model for Coroutines with Function Calls

- when the coroutine $(1-i)$ switches back, the state $\mathbb{S}'$ satisfy the precondition p' of the return continuation.

The rest inference rules, such as rules for sequential instructions and direct jumps, are the same with those in SCAP, except that the g's in SCAP has been replaced by g_t. To derive the SWITCH rule and other rules as lemmas in CAP0, we use the following interpretation for $(\mathrm{p}, \mathrm{g}_t)_i$.

$$
\begin{array}{l}
[\![(\mathrm{p}, \mathrm{g}_t)_i]\!] \triangleq \lambda \Psi. \lambda \mathbb{S}. \mathrm{p}\, \mathbb{S} \wedge \\
\quad\quad \forall \mathbb{S}'. \mathrm{g}_t\, \mathbb{S}\, \mathbb{S}' \rightarrow \mathbb{S}'.\mathbb{R}(\$\mathsf{ry}) \in dom(\Psi) \wedge \mathrm{p}'\, \mathbb{S}' \wedge (\mathrm{g}'_t \Rightarrow \mathbb{G}_{1-i}) \\
\text{where } (\mathrm{p}', \mathrm{g}'_t) = \Psi(\mathbb{S}'.\mathbb{R}(\$\mathsf{ra}))
\end{array}
$$

The interpretation function requires that:

- the current state be valid, *i.e.,* $\mathrm{p}\, \mathbb{S}$;
- at the switch point $\$\mathsf{ry}$ will be a valid code pointer in the coroutine $(1-i)$ with specification $(\mathrm{p}', \mathrm{g}'_t)_{1-i}$;
- the precondition of the label to which we are switching be satisfied, *i.e.,* $\mathrm{p}'\, \mathbb{S}'$; and
- the code to which we are switching will satisfy the coroutine $(1-i)$'s global guarantee, *i.e.,* $\mathrm{g}'_t \Rightarrow \mathbb{G}_{1-i}$.

Given the interpretation function, CAP-CR inference rules can be proved as lemmas in CAP0.

6.2 Coroutines with Function Calls

In the system CAP-CR, each coroutine does not make function calls, so we do not have to model stacks. Coroutines with function calls are trickier to verify because functions called by one coroutine may switch to another coroutine in the middle. It is harder to specify the behavior of functions.

In SCAP-CR, we instantiate the code specification θ in CAP0 as follows:

$$
\begin{array}{rlll}
(\mathit{Assertion}) & \mathrm{p} & \in & \mathit{State} \rightarrow \mathit{Prop} \\
(\mathit{Guarantee}) & \mathrm{g}, \mathrm{g}_t, \mathrm{g}_r & \in & \mathit{State} \rightarrow \mathit{State} \rightarrow \mathit{Prop} \\
(\mathit{CdSpec}) & \theta & ::= & (\mathrm{p}, \mathrm{g}, \mathrm{g}_t, \mathrm{g}_r)_i \ \text{ where } (0 \leq i \leq 1)
\end{array}
$$

As in SCAP, the function specification in SCAP-CR contains the specification p of the expected input and the behavior g of the function. Since a switch may occur within a function, we use g_t as in CAP-CR to specify the code segment from the current point to the next *switch* point, as shown in Figure 20. Also, because the *return* point and the *switch* point may not match, we use an extra guarantee g_r to specify the remaining state transition the current coroutine needs to make between the *return* point and the next *switch* point. Intuitively, g_r tells the caller of the current function what the caller needs to do after the function returns so that it can fulfill the guaranteed behavior before switching to another coroutine.[1] The switch operation is implemented in the same way shown in Section 6.1. For each coroutine, we also need a global guarantee $\mathbb{G}_i$ which captures the invariant of the code segments between any two consecutive switch points.

As shown in Figure 20, we need to enforce the following constraints for the function call in SCAP-CR.

- the behavior g'_t satisfies the caller's guaranteed behavior g_t from the calling point to the next switch point;
- when the callee returns, the caller's behavior g''_t from the return point to the next switch point satisfies the callee's expectation g'_r; and
- the constraints for return code pointers and function behaviors, as enforced in the CALL rule of SCAP-I.

These constraints are reflected in the following CR-CALL rule.

$$
\frac{\begin{array}{l}
(\mathrm{p}', \mathrm{g}', \mathrm{g}'_t, \mathrm{g}'_r)_i = \Psi_{\mathrm{L}}(\mathtt{f}) \\
\forall \mathbb{S}.\, \mathrm{p}\, \mathbb{S} \rightarrow \mathrm{p}'\, \mathbb{S} \qquad\qquad \forall \mathbb{S}, \mathbb{S}'.\, \mathrm{p}\, \mathbb{S} \rightarrow \mathrm{g}'_t\, \mathbb{S}\, \mathbb{S}' \rightarrow \mathrm{g}_t\, \mathbb{S}\, \mathbb{S}' \\
\forall \mathbb{S}, \mathbb{S}'.\, \mathrm{p}\, \mathbb{S} \rightarrow \mathrm{g}'\, \mathbb{S}\, \mathbb{S}' \rightarrow \\
\quad\quad (\mathbb{S}'.\mathbb{R}(\$\mathsf{ra}) \in dom(\Psi_{\mathrm{L}}) \wedge \mathrm{p}''\, \mathbb{S}' \wedge \\
\quad\quad\quad (\forall \mathbb{S}''.\, \mathrm{g}''\, \mathbb{S}'\, \mathbb{S}'' \rightarrow \mathrm{g}\, \mathbb{S}\, \mathbb{S}'') \wedge (\forall \mathbb{S}''.\, \mathrm{g}''_t\, \mathbb{S}'\, \mathbb{S}'' \rightarrow \mathrm{g}'_r\, \mathbb{S}'\, \mathbb{S}'')) \\
\text{where } (\mathrm{p}'', \mathrm{g}'', \mathrm{g}''_t, \mathrm{g}_r)_i = \Psi_{\mathrm{L}}(\mathbb{S}'.\mathbb{R}(\$\mathsf{ra}))
\end{array}}{\vdash \{\langle [\![(\mathrm{p}, \mathrm{g}, \mathrm{g}_t, \mathrm{g}_r)_i]\!] \rangle_{\Psi_{\mathrm{L}}}\}\, \mathsf{j}\ \mathtt{f}}
$$

(CR-CALL)

The return rule CR-RET is similar to the RET rule in SCAP, except that we also need to ensure that the expected caller's behavior g_r from the return point to the next switch point satisfies the guaranteed behavior g_t.

$$
\frac{\forall \mathbb{S}.\, \mathrm{p}\, \mathbb{S} \rightarrow \mathrm{g}\, \mathbb{S}\, \mathbb{S} \qquad\qquad \forall \mathbb{S}, \mathbb{S}'.\, \mathrm{p}\, \mathbb{S} \rightarrow \mathrm{g}_r\, \mathbb{S}\, \mathbb{S}' \rightarrow \mathrm{g}_t\, \mathbb{S}\, \mathbb{S}'}{\vdash \{\langle [\![(\mathrm{p}, \mathrm{g}, \mathrm{g}_t, \mathrm{g}_r)_i]\!] \rangle_{\Psi_{\mathrm{L}}}\}\, \mathsf{jr}\ \$\mathsf{ra}} \quad \text{(CR-RET)}
$$

The CR-SWITCH rule is similar to the SWITCH rule in CAP-CR, but we also need to enforce that the guaranteed behavior of the function is satisfied, *i.e.,* $\mathbb{G}_{1-i}\, \mathbb{S}\, \mathbb{S}' \rightarrow \mathrm{g}'\, \mathbb{S}'\, \mathbb{S}'' \rightarrow \mathrm{g}\, \mathbb{S}\, \mathbb{S}''$.

$$
\frac{\begin{array}{l}
\forall \mathbb{S}.\, \mathrm{p}\, \mathbb{S} \rightarrow \mathrm{g}_t\, \mathbb{S}\, \mathbb{S} \\
\forall \mathbb{S}.\, \mathrm{p}\, \mathbb{S} \rightarrow \\
\quad\quad (\mathbb{S}.\mathbb{R}(\$\mathsf{rx}) \in dom(\Psi_{\mathrm{L}}) \wedge (\mathrm{g}'_t \Rightarrow \mathbb{G}_i) \\
\quad\quad\quad (\forall \mathbb{S}'.\, \mathbb{G}_{1-i}\, \mathbb{S}\, \mathbb{S}' \rightarrow \\
\quad\quad\quad\quad \mathbb{S}'.\mathbb{R}(\$\mathsf{ry}) = \mathbb{S}.\mathbb{R}(\$\mathsf{rx}) \wedge \mathrm{p}'\, \mathbb{S}' \wedge (\forall \mathbb{S}''.\, \mathrm{g}'\, \mathbb{S}'\, \mathbb{S}'' \rightarrow \mathrm{g}\, \mathbb{S}\, \mathbb{S}''))) \\
\text{where } (\mathrm{p}', \mathrm{g}', \mathrm{g}'_t, \mathrm{g}_r)_i = \Psi_{\mathrm{L}}(\mathbb{S}.\mathbb{R}(\$\mathsf{rx}))
\end{array}}{\vdash \{\langle [\![(\mathrm{p}, \mathrm{g}, \mathrm{g}_t, \mathrm{g}_r)_i]\!] \rangle_{\Psi_{\mathrm{L}}}\}\, \mathsf{jr}\ \$\mathsf{ry}}
$$

(CR-SWITCH)

The following CR-SEQ rule is straightforward, which is simply a combination of the SEQ rules in SCAP and CAP-CR.

$$
\frac{\begin{array}{l}
\vdash \{\langle [\![(\mathrm{p}', \mathrm{g}', \mathrm{g}'_t, \mathrm{g}_r)_i]\!] \rangle_{\Psi_{\mathrm{L}}}\}\, \mathbb{I} \qquad\qquad \forall \mathbb{S}.\, \mathrm{p}\, \mathbb{S} \rightarrow \mathrm{p}'\, (\mathtt{Next}_{\mathtt{c}}(\mathbb{S})) \\
\forall \mathbb{S}, \mathbb{S}'. \mathrm{p}\, \mathbb{S} \rightarrow \\
\quad\quad (\mathrm{g}'\, (\mathtt{Next}_{\mathtt{c}}(\mathbb{S}))\, \mathbb{S}' \rightarrow \mathrm{g}\, \mathbb{S}\, \mathbb{S}') \wedge (\mathrm{g}'_t\, (\mathtt{Next}_{\mathtt{c}}(\mathbb{S}))\, \mathbb{S}' \rightarrow \mathrm{g}_t\, \mathbb{S}\, \mathbb{S}')
\end{array}}{\vdash \{\langle [\![(\mathrm{p}, \mathrm{g}, \mathrm{g}_t, \mathrm{g}_r)_i]\!] \rangle_{\Psi_{\mathrm{L}}}\}\, \mathtt{c}; \mathbb{I}}
$$

(CR-SEQ)

In SCAP-CR, we need to enforce the invariant on two well-formed control stacks, as we did in SCAP. The interpretation function for the specification $(\mathrm{p}, \mathrm{g}, \mathrm{g}_t, \mathrm{g}_r)_i$ is defined in Figure 21. The predicate WFCR ensures that:

- there is a well formed control stack for the current coroutine;
- at the switch point, $\$\mathsf{ry}$ contains a valid code pointer;

[1] We may not need g_r if we require that the global guarantee $\mathbb{G}$ be a transitive relation, *i.e.,* $\forall \mathbb{S}, \mathbb{S}', \mathbb{S}''.\ \mathbb{G}\, \mathbb{S}\, \mathbb{S}' \wedge \mathbb{G}\, \mathbb{S}'\, \mathbb{S}'' \rightarrow \mathbb{G}\, \mathbb{S}\, \mathbb{S}''$. Although reasonable in a non-deterministic concurrent setting, this constraint on $\mathbb{G}$ is too restrictive for coroutines. We decide to present SCAP-CR in the most general setting and use an extra g_r to link the caller and callee.

$$[\![(\mathsf{p},\mathsf{g},\mathsf{g}_t,\mathsf{g}_r)_i]\!] \triangleq \lambda\Psi.\lambda\mathbb{S}.\ \mathsf{p}\,\mathbb{S} \wedge \mathsf{WFCR}(i,\mathsf{g},\mathsf{g}_t,\mathsf{g}_r,\mathbb{S},\Psi)$$

$$\begin{array}{l}
\mathsf{WFCR}(i,\mathsf{g},\mathsf{g}_t,\mathsf{g}_r,\mathbb{S},\Psi) \triangleq \\
\quad \exists m.\mathsf{WFCRST}(m,\mathsf{g},\mathsf{g}_r,\mathbb{S},\Psi) \wedge \\
\quad (\forall \mathbb{S}'.\mathsf{g}_t\,\mathbb{S}\,\mathbb{S}' \rightarrow \mathbb{S}'.\mathbb{R}(\$\mathsf{ry}) \in dom(\Psi) \wedge (\mathsf{g}_t' \Rightarrow \mathbb{G}_{1-i}) \wedge \mathsf{p}'\,\mathbb{S}' \wedge \\
\qquad\qquad\qquad\qquad \exists n.\mathsf{WFCRST}(n,\mathsf{g}',\mathsf{g}_r',\mathbb{S}',\Psi)) \\
\text{where } (\mathsf{p}',\mathsf{g}',\mathsf{g}_t',\mathsf{g}_r')_{1-i} = \Psi(\mathbb{S}'.\mathbb{R}(\$\mathsf{ry}))
\end{array}$$

$$\begin{array}{l}
\mathsf{WFCRST}(0,\mathsf{g},\mathsf{g}_r,\mathbb{S},\Psi) \triangleq \neg\exists\mathbb{S}'.\ \mathsf{g}\,\mathbb{S}\,\mathbb{S}' \\
\mathsf{WFCRST}(n,\mathsf{g},\mathsf{g}_r,\mathbb{S},\Psi) \triangleq \\
\quad \forall\mathbb{S}'.\ \mathsf{g}\,\mathbb{S}\,\mathbb{S}' \rightarrow \\
\qquad \mathbb{S}'.\mathbb{R}(\$\mathsf{ra}) \in dom(\Psi) \wedge \mathsf{p}'\,\mathbb{S}' \wedge (\mathsf{g}_t' \Rightarrow \mathsf{g}_r) \wedge \\
\qquad\qquad\qquad\qquad \mathsf{WFCRST}(n-1,\mathsf{g}',\mathsf{g}_r',\mathbb{S}',\Psi) \\
\text{where } (\mathsf{p}',\mathsf{g}',\mathsf{g}_t',\mathsf{g}_r')_i = \Psi(\mathbb{S}'.\mathbb{R}(\$\mathsf{ra}))
\end{array}$$

Figure 21. The Interpretation Function for SCAP-CR

- the precondition of the $ry to which we are switching is satisfied at the switch point, *i.e.,* $\mathsf{p}'\,\mathbb{S}'$;
- the code to which we are switching will satisfy the coroutine $(1-i)$'s global guarantee, *i.e.,* $\mathsf{g}_t' \Rightarrow \mathbb{G}_{1-i}$; and
- at the switch point, there is a well-formed control stack in the coroutine $(1-i)$.

The definition of the well-formed control stack is similar to the definition of WFST in SCAP, except we also need to ensure that the caller's behavior from the return point to the next switch point actually satisfies the callee's expected behavior, *i.e.,* $\mathsf{g}_t' \Rightarrow \mathsf{g}_r$.

As usual, inference rules in SCAP-CR are provable as CAP0 lemmas based on this interpretation function.

7. Other Extensions and Implementation

Our methodology for reasoning about stack-based control abstractions can also be easily applied to specify and reason about exception handling and threads. Due to space limitation, we only give a very brief overview of these extensions. Detailed systems are presented in our companion technical report [13].

Exception handling. Although expressive enough, SCAP-I and SCAP-II presented in Sec 5 are not convenient to use for reasoning about exceptions because of their low abstraction level. In the TR [13], we propose two higher-level systems EUCAP and ECAP to support stack unwinding and stack cutting. EUCAP and ECAP allow the programmer to specify the *normal* behavior and the *exceptional* behavior separately, which makes specification and reasoning easier than in SCAP-I and SCAP-II.

User threads and the thread library. We propose Foundational CCAP (or FCCAP) to certify both the user code and the implementation of `yield`. Unlike previous work for certifying concurrent assembly code where `yield` is a primitive pseudo instruction [40], FCCAP does not use any special instructions and treats `yield` as a function call to the certified thread library. Two different logics are used in FCCAP: we use the rely-guarantee method in CCAP [40] to certify the user level code, and use SCAP to certify the implementation of the "`yield`" function. FCCAP shows how to combine different PCC logics in the CAP0 framework.

We use the Coq proof assistant [35] and the underlying higher-order predicate logic for fully mechanized verification of assembly code. The syntax of the TM is encoded in Coq using inductive definitions. Operational semantics of TM and the inference rules of CAP0 are defined as inductive relations. The soundness of the CAP0 rules is formalized and proved in Coq.

Instead of defining the syntax and semantics of the assertion language (which is known as the deep embedding approach), we use CiC, the underlying higher-order logic in Coq, as our assertion language. This shallow embedding approach greatly reduces the work load of formulating our logic systems.

Our implementation includes around 370 lines of Coq encoding of TM and its operational semantics, 200 lines encoding of CAP0 rules, and 700 lines of Coq tactics for the soundness proof. We also encoded in Coq the definition of SCAP inference rules and their proofs as CAP0 lemmas, which consists of around 900 lines of Coq inductive definitions and tactics. We have written more than 10 thousand lines of Coq tactics to certify practical programs, including the `malloc/free` library which was first certified in the original CAP [39]. According to our experience, human smartness is required to come up with proper program specifications, the difficulty depending on the property one is interested in and the subtlety of algorithms. Given proper specifications, proof construction of assembly code is mostly routine work. Some premises of SCAP rules can be automatically derived after defining lemmas for common instructions. For generality, we intentionally avoid specifying the layout of the physical stack and calling convention in SCAP. The low abstraction level causes lengthy (but still straightforward) proof for instructions involving memory operations. The burden of the programmer can be reduced if we define higher-level lemmas for specific stack organization. We leave this as the future work.

8. More Related Work and Conclusion

Reasoning about Stacks and Exceptions. Continuing over the related work discussed in Section 2.1, STAL [23] and its variations [11, 36] support static type-checking of function call/return and stack unwinding, but they all treat return code pointers as first-class code pointers and stacks as "closures". Introducing a "ret" instruction [11] does not change this fact because there the typing rule for "ret" requires a valid code pointer on the top of the stack, which is very different from our SCAP RET rule. Impredicative polymorphism has to be used in these systems to abstract over unused portions of the stack (as a closure), even though only return addresses are stored on the stack. Using compound stacks, STAL can type-check exceptions, but this approach is rather limited. If multiple exception handlers defined at different depths of the stack are passed to the callee, the callee has to specify their order on the stack, which breaks modularity. This problem may be overcome by using intersection types [11], though it has never been shown. Moreover, there is no known work certifying `setjmp/longjmp` and weak continuations using these systems.

Also, unlike STAL, SCAP does not require any built-in stack structure in the target machine (TM), so it does not need two sets of instructions for heap and stack operations. As shown in Figure 13, SCAP can easily support general data pointers into the stack or heap, which are not supported in STAL. In addition, SCAP does not enforce any specific stack layout, therefore it can be used to support sequential stacks, linked stacks, and heap-allocated activation records.

Concurrently with our work, Benton [5] proposed a typed program logic for a stack-based abstract machine. His instruction sequence specification is similar to the g in SCAP. Typing rules in his system also look similar to SCAP rules. However, to protect return code pointers, Benton uses a higher-level abstract machine with separate data stack and control stack; the latter cannot be touched by regular instructions except call and ret. Benton also uses a pair of pre- and postcondition as the specification which requires complex formalization of auxiliary variables.

At higher-level, Berdine *et al.* [6] showed that function call and return, exceptions, goto statements and coroutines follow a discipline of linearly used continuations. The idea is formalized by typing continuation transformers as linear functions, but no verification logic was proposed for reasoning about programs. Following the producer/consumer model (in Figure 2), our reasoning has a flavor of linearity, but it is not clear how our work and linear continuation-passing relate to each other.

Walker *et al.* [1, 17] proposed logical approaches for stack typing. They used CPS to reason about function calls. Their work focused on memory management and alias reasoning, while in SCAP we left the stack layout unspecified. Although the higher-order predicate logic is general enough to specify memory properties, substructural logic provides much convenience for memory specification. Applying their work to provide lemmas for different stack layouts and calling conventions will be our future work.

Reasoning about First-Class Code Pointers. Ni and Shao [29] introduce a special syntax $\mathsf{cptr}(\mathtt{f},\mathtt{a})$ in their assertion language to certify first-class code pointers. To support first-class code pointers in SCAP, we can extend it in a similar way by using $\mathsf{cptr}(\mathtt{f},(\mathtt{p},\mathtt{g}))$, which means $\mathtt{f}$ is a function pointer with the specification $(\mathtt{p},\mathtt{g})$. However, as we mentioned before, return code pointers and exception handlers have subtly different invariants from general first-class code pointers. So even with the support of first-class code pointers, it is still desirable to *not* treat regular stack-based control abstractions as general code pointers. Embedding SCAP and its extensions into the CAP0 framework allows interoperability between SCAP and other systems. We can reason about function call/return, exception handling, and coroutine as before and then use $\mathsf{cptr}(\mathtt{f},(\mathtt{p},\mathtt{g}))$ to reason about unavoidable first-class code pointers. Another interesting observation is that some seemingly first-class code pointers, such as threads' return code pointers stored in the thread queue, can actually be reasoned using SCAP-based systems. We need more experience to fully explore the applicability and the limitations of SCAP.

State Relations as Program Specifications. SCAP is not the first to use relations between two states as program specifications. The rely-guarantee method [18], TLA [21], and VDM [19] all use state relations to specify programs. However, the guarantee $\mathtt{g}$ used in SCAP is different from those used in previous systems. Generalizing the idea of local guarantee [40], SCAP uses $\mathtt{g}$ to describe the obligation that the current function must fulfill before it can return, raise an exception, or switch to other coroutines and threads. Notice that at the beginning of a function, our $\mathtt{g}$ matches precisely the VDM postcondition, but intermediate $\mathtt{g}$'s used in our SCAP-SEQ rule differ from the intermediate postconditions used in the sequential decomposition rule in VDM: the second state specified in our $\mathtt{g}$'s always refers to the (same) state at the exit point. We use these intermediate $\mathtt{g}$'s to bridge the gap between the entry and exit points of functions—this is hard to achieve using VDM's post conditions.

Yu's pioneer work [41] on machine code verification can also support stack-based procedure call and return. His correctness theorem for each subroutine resembles our guarantee $\mathtt{g}$, but it requires auxiliary logical predicates counting the number of instructions executed between different program points. It is unclear whether their method can be extended to handle complex stack-based controls as discussed in our current paper.

Conclusion. We have proposed a new methodology for modular verification of assembly code with all kinds of stack-based control abstractions, including function call/return, tail call, weak continuation, `setjmp/longjmp`, stack cutting, stack unwinding, multi-return function call, coroutines, and thread context switch. For each control abstraction, we have formalized its invariants and showed how to certify its implementation. All reasoning systems are proposed as instances of the generic CAP0 framework, which allows programs certified in different PCC systems to be linked together. Our system is fully mechanized [13]: we give the complete soundness proof and a full verification of several examples in the Coq proof assistant [35].

Acknowledgments

We thank anonymous referees for suggestions and comments on an earlier version of this paper. This research is based on work supported in part by gifts from Intel and Microsoft, and NSF grants CCR-0208618 and CCR-0524545. Sen Xiang's research is supported in part by the National Natural Science Foundation of China under Grant No. 60473068. 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. Ahmed and D. Walker. The logical approach to stack typing. In *Proc. of the 2003 ACM SIGPLAN Int'l workshop on Types in Lang. Design and Impl.*, pages 74–85. ACM Press, 2003.

[2] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

[3] A. W. Appel. Foundational proof-carrying code. In *Symp. on Logic in Comp. Sci. (LICS'01)*, pages 247–258. IEEE Comp. Soc., June 2001.

[4] K. R. Apt. Ten years of Hoare's logic: A survey – part I. *ACM Trans. on Programming Languages and Systems*, 3(4):431–483, 1981.

[5] N. Benton. A typed, compositional logic for a stack-based abstract machine. In *Proc. Third Asian Symp. on Prog. Lang. and Sys., LNCS 3780*, pages 364–380. Springer-Verlag, November 2005.

[6] J. Berdine, P. O'hearn, U. Reddy, and H. Thielecke. Linear continuation-passing. *Higher Order Symbol. Comput.*, 15(2-3):181–208, 2002.

[7] D. Chase. Implementation of exception handling, Part I. *The Journal of C Language Translation*, 5(4):229–240, June 1994.

[8] W. D. Clinger. Proper tail recursion and space efficiency. In *Proc. 1997 ACM Conf. on Prog. Lang. Design and Impl.*, pages 174–185, New York, 1997. ACM Press.

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

[10] M. E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, July 1963.

[11] K. Crary. Toward a foundational typed assembly language. Technical Report CMU-CS-02-196, Carnegie Mellon University, School of Computer Science, Dec. 2002.

[12] S. J. Drew, J. Gough, and J. Ledermann. Implementing zero overhead exception handling. Technical Report 95-12, Faculty of Information Technology, Queensland U. of Technology, Brisbane, Australia, 1995.

[13] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. Technical Report YALEU/DCS/TR-1336 and Coq Implementation, Dept. of Computer Science, Yale University, New Haven, CT, Nov. 2005. `http://flint.cs.yale.edu/publications/sbca.html`.

[14] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proc. Seventeenth Annual IEEE Symposium on Logic In Computer Science (LICS'02)*, pages 89–100. IEEE Computer Society, July 2002.

[15] D. R. Hanson. *C Interface & Implementations*. Add. Wesley, 1997.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct. 1969.

[17] L. Jia, F. Spalding, D. Walker, and N. Glew. Certifying compilation for a language with stack allocation. In *Proc. 20th IEEE Symposium on Logic in Computer Science*, pages 407–416, June 2005.

[18] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. on Programming Languages and Systems*, 5(4):596–619, 1983.

[19] C. B. Jones. *Systematic software development using VDM*. Prentice Hall International (UK) Ltd., 1986.

[20] B. W. Kernighan and D. M. Ritchie. *The C Programming Language (Second Edition).* Prentice Hall, 1988.

[21] L. Lamport. The temporal logic of actions. *ACM Trans. on Programming Languages and Systems*, 16(3):872–923, May 1994.

[22] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (Second Edition).* Addison-Wesley, 1999.

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

[24] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proc. 25th ACM Symp. on Principles of Prog. Lang.*, pages 85–97. ACM Press, Jan. 1998.

[25] D. A. Naumann. Predicate transformer semantics of a higher-order imperative language with record subtyping. *Science of Computer Programming*, 41(1):1–51, 2001.

[26] G. Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon Univ., Sept. 1998.

[27] G. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proc. 1998 ACM Conf. on Prog. Lang. Design and Impl.*, pages 333–344, New York, 1998.

[28] Z. Ni and Z. Shao. A translation from typed assembly languages to certified assembly programming. Technical report, Dept. of Computer Science, Yale Univ., New Haven, CT, Nov. 2005. `http://flint.cs.yale.edu/flint/publications/talcap.html`.

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

[30] N. Ramsey and S. P. Jones. A single intermediate language that supports multiple implementations of exceptions. In *Proc. 2000 ACM Conf. on Prog. Lang. Design and Impl.*, pages 285–298, 2000.

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

[32] O. Shivers and D. Fisher. Multi-return function call. In *Proc. 2004 ACM SIGPLAN Int'l Conf. on Functional Prog.*, pages 79–89. ACM Press, Sept. 2004.

[33] R. Stata and M. Abadi. A type system for java bytecode subroutines. In *Proc. 25th ACM Symp. on Principles of Prog. Lang.*, pages 149–160. ACM Press, 1998.

[34] G. L. Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, MA, 1978.

[35] The Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.0, Oct. 2004.

[36] J. C. Vanderwaart and K. Crary. A typed interface for garbage collection. In *Proc. 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 109–122, 2003.

[37] D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[38] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[39] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. In *Proc. 2003 European Symposium on Programming, LNCS Vol. 2618*, pages 363–379, 2003.

[40] D. Yu and Z. Shao. Verification of safety properties for concurrent assembly code. In *Proc. 2004 ACM SIGPLAN Int'l Conf. on Functional Prog.*, pages 175–188, September 2004.

[41] Y. Yu. *Automated Proofs of Object Code For A Widely Used Microprocessor*. PhD thesis, University of Texas at Austin, 1992.

A. SCAP Rules as Lemmas

In this section, we show how SCAP inference rules can be derived as lemmas from corresponding CAP0 rules. We only show the proof of the most interesting rules, *i.e.,* the CALL and RET rules. Proof for the complete set of SCAP rules are formalized in the Coq proof assistant, which is available at [13].

Lemma A.1 (Stack Strengthen) For all n, g, g', $\mathbb{S}$, $\mathbb{S}'$ and Ψ, if

$$\mathsf{WFST}(n,\mathsf{g},\mathbb{S},\Psi) \text{ and } \forall\mathbb{S}''.\mathsf{g}'\ \mathbb{S}'\ \mathbb{S}'' \to \mathsf{g}\ \mathbb{S}\ \mathbb{S}'',$$

we have $\mathsf{WFST}(n,\mathsf{g}',\mathbb{S}',\Psi)$.

Proof. This trivially follows the definition of WFST. □

Lemma A.2 (Call) Suppose $\texttt{f},\texttt{f}_{ret} \in dom(\Psi_{\mathrm{L}})$, $(\mathsf{p}',\mathsf{g}') = \Psi_{\mathrm{L}}(\texttt{f})$ and $(\mathsf{p}'',\mathsf{g}'') = \Psi_{\mathrm{L}}(\texttt{f}_{ret})$. If

1. $\forall\mathbb{H},\mathbb{R}.\mathsf{p}\ (\mathbb{H},\mathbb{R}) \to \mathsf{p}'\ (\mathbb{H},\mathbb{R}\{\$\mathsf{ra}\leadsto\texttt{f}_{ret}\})$;
2. $\forall\mathbb{H},\mathbb{R},\mathbb{S}'.\ \mathsf{p}\ (\mathbb{H},\mathbb{R}) \to \mathsf{g}'\ (\mathbb{H},\mathbb{R}\{\$\mathsf{ra}\leadsto\texttt{f}_{ret}\})\ \mathbb{S}' \to$
   $(\mathsf{p}''\ \mathbb{S}' \wedge (\forall\mathbb{S}''.\ \mathsf{g}''\ \mathbb{S}'\ \mathbb{S}'' \to \mathsf{g}\ (\mathbb{H},\mathbb{R})\ \mathbb{S}''))$;
3. $\forall\mathbb{S},\mathbb{S}'.\mathsf{g}'\ \mathbb{S}\ \mathbb{S}' \to \mathbb{S}.\mathbb{R}(\$\mathsf{ra}) = \mathbb{S}'.\mathbb{R}(\$\mathsf{ra})$;

we have

$$\forall\Psi,\mathbb{H},\mathbb{R}.\langle\llbracket\,(\mathsf{p},\mathsf{g})\,\rrbracket\rangle_{\Psi_{\mathrm{L}}}\ \Psi\ (\mathbb{H},\mathbb{R}) \to \llbracket\,\Psi(\texttt{f})\,\rrbracket\ \Psi\ (\mathbb{H},\mathbb{R}\{\$\mathsf{ra}\leadsto\texttt{f}_{ret}\}).$$

(In short, the CALL rule can be derived from the JAL rule).

Proof. Unfolding the definition of the interpretation function, we know that, given

4. $\Psi_{\mathrm{L}} \subseteq \Psi$;
5. $\mathsf{p}\ (\mathbb{H},\mathbb{R})$;
6. $\mathsf{WFST}(n,\mathsf{g},(\mathbb{H},\mathbb{R}),\Psi)$;

we need to prove

a. $\mathsf{p}'\ (\mathbb{H},\mathbb{R}\{\$\mathsf{ra}\leadsto\texttt{f}_{ret}\})$; and

b. $\mathsf{WFST}(n+1,\mathsf{g}',(\mathbb{H},\mathbb{R}\{\$\mathsf{ra}\leadsto\texttt{f}_{ret}\}),\Psi)$;

The proof of a is trivial (by 1 and 5). We focus on the proof of b.

By 4 and the assumption, we know that $\texttt{f},\texttt{f}_{ret} \in dom(\Psi)$, $\Psi(\texttt{f}) = (\mathsf{p}',\mathsf{g}')$ and $\Psi(\texttt{f}_{ret}) = (\mathsf{p}'',\mathsf{g}'')$. For all $\mathbb{S}$, if $\mathsf{g}'\ (\mathbb{H},\mathbb{R}\{\$\mathsf{ra}\leadsto\texttt{f}_{ret}\})\ \mathbb{S}$,

- by 3 we know $\mathbb{S}.\mathbb{R}(\$\mathsf{ra}) = \texttt{f}_{ret}$, therefore $\mathbb{S}.\mathbb{R}(\$\mathsf{ra}) \in dom(\Psi)$;
- by 5 and 2 we know $\mathsf{p}''\ \mathbb{S}$;
- by 5, 2, 6, and Lemma A.1 we know $\mathsf{WFST}(n,\mathsf{g}'',\mathbb{S},\Psi)$.

Then, by the definition of WFST we get

$$\mathsf{WFST}(n+1,\mathsf{g}',(\mathbb{H},\mathbb{R}\{\$\mathsf{ra}\leadsto\texttt{f}_{ret}\}),\Psi).$$ □

Lemma A.3 (Return) If $\forall\mathbb{S}.\mathsf{p}\ \mathbb{S} \to \mathsf{g}\ \mathbb{S}\ \mathbb{S}$, then for all Ψ, $\mathbb{H}$ and $\mathbb{R}$, we have

$$\llbracket\,(\mathsf{p},\mathsf{g})\,\rrbracket\ \Psi\ (\mathbb{H},\mathbb{R}) \to \llbracket\,\Psi(\mathbb{R}(\$\mathsf{ra}))\,\rrbracket\ \Psi\ (\mathbb{H},\mathbb{R}).$$

That is, the RET rule can be derived from an instantiation of the JR rule, where $\texttt{r}_s$ is instantiated to \$ra.

Proof. Given $\llbracket\,(\mathsf{p},\mathsf{g})\,\rrbracket\ \Psi\ (\mathbb{H},\mathbb{R})$ and our assumption, we know that

1. $\mathsf{p}\ (\mathbb{H},\mathbb{R})$;
2. $\mathsf{g}\ (\mathbb{H},\mathbb{R})\ (\mathbb{H},\mathbb{R})$; and
3. $\mathsf{WFST}(n,\mathsf{g},(\mathbb{H},\mathbb{R}),\Psi)$ for some n.

By 2, 3 and the definition of WFST we know that $n > 0$. Therefore, according to the definition of WFST, we can prove

4. $\mathbb{R}(\$\mathsf{ra}) \in dom(\Psi)$;
5. $\mathsf{p}'\ (\mathbb{H},\mathbb{R})$;
6. $\mathsf{WFST}(n-1,\mathsf{g}',(\mathbb{H},\mathbb{R}),\Psi)$;

where $(\mathsf{p}',\mathsf{g}') = \Psi(\mathbb{R}(\$\mathsf{ra}))$. By the definition of the interpretation function, we know $\llbracket\,\Psi(\mathbb{R}(\$\mathsf{ra}))\,\rrbracket\ \Psi\ (\mathbb{H},\mathbb{R})$. □