Abstract

An Open Framework for Certified System Software

Xinyu Feng

2008

Certified software consists of a machine executable program plus a machine checkable proof showing that the software is free of bugs with respect to a particular specification. Constructing certified system software is an important step toward building a reliable and secure computing platform for future critical applications. In addition to the benefits from provably safe components, architectures of certified systems may also be simplified to achieve better efficiency. However, because system software consists of program modules that use many different computation features and span different abstraction levels, it is difficult to design a single type system or program logic to certify the whole system. As a result, significant amount of kernel code of today's operating systems has to be implemented in unsafe languages, despite recent progress on type-safe languages.

In this thesis, I develop a new methodology to solve this problem, which applies different verification systems to certify different program modules, and then links the certified modules in an open framework to compose the whole certified software package. Specifically, this thesis makes contributions in the following two aspects.

First, I develop new Hoare-style program logics to certify low-level programs with different features, such as sequential programs with stack-based control abstractions and multi-threaded programs with unbounded thread creations. A common scheme in the design of these logics is modularity. They all support modular verification in the sense that program modules, such as functions and threads, can be certified separately without looking into implementation details of others.

Second, I propose an open verification framework that enables interoperation between different verification systems (a.k.a. foreign systems). The framework is open in that it is

not designed with a priori knowledge of foreign systems. It is general enough to incorporate both existing verification systems and new program logics presented in this thesis. It also supports modularity and proof reuse. Modules can be certified separately without knowing about implementation details of other modules and about the verification systems in which other modules are certified. Soundness of the framework guarantees that specifications of modules are respected after linking.

An Open Framework for Certified System Software

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Xinyu Feng

Dissertation Director: Zhong Shao

May 2008

An Open Framework for Certified System Software

Contents

List of Figures

Acknowledgments

First and foremost, I would like to thank my advisor, Zhong Shao, for his encouragement, support and guidance during my graduate study. He led me over the initial hurdles affronting an apprentice, exposed me to the beauty of programming language theories, and taught me by example the skill of a first-class researcher. I am particularly grateful for his energy and contagious enthusiasm, which always made research a pleasure. This work would not have been possible without the tremendous effort that he put into guiding me.

I would like to thank my thesis readers, Paul Hudak, Hongwei Xi and Bratin Saha, for all the time spent reading my thesis. Their keen comments and suggestions helped a lot to improve the quality of the thesis.

I am very lucky to have worked with many talented colleagues in the FLINT group and the University of Science and Technology of China (USTC). Zhaozhong Ni, Alexander Vaynberg, Sen Xiang and Yu Guo have collaborated on the work on SCAP and OCAP, and helped with Coq implementations. Rodrigo Ferreira contributed ideas on the development of the SAGL logic, and helped with the Coq implementation and proofs for the CMAP logic and the SAGL logic. I also benefitted a lot from discussions with Andrew McCreight about the use of Coq tactics. Valery Trifonov and Hai Fang helped answer many questions on type theories.

Finally, I want to thank my parents for their constant love and support. My deepest thanks goes to my wife, Xiaokun, for her patience, understanding and encouragement, without which the thesis would be impossible.

[0]This research is based on work supported in part by gifts from Intel and Microsoft, and NSF grants CCR-0208618 and CCR-0524545. Any opinions, findings and conclusions contained in this document are those of the author and do not reflect the views of these agencies.

Chapter 1

Introduction

Type-safe languages, such as Java, C♯ and ML, have been great success to improve the reliability of software. Type systems in these high level languages guarantee that some classes of runtime errors never occur. However, the guarantee usually assumes the underlying computing platform, including operating system kernels, system libraries and runtime systems, is safe. For instance, ML programs need to call garbage collectors at run time. If there are bugs in the garbage collector, ML programs may crash even if they are well-typed with respect to the ML type system.

Traditionally, system software is implemented using "unsafe" languages, such as C, C++ and assembly languages. Although there have been efforts to re-implement systems using type safe languages [55, 44], significant amount of code still has to be implemented using these "unsafe" languages because of the low abstraction level and architecture-dependencies [1]. This thesis studies methodologies and logics to build formally certified system software, with rigorous guarantees of safety and correctness properties. In particular, I concentrate on challenges to modularly specifying and certifying low-level system code.

[1]For instance, in the kernel of Singularity, 17% of files contain unsafe C♯ code and 5% of files contain x86 or C++ code [56].

1.1 Certified Software

Foundational certified software package [6, 46] contains machine code and mechanized mathematical proof showing that the code satisfies its specifications, which may specify safety, security or other interesting properties of the code. Given such a software package, we can use a proof checker to automatically check the validity of the proof with respect to the specifications and the machine code. If the proof is valid, we can predict the behavior of the code based on the specifications.

There are several key elements required to build certified software:

- A formal model of the hardware. To prove the execution of software satisfies the specifications, we need to first formalize how it is executed, which requires formalization of the hardware. This thesis concentrates on software reliability. We trust the correctness of hardware implementation and only formalize the hardware/software interface. Specifically, we need to model the set of valid machine instructions, machine states at the instruction set architecture (ISA) level, and how machine states are updated by executing each instruction (*i.e.,* the operational semantics of instructions). Given such a formalization, software is just a list of binary encoding of machine instructions and constants. The behavior of the software is rigorously defined.

- A mechanized meta-logic. We need the logic to specify interesting properties of the machine code, and to develop proofs showing that the code satisfies the specification. The logic needs to be expressive so that we can express everything a programmer wants to say about his or her programs. It must be consistent in the sense that a proposition and its negation cannot be proved in the logic at the same time. Also, it needs to be mechanized so that we can get machine-checkable proofs.

- Specifications. The specifications describe properties of the machine code, which may be memory safety, partial correctness, termination properties, resources usages or real-time properties. In this thesis, we will concentrate on the techniques to certify safety properties, and we call the specifications of safety properties *safety policies*. The safety policies are formally defined in our mechanized meta-logic.

- Certified package. The certified package contains the machine code and the proof. It is the ultimate product we want from the framework. The proof needs to show the code satisfies the safety policy. It is developed in our mechanized meta-logic and is machine-checkable.

- Proof checker. The proof checker is a program that takes the certified package and the specifications as inputs, and generates an output of "yes" or "no", *i.e.,* whether the proof is valid or not in the logic.

In the framework for foundational certified software, we do not need to trust the soundness of specific techniques that we use to certify the code, *e.g.,* type systems or Hoare logics, as long as the technique can finally give us a proof showing that the program satisfy the specification. However, there are several things we have to trust, which form our *trusted computing base* (TCB):

- We need to trust the correctness of our formalization of the hardware, and the formal specifications of the properties we want to certify. These are the starting points where we bridge the informal real world with the formal world in computer, and we can never formally prove the correctness of the formalization because one side is informal. To validate our formalization, we can either let experts to review our formalization, or test the validity in practice.

- We need to trust the consistency of our meta-logic to bootstrap. This means we should use a logic whose consistency has been proved and widely accepted.

- We need to trust the soundness of our proof checker with respect to the logic: if the proof checker says "yes", the proof must be valid in the logic. Based on the Curry-Howard isomorphism [54] applied in most logical frameworks, the problem of proof checking is reduced to type checking, which is a fairly standard and simple process and the implementation can be reviewed and tested by human being.

In this thesis, I would base my work on a MIPS-style machine, but the framework and logics presented here are independent with any specific hardware architectures. I use the Calculus of Inductive Constructions (CiC) [83, 24] as the meta-logic, which is mechanized in the Coq proof assistant [24]. CiC corresponds to the higher order logic via Curry-Howard isomorphism and serves as a very expressive logic to formalize our machine models and safety policies. Coq, the tactics-based proof assistant, also generates explicit proof terms in a semi-automatic way. The type checker of CiC implemented in Coq is used as a proof checker.

1.2 Challenges for Certifying System Software

Constructing certified system software is an important step toward building a reliable and secure computing platform for future critical applications. In addition to the benefits from provably safe components, architectures of certified systems may also be simplified to achieve better performance [4]. However, there are substantial challenges for building fully certified system software, because of the complexity and low abstraction levels. System software consists of program modules that use many language features and span different abstraction levels, including the lowest architecture-dependent level. To certify the whole software system, we need to address the following problems:

- Building formal models of language features. Specifications of safety and correctness properties of software involve formalization of language features. For instance, we may need to say in our specifications that "the code will never dereference a dangling pointer", "the target address of the jump instruction actually points to some code" or "there is no interference between threads". To allow such specifications, we need to give a formal model of "memory pointers", "code pointers" and "threads", *etc.*

- Modular development of proofs. To make the technique to scale, we need to develop composable program logics to support modular development of proofs. We want to be able to certify program modules separately without knowing details about the context where they are used other than an abstract specification of the interface. Updating one program module would not affect the proof for other modules as long as the specification for the updated module remains unchanged. This also supports the reuse of certified program modules and helps achieve the goal of “certify once, use everywhere”, which is especially important for system level software, a common platform shared by user applications.

- Certifying and linking code at different abstraction levels. A complete software system, *e.g.*, an OS kernel, consists of modules at different abstraction levels. We need to certify all these system modules and to link them together to get a fully certified system. On the other hand, the system level code would be used by user applications, which are at higher abstraction levels. Therefore, we also need to make sure our certified system software can be linked with user applications.

  The following two examples illustrate the difficulty to certify and link program modules at different abstraction levels. When we write multi-threaded programs, we usually do not need to consider the behavior of thread schedulers: threads are viewed as built-in language constructs for concurrency and the implementation of the scheduler is abstracted away. Most program logics for concurrency verification are based on this abstract model. However, in a real system, the scheduler might be implemented as sequential code manipulating data structures (thread control blocks) that stores execution contexts of threads. Existing program logics cannot be applied to certify both multi-threaded concurrent code and the scheduler code to get a fully certified system. The other example involves the type-safe mutator code and the underlying implementation of garbage collectors, as we mentioned at the beginning of this Chapter. It is extremely difficult to design a type system that can certify the low-level garbage collectors, but still preserves the abstractions for mutator code.

- Efficient production of proofs. Depending on the safety policy, certifying software usually requires explicit annotations and specifications of code blocks, and theorem proving to discharge verification conditions. The burden of writing specifications and proofs might be overwhelming if the safety policy is complex. However, it might be reduced by applying techniques for program analysis and automated theorem proving.

This thesis is focused on addressing the first three challenges. That is, I will concentrate on developing methodologies, frameworks and theories to certify low-level program modules. I leave the fourth problem as future work.

1.3 Contributions and Thesis Outline

Just like we never use one programming language to implement the whole software stack, from bootloaders and OS kernels to user applications, we should not try to find a single type system or program logic to certify the whole system, which would be extremely difficult. In this thesis, I propose a new methodology to build certified system software. I use different program logics to certify program modules with different language features and at different abstraction levels, and then link certified modules in an open framework to get a fully certified software package. Specifically, I try to address two important problems. First, I develop program logics allowing language features used in assembly level code to be specified and certified with higher level of abstractions. Second, I propose an open verification framework to enable interoperation between different verification systems. It supports the composition of a certified system from program modules specified and certified using different techniques. The framework is general enough to incorporate both existing verification systems and the new program logics proposed by me.

Before presenting my dissertation research, I first show the formal settings for this thesis in Chapter 2, including the meta-logic, the formal model of the machine, and a formalization of certified software package in the meta-logic. There is nothing particularly new in this chapter. Instead, it serves as a formal basis for the technical development in the following chapters.

In Chapter 3, I propose the open framework, OCAP. It is the first framework that systematically supports interoperability of different verification systems, *e.g.,* type systems or program logics. It lays a set of Hoare-style inference rules above the raw machine semantics, so that proofs can be constructed following these rules instead of directly using the mechanized meta-logic. Soundness of these rules are proved in the meta-logic with machine-checkable proof, therefore these rules are not trusted and do not increase the size of our TCB. OCAP supports modular verification. It is also extensible in the sense that, instead of being designed for interoperation of specific verification systems, OCAP is independent of specification languages and verification systems. New type systems or logics can be designed and integrated into the framework. To show the expressiveness of the framework, I also show in this Chapter the embedding of typed assembly language (TAL) [69] into OCAP.

In Chapter 4, I present a new program logic, SCAP, for modular verification of stack-based control abstractions at the assembly level. Because of the lack of structured language constructs, such as functions in high level languages, control flows at assembly level are usually reasoned about following the continuation passing style (CPS) [5]. Although general, the CPS-based reasoning treats all control abstractions as continuations, which is a very low-level language constructs. It also requires a semantic model of first-class code pointers, which is complex and difficult to use. I observe that stack based control abstractions at assembly level can be reasoned about in a more natural way without following CPS-based reasoning. The SCAP logic allows programmers to specify assembly code in terms of high level control abstractions, such as functions and exceptions. In this Chapter, I also give a thorough study of common stack-based control abstractions in the literatures. For each control abstraction, I formalize its invariants and show how to certify its implementation in SCAP. One more contribution made in this Chapter is an embedding of SCAP into the OCAP framework.

In Chapter 5, I show how to modularly certify concurrent code with dynamic thread creation. Yu and Shao [102] applied rely-guarantee based methodology [59] at assembly level to certify concurrent code in a modular way. Their thread model, however, is rather restrictive in that no threads can be created or terminated dynamically, which is an important feature widely supported and used in mainstream programming languages and operating systems. I extend their logic to support dynamic thread creation, which causes dynamic thread environment. In particular, I allow threads to have different assumptions and guarantees at different stages of their life time, so they can coexist with the dynamically changing thread environment. Some practical issues are also addressed, such as sharing of code between threads, argument passing at the time of thread creation, and the saving and restoring of thread-private data at context switches. These issues are important for realistic multi-threaded programming but as far as I know have never been discussed in previous work.

Given SCAP and CMAP logics for sequential and concurrent code, respectively, I show two applications of the OCAP framework in Chapter 6. In the first application, I show how to link user code in TAL with a simple certified memory management library. TAL only supports type-preserving memory update (a.k.a. weak memory update) and the free memory is invisible to TAL code. The memory management library is specified in SCAP, which supports reasoning about operations over free memory and still ensures that the invariants of TAL code is maintained. In the second application, I show how to construct foundational certified concurrent code *without* trusting the scheduler. The user thread code is certified using the rely-guarantee method, which supports thread-modular verification; the thread scheduler is certified as sequential code in SCAP. They are linked in OCAP to construct the foundational certified package. This is the first time that threads and the thread scheduler are certified together in the same framework.

In Chapter 7, I study the relationship between rely-guarantee reasoning and concurrent separation logic (CSL) [78, 14], both supporting modular verification of concurrent programs. I propose a new program logic, SAGL, to combine the merits of both sides. SAGL is more general than both rely-guarantee reasoning and CSL. I would show that, given a encoding of the invariants enforced in CSL as a special assumption and guarantee, we can formally prove that CSL can be derived as a special case of our SAGL logic. This result can also be viewed as an alternative proof of the soundness of CSL, which is simpler than existing proof by Brookes [14].

Chapter 2

Basic Settings

We first present the basic formal settings of our framework for certified software. In our framework, the machine and the operational semantics of machine instructions are formally defined in a mechanized meta-logic. Safety policies are also formalized in the meta-logic. The certified software is then a package containing the machine code and a formal proof showing that the code satisfies the safety policies.

2.1 The Mechanized Meta-Logic

We use the calculus of inductive constructions (CiC) [83] as our meta-logic, which is an extension of the calculus of constructions (CC) [25] with inductive definitions. CC corresponds to Church's higher-order predicate logic via the Curry-Howard isomorphism. CiC is supported by the Coq proof assistant [24], which we use to implement the results presented in this thesis.

$$\begin{array}{rl}(\textit{Term})\ A, B ::= & \mathsf{Set} \mid \mathsf{Prop} \mid \mathsf{Type} \mid X \mid \lambda X\!:\!A.B \mid A\ B \mid A \rightarrow B \mid \Pi X\!:\!A.\ B \\ & \mid \textit{inductive def.} \mid \ldots\end{array}$$

The syntax of commonly used CiC terms are shown above, where Set and Prop are the universes of objects and propositions, respectively. Type is the (stratified) universe of all terms. The term $\Pi X : A.\ B$ is the type of functions mapping values X with type A to

$$
\begin{array}{rcl}
(\textit{Program}) & \mathbb{P} & ::= (\mathbb{C}, \mathbb{S}, \mathsf{pc}) \\
(\textit{CodeHeap}) & \mathbb{C} & ::= \{\mathtt{f}_0 \rightsquigarrow \iota_0, \ldots, \mathtt{f}_n \rightsquigarrow \iota_n\} \\
(\textit{State}) & \mathbb{S} & ::= (\mathbb{H}, \mathbb{R}) \\
(\textit{Memory}) & \mathbb{H} & ::= \{\mathtt{l}_0 \rightsquigarrow \mathtt{w}_0, \ldots, \mathtt{l}_n \rightsquigarrow \mathtt{w}_n\} \\
(\textit{RegFile}) & \mathbb{R} & ::= \{\mathtt{r}_0 \rightsquigarrow \mathtt{w}_0, \ldots, \mathtt{r}_{31} \rightsquigarrow \mathtt{w}_{31}\} \\
(\textit{Register}) & \mathtt{r} & ::= \mathtt{r}_0 \mid \ldots \mid \mathtt{r}_{31} \\
(\textit{Labels}) & \mathtt{f}, \mathtt{l}, \mathsf{pc} & ::= n \ \ (\textit{nat nums}) \\
(\textit{Word}) & \mathtt{w} & ::= i \ \ (\textit{integers}) \\
(\textit{Instr}) & \iota & ::= \mathsf{addu}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \mid \mathsf{addiu}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{w} \mid \mathsf{beq}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f} \mid \mathsf{bgtz}\ \mathtt{r}_s, \mathtt{f} \\
 & & \quad \mid \mathsf{lw}\ \mathtt{r}_d, \mathtt{w}(\mathtt{r}_s) \mid \mathsf{subu}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \mid \mathsf{sw}\ \mathtt{r}_t, \mathtt{w}(\mathtt{r}_s) \mid \mathsf{j}\ \mathtt{f} \mid \mathsf{jal}\ \mathtt{f} \mid \mathsf{jr}\ \mathtt{r}_s \\
(\textit{InstrSeq}) & \mathbb{I} & ::= \iota \mid \iota; \mathbb{I}
\end{array}
$$

Figure 2.1: The Machine

type B, and X may be free in B. If X is not free in B, the term is abbreviated as $A \rightarrow B$. Via Curry-Howard isomorphism, we can view $\Pi X : A.\ B$ as a proposition with universal quantification over objects with type A (*i.e.*, $\forall X : A.\ B$) if B has kind Prop, while $A \rightarrow B$ represents logical implication if A and B both have kind Prop. Although we implement the certified framework in Coq, the thesis does not assume familiarity with Coq or CiC. In the rest of the thesis, we will not strictly follow the syntax of Coq, and use normal representations of logical connectives instead, which can be viewed as syntactic sugars and can be encoded in Coq.

2.2 Model of the Machine

In Figure 2.1 we show the formal modeling of a MIPS-style machine. A machine program $\mathbb{P}$ contains a code heap $\mathbb{C}$, an updatable program state $\mathbb{S}$ and a program counter pc pointing to the next instruction to execute. $\mathbb{C}$ is a partial mapping from code labels ($\mathtt{f}$) to instructions (ι). The program state consists of a data heap $\mathbb{H}$ and a register file $\mathbb{R}$. $\mathbb{H}$ is a partial mapping from memory locations ($\mathtt{l}$) to word values. The register file $\mathbb{R}$ is a total function from registers to word values.

There are 32 registers in the machine. Following the MIPS convention, Figure 2.2

\$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 2.2: Register Aliases and Usage

shows the register aliases and usage. Assembly code shown in this thesis will follow this convention.

To simplify the presentation, we do not model the von Neumann architecture since reasoning about self-modifying code is beyond the scope of this thesis. We model the code and data heaps separately and make the code heap read-only. This allows us to avoid formulating the encoding/decoding of instructions and the protection of code heaps, which is straightforward and is orthogonal to the focus of this thesis [17]. Also, we only show a small set of commonly used instructions, but they cover most of the interesting language features at the machine code level: direct/indirect jumps, branch instructions (also known as conditional jumps), memory access instructions and arithmetic instructions. Adding more instructions to the framework is straightforward.

In this thesis, we will use the "dot" notations to represent a component in a tuple. For instance, $\mathbb{P}.\mathsf{pc}$ means the program counter in the machine configuration $\mathbb{P}$. We also use $dom(F)$ to represent the domain of the function F, and use $F\{a \rightsquigarrow b\}$ to represent the update of the function F (total or partial) at a with new value b, which is formally defined below:

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

To lay some structure over the flat code heap $\mathbb{C}$, we use the instruction sequence $\mathbb{I}$

if $\iota =$	then $\mathsf{NextS}_{(\mathsf{pc},\,\iota)}(\mathbb{H},\mathbb{R}) =$	when
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 $\mathtt{r}_d, \mathtt{r}_s, \mathtt{w}$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{R}(\mathtt{r}_s)+\mathtt{w}\})$	
addiu $\mathtt{r}_d, \mathtt{r}_s, \mathtt{w}$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{R}(\mathtt{r}_s)+\mathtt{w}\})$	
subu $\mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{R}(\mathtt{r}_s)-\mathbb{R}(\mathtt{r}_t)\})$	
lw $\mathtt{r}_d, \mathtt{w}(\mathtt{r}_s)$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \leadsto \mathbb{H}(\mathbb{R}(\mathtt{r}_s)+\mathtt{w})\})$	$\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \in dom(\mathbb{H})$
sw $\mathtt{r}_t, \mathtt{w}(\mathtt{r}_s)$	$(\mathbb{H}\{\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \leadsto \mathbb{R}(\mathtt{r}_t)\}, \mathbb{R})$	$\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \in dom(\mathbb{H})$
jal $\mathtt{f}$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_{31} \leadsto \mathsf{pc}+1\})$	
other ι	$(\mathbb{H}, \mathbb{R})$	

if $\iota =$	then $\mathsf{NextPC}_{(\iota,\,\mathbb{S})}(\mathsf{pc}) =$	when
beq $\mathtt{r}_s, \mathtt{r}_t, \mathtt{f}$	$\mathtt{f}$	$\mathbb{S}.\mathbb{R}(\mathtt{r}_s) = \mathbb{S}.\mathbb{R}(\mathtt{r}_t)$
beq $\mathtt{r}_s, \mathtt{r}_t, \mathtt{f}$	$\mathsf{pc}+1$	$\mathbb{S}.\mathbb{R}(\mathtt{r}_s) \neq \mathbb{S}.\mathbb{R}(\mathtt{r}_t)$
bgtz $\mathtt{r}_s, \mathtt{f}$	$\mathtt{f}$	$\mathbb{S}.\mathbb{R}(\mathtt{r}_s) > 0$
bgtz $\mathtt{r}_s, \mathtt{f}$	$\mathsf{pc}+1$	$\mathbb{S}.\mathbb{R}(\mathtt{r}_s) \leq 0$
j $\mathtt{f}$	$\mathtt{f}$	
jal $\mathtt{f}$	$\mathtt{f}$	
jr $\mathtt{r}_s$	$\mathbb{S}.\mathbb{R}(\mathtt{r}_s)$	
other ι	$\mathsf{pc}+1$	

$$\frac{\iota = \mathbb{C}(\mathsf{pc}) \qquad \mathbb{S}' = \mathsf{NextS}_{(\mathsf{pc},\,\iota)}(\mathbb{S}) \qquad \mathsf{pc}' = \mathsf{NextPC}_{(\iota,\,\mathbb{S})}(\mathsf{pc})}{(\mathbb{C}, \mathbb{S}, \mathsf{pc}) \longmapsto (\mathbb{C}, \mathbb{S}', \mathsf{pc}')}$$

$$\frac{}{\mathbb{P} \longmapsto^0 \mathbb{P}} \qquad \frac{\mathbb{P} \longmapsto \mathbb{P}'' \qquad \mathbb{P}'' \longmapsto^k \mathbb{P}'}{\mathbb{P} \longmapsto^{k+1} \mathbb{P}'} \qquad \frac{\mathbb{P} \longmapsto^k \mathbb{P}'}{\mathbb{P} \longmapsto^* \mathbb{P}'}$$

Figure 2.3: Operational Semantics

to represent a basic code block. $\mathbb{C}[\mathtt{f}]$ extracts from $\mathbb{C}$ a basic block ending with a jump instruction.

$$\mathbb{C}[\mathtt{f}] = \begin{cases} \mathbb{C}(\mathtt{f}) & \text{if } \mathbb{C}(\mathtt{f}) = \mathsf{j}\ \mathtt{f}' \text{ or } \mathbb{C}(\mathtt{f}) = \mathsf{jr}\ \mathtt{r}_s \\ \mathbb{C}(\mathtt{f});\ \mathbb{I} & \text{if } \mathtt{f} \in dom(\mathbb{C}) \text{ and } \mathbb{I} = \mathbb{C}[\mathtt{f}+1] \\ \textit{undefined} & \text{otherwise} \end{cases}$$

We define the operational semantics of machine programs in Figure 2.3. One-step execution of a program is modeled as a transition relation $\mathbb{P} \longmapsto \mathbb{P}'$. The transition relation is defined based on the auxiliary functions $\mathsf{NextS}_{(\mathsf{pc},\ \iota)}(\_)$ and $\mathsf{NextPC}_{(\iota,\ \mathbb{S})}(\_)$. Given the current program state, the *partial* function $\mathsf{NextS}_{(\mathsf{pc},\ \iota)}(\_)$ defines the new program state after executing the the instruction ι at the current program counter pc. In most cases, the new program state is independent of pc, except that the $\mathsf{jal}\ \mathtt{f}$ instruction saves the next program counter $\mathsf{pc}+1$ into the register $\mathtt{r}_{31}$. For memory access instructions, the next state will be undefined if the target memory address is not in the domain of the current data heap. This corresponds to a "segmentation fault" when a program tries to access memory that it is not allowed to access. The function $\mathsf{NextPC}_{(\iota,\ \mathbb{S})}(\_)$ takes the current program counter as argument and gives the next program counter after executing ι at the current program state $\mathbb{S}$. By definition, we can see that the step transition $\mathbb{P} \longmapsto \mathbb{P}'$ is also a "partial" relation in the sense that, given a $\mathbb{P}$, there exists a $\mathbb{P}'$ satisfying $\mathbb{P} \longmapsto \mathbb{P}'$ only if:

- $\mathbb{P}.\mathsf{pc}$ is a valid address pointing to an instruction ι in the code heap $\mathbb{C}$; and
- given the program state in $\mathbb{P}$, the next state after executing ι is defined by the NextS function (*i.e.,* there is no "segmentation fault").

If there exists no such $\mathbb{P}'$, we say the execution of the program get stuck. One basic requirement of "safe" programs is that they will never get stuck. We will discuss more about safety policies in the next section.

As defined in Figure 2.3, $\mathbb{P} \longmapsto^k \mathbb{P}'$ means $\mathbb{P}$ reaches $\mathbb{P}'$ in k steps. The relation $\_ \longmapsto^* \_$ is the reflexive and transitive closure of the step transition relation. The following lemma shows the determinism of the operational semantics.

Lemma 2.1 If $\mathbb{P} \longmapsto^* \mathbb{P}'$ and $\mathbb{P} \longmapsto^* \mathbb{P}''$, then $\mathbb{P}' = \mathbb{P}''$.

Proof of the lemma is trivial because $\mathsf{NextS}_{(\mathsf{pc},\ \iota)}(\_)$ and $\mathsf{NextPC}_{(\iota,\ \mathbb{S})}(\_)$ are all defined as partial functions.

As an example, we show in Figure 2.4 a machine program, which is the MIPS code compiled from the following C program:

```
f:
    addiu  $sp, $sp, -32    ;allocate stack frame
    sw     $fp, 32($sp)     ;save old $fp
    addiu  $fp, $sp, 32     ;set new $fp
    sw     $ra, -4($fp)     ;save $ra
    jal    h                ;call h
ct: lw     $ra, -4($fp)     ;restore $ra
    lw     $fp, 0($fp)      ;restore $fp
    addiu  $sp, $sp, 32     ;deallocate frame
    jr     $ra              ;return

h:
    jr     $ra              ;return
```

Figure 2.4: Example of Machine Code: Function Call/Return

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

Before calling the 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`. Then the function `f` restores its own return code pointer and stack pointers, and jumps back to its caller's code.

2.3 Program Safety

Safety of the program means the execution of the program $\mathbb{P}$ satisfies some safety policy SP, which can be formalized as follows:

$$\forall \mathbb{P}'.\ (\mathbb{P} \longmapsto^* \mathbb{P}') \rightarrow \mathsf{SP}(\mathbb{P}').$$

Usually we use the invariant-based proof [63, 99, 7] to prove the program safety. We first define a program invariant INV, which is stronger than the safety policy. Then we prove:

1. *initial condition*: the initial program $\mathbb{P}_0$ satisfies INV, *i.e.,* $\mathsf{INV}(\mathbb{P}_0)$;

2. *progress*: for all $\mathbb{P}$, if $\mathsf{INV}(\mathbb{P})$, there exists $\mathbb{P}'$ such that $\mathbb{P} \longmapsto \mathbb{P}'$; and

3. *preservation*: for all $\mathbb{P}$ and $\mathbb{P}'$, if $\mathsf{INV}(\mathbb{P})$ and $\mathbb{P} \longmapsto \mathbb{P}'$, $\mathbb{P}'$ also satisfies the invariant, *i.e.*, $\mathsf{INV}(\mathbb{P}')$.

Using CiC as the meta-logic, we can support very general specifications of safety policies, which may range from simple memory safety (*i.e.*, programs never get stuck) to correctness of programs with respect to their specifications (a.k.a. partial correctness). For instance, we can ensure the memory safety by defining $\mathsf{SP}(\mathbb{P})$ as:

$$\mathsf{OneStep}(\mathbb{P}) \triangleq \exists \mathbb{P}'.\ \mathbb{P} \longmapsto \mathbb{P}'.$$

Such an SP can be trivially implied by the invariant-based proof methodology. On the other hand, suppose we have a program specification Ψ that defines the loop-invariants at some points of the program. We can define SP as:

$$\mathsf{SP}(\mathbb{P}) \triangleq \mathsf{OneStep}(\mathbb{P}) \wedge (\mathbb{P}.\mathsf{pc} \in dom(\Psi) \rightarrow \Psi(\mathbb{P}.\mathsf{pc})\ \mathbb{P}.\mathbb{S}),$$

which says that the program can make one step, and that if it reaches the point where a loop invariant is specified in Ψ, the loop invariant will hold over the program state. In this way, we capture the *partial correctness* of programs.

An FPCC package represented in the meta-logical framework is then a pair F containing the program and a proof showing that the program satisfies the safety policy [46]. Through Curry-Howard isomorphism, we know that

$$F \ \in\ \Sigma\, \mathbb{P} : \mathit{Program}.\ \forall \mathbb{P}'.\ (\mathbb{P} \longmapsto^{*} \mathbb{P}') \rightarrow \mathsf{SP}(\mathbb{P}')\,,$$

where $\Sigma x\!:\!A.P(x)$ represents the type of a dependent pair.

Chapter 3

An Open Framework — OCAP

As our formalization in Chapter 2 shows, we need to construct proofs for machine code to get the certified package. Although many type systems and program logics have been proposed in the last decades to certify properties of low-level code, they work at different abstraction levels, use different specification languages and axioms, and have different emphasis on computation features and properties. For instance, the typed assembly language (TAL) [69] uses types to specify assembly code and proves type safety. TAL code is at a higher abstraction level than machine code because it uses the abstract `malloc` instruction for memory allocation, while the actual implementation of `malloc` cannot be certified using TAL itself. In addition, TAL also assumes a trusted garbage collector in the run-time system. Yu and Shao's work on certifying concurrent assembly code [102] applies the rely-guarantee method to prove concurrency properties. They also use abstract machines with abstract instructions such as `yield`.

It is extremely difficult (if possible) to design a verification system supporting all the computation features. It may not be necessary to do so either because, fortunately, programmers do not use all these features at the same time. Instead, in each program module, only certain combination of limited features are used at certain abstraction level. If each module can be certified using existing systems (which is usually the case), it will be desirable to link each certified modules (code + proof) constructed in different verification

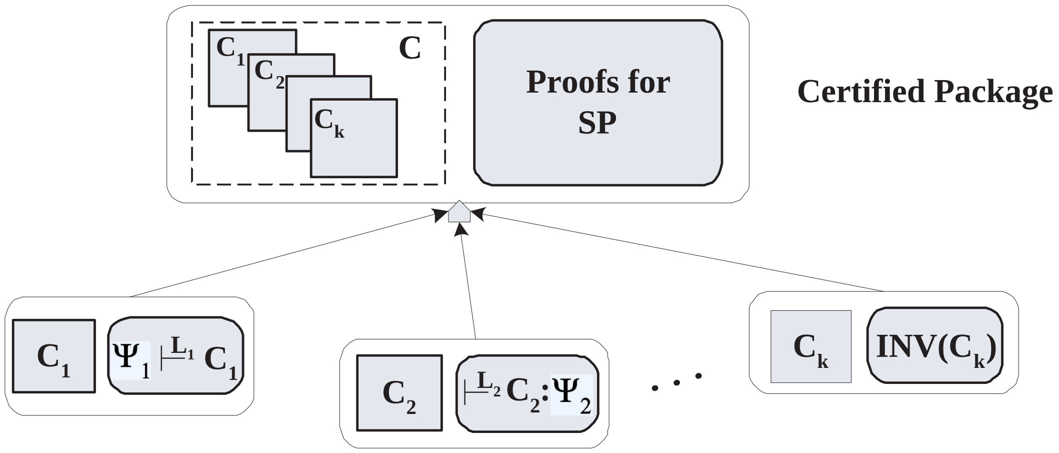


Figure 3.1: Building Certified Systems by Linking Certified Modules

systems to compose a completely certified system.

Suppose we want to build certified software package that contains the machine code C and a proof showing that C satisfies the safety policy SP, as shown in Fig. 3.1. The system C consists of code modules C_1, $C_2 \ldots C_k$. Some of them are system libraries or code of the run-time system, others are compiled from user modules. Each C_i is certified using certain verification system, with specifications about imported and exported interfaces. We want to reuse proofs for the modules and link them to generate the proof about the safety of the whole system. It is a challenging job because modules are certified separately using different specification languages and verification systems. When some of the modules (*e.g.*, system libraries) are specified and verified, the programmer may have no idea about the context where the code gets used and the verification system with which they will interoperate.

To compose the certified modules, we need an open framework which satisfies the following requirements:

- modularity: modules can be specified and certified separately; when they are linked the proof for each module can be reused;
- extensibility: instead of being designed specifically for certain combination of veri-

fication systems, the framework should be (mostly) independent with specification languages and verification systems (foreign systems hereafter); new systems can be designed and integrated into this framework;

- expressiveness: invariants enforced in foreign systems should be maintained in the framework, so that we can infer interesting properties about the composed program other than an overly-conservative safety policy.

Existing work on Foundational Proof-Carrying Code (FPCC) [7, 46, 28] only shows how to construct foundational proof for each specific verification system and does not support interoperation between systems, with the only exception of [45] which shows the interoperation between two specific systems (TAL and CAP). It is not trivial to make existing FPCC frameworks open either. The syntactic approach to FPCC [46, 28] simply formalizes the global syntactic soundness proof of verification systems in a mechanized meta-logic framework. It is unclear how different foreign verification systems can interoperate. The Princeton FPCC [7, 8, 94] uses a semantic approach. They construct FPCC for TAL by building semantic models for types. The semantic approach may potentially have nice support of interoperability as long as consistent models are built for foreign systems. However, sometimes it is hard to build and use semantic models. Most importantly, the step-indexed model [8] is defined specifically for type safety (*i.e.,* program never gets stuck). It is difficult to use the indexed model for embedded code pointers to support Hoare-style program logics, which usually certifies the partial correctness of programs with respect to program specifications.

People may argue that, since we use a mechanized meta-logic anyway, why bother to design an open framework? Users can bridge specific verification systems in an ad-hoc approach; and the proof checker of the meta-logic framework will ensure that the composed proof for the whole system is a valid proof about the safety property. There are several reasons that we prefer an open framework to the ad-hoc interoperation in the mechanized meta-logic. First, we need a uniform model of control flows. The control flow weaves program modules into a whole system. Without such a model, it is hard to formulate the composition of program modules. Second, enforcing principled interoperation between verification systems will help us infer properties of the whole program. Finally, a framework can certainly provide general guidance and reduce the workload of users.

In this Chapter, we propose an open framework, OCAP, for modular development of certified software. OCAP is the first framework for foundational certified software packages that systematically supports interoperation of different verification systems. It lays a set of Hoare-style inference rules above the raw machine semantics, so that proofs can be constructed following these rules instead of directly using the mechanized meta-logic. Soundness of these rules are proved in the meta-logic framework with machine-checkable proof, therefore these rules are not trusted. OCAP is modular, extensible and expressive, therefore it satisfies all the requirements mentioned above for an open framework. Specifically, the design of OCAP has the following important features:

- OCAP supports modular verification. When user code and runtime code are specified and certified, no knowledge about the other side is required. Modules certified in one verification system can be easily adapted to interoperate with other modules in a different system without redoing the proof.

- OCAP uses an extensible and heterogeneous program specification. Taking advantage of Coq's support of dependent types, specifications in foreign systems for modules can be easily incorporated as part of OCAP specifications. The heterogeneous program specification also allows OCAP to specify embedded code pointers, which enables OCAP's support for modularity.

- The assertions used in OCAP inference rules are expressive enough to specify invariants enforced in most type systems and program logics, such as memory safety, well-formedness of stacks, non-interference between concurrent threads, *etc.*. The soundness of OCAP ensures that these invariants are maintained when foreign systems are embedded in the framework.

The material presented in this chapter is based on the research I have done jointly with Zhaozhong Ni, Zhong Shao and Yu Guo [33]. In the rest part of this chapter, we first give an overview of previous work on certified assembly programming, and explain the challenges to design an open framework. Then we present the OCAP framework. At the end, we also show an embedding of TAL into OCAP. More applications of OCAP will be given in Chapters 4 and 6.

3.1 Overview of Certified Assembly Programming

Yu *et al.* first adapted Hoare logic to assembly level for Certified Assembly Programming (CAP) [101]. Here we give an brief overview of CAP and its extensions. Our OCAP framework is a generalization of previous work on CAP systems.

3.1.1 The CAP system

CAP is a Hoare-style program logic for proving partial correctness of assembly code. In addition to the basic safety, *i.e.,* "non-stuckness" of programs, it also guarantees that certified programs work correctly with respect to the user's specifications.

CAP expects a program specification Ψ which collects the loop invariants asserted for each basic code block. Each assertion $\mathtt{p}$ is a predicate over the program state, as shown below.

$$
\begin{aligned}
(\mathit{CHSpec})\ \ \Psi &\in \mathit{Labels} \rightharpoonup \mathit{StatePred}\\
(\mathit{StatePred})\ \ \mathtt{p} &\in \mathit{State} \to \mathsf{Prop}
\end{aligned}
$$

Here the program specification Ψ is a partial mapping from labels to assertions. The assertion $\mathtt{p}$ is defined directly in the meta-logic CiC as a function that takes a state as an argument and returns a proposition (recall that Prop is the universe of propositions in CiC, as shown in Section 2.1). Therefore $\mathtt{p}$ is a predicate over states. By using directly the meta-logic for specifications, we do not need to define the syntax and proof theory of the assertion language. This technique is also known as shallow embedding [62] of the

$\boxed{\Psi \vdash \mathbb{P}}$ **(*Well-formed program*)**

$$\frac{\Psi \vdash \mathbb{C}\!:\!\Psi \quad (\mathsf{p}\ \mathbb{S}) \quad \Psi \vdash \{\mathsf{p}\}\ \mathsf{pc} :\ \mathbb{C}[\mathsf{pc}]}{\Psi \vdash (\mathbb{C}, \mathbb{S}, \mathsf{pc})} \ (\textsc{prog})$$

$\boxed{\Psi \vdash \mathbb{C}\!:\!\Psi'}$ **(*Well-formed code heap*)**

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

$\boxed{\Psi \vdash \{\mathsf{p}\}\ \mathtt{f} :\ \mathbb{I}}$ **(*Well-formed instruction sequence*)**

$$\frac{\begin{array}{l}\iota \in \{\mathsf{addu}, \mathsf{addiu}, \mathsf{lw}, \mathsf{subu}, \mathsf{sw}\}\\ \Psi \vdash \{\mathsf{p}'\}\ \mathtt{f}{+}1 :\ \mathbb{I} \qquad \mathsf{p} \Rightarrow \mathsf{p}' \circ \mathsf{NextS}_{(\mathtt{f},\ \iota)}\end{array}}{\Psi \vdash \{\mathsf{p}\}\ \mathtt{f} :\ \iota;\ \mathbb{I}} \ (\textsc{seq})$$

$$\frac{\forall \mathbb{S}.\ \mathsf{p}\ \mathbb{S} \rightarrow \exists \mathsf{p}'.\ \mathsf{codeptr}(\mathbb{S}.\mathbb{R}(\mathtt{r}_s), \mathsf{p}')\ \Psi \wedge \mathsf{p}'\ \mathbb{S}}{\Psi \vdash \{\mathsf{p}\}\ \mathtt{f} :\ \mathsf{jr}\ \mathtt{r}_s} \ (\textsc{jr})$$

Figure 3.2: Selected CAP Rules

assertion languages in the meta-logical framework.

CAP inference rules. Figure 3.2 shows some selected inference rules of CAP. Using the invariant-based proof, CAP enforces the program invariant $\Psi \vdash \mathbb{P}$. As shown in the PROG rule, the invariant requires that:

- Ψ characterize the code heap $\mathbb{C}$ and guarantee the safe execution of $\mathbb{C}$, *i.e.,* $\Psi \vdash \mathbb{C}\!:\!\Psi$.
- There exist a precondition p for the current instruction sequence $\mathbb{C}[\mathsf{pc}]$ (recall our definition of $\mathbb{C}[\mathtt{f}]$ in Section 2.2). Given the knowledge Ψ about the complete code heap, the precondition p will guarantee the safe execution of $\mathbb{C}[\mathsf{pc}]$, *i.e.,* $\Psi \vdash \{\mathsf{p}\}\ \mathsf{pc} :\ \mathbb{C}[\mathsf{pc}]$.
- The current program state $\mathbb{S}$ satisfy p.

To certify a program, we only need to prove that the initial program $(\mathbb{C}, \mathbb{S}_0, \mathsf{pc}_0)$ satisfies the invariant, *i.e.,* $\Psi \vdash (\mathbb{C}, \mathbb{S}_0, \mathsf{pc}_0)$. As part of the soundness of the program logic, we can prove that our formulation of the invariant (based on Ψ) guarantees progress and preservation properties shown in Section 2.3.

The CDHP rule defines well-formedness of code heap, *i.e.*, $\Psi \vdash \mathbb{C} : \Psi'$. The rule says that it is safe to execute code in $\mathbb{C}$ if the loop invariant asserted at each label $\mathtt{f}$ in Ψ' guarantees the safe execution of the corresponding basic block $\mathbb{C}[\mathtt{f}]$, *i.e.*, $\Psi \vdash \{\Psi'(\mathtt{f})\}\, \mathtt{f} : \mathbb{C}[\mathtt{f}]$. The Ψ on the left hand side specifies the preconditions of code which may be reached from $\mathbb{C}[\mathtt{f}]$. In other words, Ψ specifies imported interfaces for each basic block in $\mathbb{C}$. Based on this definition, we can prove the following lemma.

Lemma 3.1 If $\Psi \vdash \mathbb{C} : \Psi$, then $dom(\Psi) \subseteq dom(\mathbb{C})$.

Rules for well-formed instruction sequences ensure that it is safe to execute the instruction sequence under certain precondition. For sequential instructions, the SEQ rule requires that the user find a precondition $\mathtt{p}'$ and prove that the remaining instruction sequence $\mathbb{I}$ is well-formed with respect to $\mathtt{p}'$. Also the user needs to prove that the precondition $\mathtt{p}'$ holds over the resulting state of ι. Here $\mathtt{p}' \circ \mathsf{NextS}_{(\mathsf{pc},\, \iota)}$ is the shorthand for

$$\lambda \mathbb{S}.\ \exists \mathbb{S}'.\ (\mathbb{S}' = \mathsf{NextS}_{(\mathsf{pc},\, \iota)}(\mathbb{S})) \wedge \mathtt{p}'\ \mathbb{S}'\,, \tag{3.1}$$

and $\mathtt{p} \Rightarrow \mathtt{p}'$ means logical implication between predicates:

$$\forall \mathbb{S}.\ \mathtt{p}\ \mathbb{S} \rightarrow \mathtt{p}'\ \mathbb{S}\,. \tag{3.2}$$

The premise $\mathtt{p} \Rightarrow \mathtt{p}' \circ \mathsf{NextS}_{(\mathsf{pc},\, \iota)}$ implies that $\mathtt{p}$ must ensure the safe execution of ι, since $\mathsf{NextS}_{(\mathsf{pc},\, \iota)}(\_)$ is a partial function. Usually $\mathtt{p}'$ can be the automatically derived strongest postcondition $\lambda \mathbb{S}.\ \exists \mathbb{S}_0 . \mathtt{p}\ \mathbb{S}_0 \wedge (\mathbb{S} = \mathsf{NextS}_{(\mathsf{pc},\, \iota)}(\mathbb{S}_0))$.

The JR rule essentially requires that the precondition for the target address hold at the time of jump. The proposition $\mathsf{codeptr}(\mathtt{f}, \mathtt{p})\ \Psi$ is defined as:

$$\mathsf{codeptr}(\mathtt{f}, \mathtt{p})\ \Psi \triangleq \mathtt{f} \in dom(\Psi) \wedge \Psi(\mathtt{f}) = \mathtt{p}\,.$$

Above definition also ensures that the target address is in the domain of the global code heap $\mathbb{C}$, following Lemma 3.1.

Soundness. The soundness of CAP ensures that well-formed programs never get stuck. It also guarantees that, after jump or branch instructions, the assertions specified at target addresses in Ψ hold over the corresponding program states. This shows the partial correctness of the program with respect to its specification Ψ. Yu *et al.* [101] exploited CAP's support of partial correctness to certify an implementation of malloc and free libraries.

The soundness of CAP is formalized in Theorem 3.2. Proof of the theorem follows the syntactic approach to proving type soundness [99], which requires the proof of progress and preservation properties. We will not show the proof in this thesis, which is similar to the soundness proof for OCAP given in Section 3.4.

Theorem 3.2 (CAP-Soundness)
If $\Psi \vdash \mathbb{P}$, where $\mathbb{P} = (\mathbb{C}, \mathbb{S}, \mathsf{pc})$, then for all n there exist $\mathbb{S}'$ and pc' such that
$\mathbb{P} \longmapsto^{n} (\mathbb{C}, \mathbb{S}', \mathsf{pc}')$, and

1. if $\mathbb{C}(\mathsf{pc}) = \mathsf{j}\ \mathtt{f}$, then $\Psi(\mathtt{f})\ \mathbb{S}'$;
2. if $\mathbb{C}(\mathsf{pc}) = \mathsf{jal}\ \mathtt{f}$, then $\Psi(\mathtt{f})\ \mathsf{NextS}_{(\mathsf{pc}',\ \mathsf{jal}\ \mathtt{f})}(\mathbb{S}')$;
3. if $\mathbb{C}(\mathsf{pc}) = \mathsf{jr}\ \mathtt{r}_s$, then $\Psi(\mathbb{S}.\mathbb{R}(\mathtt{r}_s))\ \mathbb{S}'$;
4. if $\mathbb{C}(\mathsf{pc}) = \mathsf{beq}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f}$ and $\mathbb{S}'.\mathbb{R}(\mathtt{r}_s) = \mathbb{S}'.\mathbb{R}(\mathtt{r}_t)$, then $\Psi(\mathtt{f})\ \mathbb{S}'$;
5. if $\mathbb{C}(\mathsf{pc}) = \mathsf{bgtz}\ \mathtt{r}_s, \mathtt{f}$ and $\mathbb{S}'.\mathbb{R}(\mathtt{r}_s) > 0$, then $\Psi(\mathtt{f})\ \mathbb{S}'$.

3.1.2 Specifications of embedded code pointers

CAP is a general framework for assembly code verification, but it does not support modularity very well, as pointed out by Ni and Shao [75]. The JR rule in CAP requires that the target address be a valid code pointer, which must be guaranteed by the precondition before the jump. The only way we can specify this is to list all the potential targets of the jump. For instance, the precondition of function h in Figure 2.4 needs to say that "\$ra contains the code label ct". This approach is not modular because, to specify a function, we need to know all the callers in advance. If there is another function that would also call h, we have to change the precondition of h, although it is safe to call h from any functions. The reason for this problem is because CAP's specification language (predicates over states) is not expressive enough to directly express $\mathsf{codeptr}(\mathtt{f},\mathtt{p})\ \Psi$, which requires the reference to Ψ. To specify a code pointer in CAP, we have to know the global Ψ in advance, which breaks modularity.

A quick attack to this problem is to extend the specification language as follows:

$$\begin{array}{rrcl} (\mathit{CHSpec}) & \Psi & \in & \mathit{Labels} \rightharpoonup \mathit{Assert} \\ (\mathit{Assert}) & \mathtt{a} & \in & \mathit{CHSpec} \to \mathit{State} \to \mathsf{Prop} \end{array}$$

and a code pointer f with specification a is defined as:

$$\mathsf{codeptr}(\mathtt{f},\mathtt{a}) \triangleq \lambda\Psi,\mathbb{S}.\ \mathtt{f} \in \mathit{dom}(\Psi) \wedge \Psi(\mathtt{f}) = \mathtt{a}\,.$$

Unfortunately, this simple solution does not work because the definitions of *CHSpec* and *Assert* mutually refer to each other and are not well-founded. To break the circularity, Ni and Shao [75] defined a syntactic specification language. In their XCAP, the program specification is in the following form.

$$\begin{array}{rrcl} (\mathit{CHSpec}) & \Psi & \in & \mathit{Labels} \rightharpoonup \mathit{Assert} \\ (\mathit{PropX}) & \mathtt{P} & ::= & \ldots \\ (\mathit{Assert}) & \mathtt{a} & \in & \mathit{State} \to \mathit{PropX} \\ (\mathit{Interp}) & [\![\_]\!] & \in & \mathit{PropX} \to (\mathit{CHSpec} \to \mathsf{Prop}) \end{array}$$

The meaning of extended proposition P is given by the interpretation $[\![\mathtt{P}]\!]_\Psi$. A code pointer specification $\mathsf{codeptr}(\mathtt{f},\mathtt{a})$ is just a built-in syntactic construct in *PropX*, whose interpretation is:

$$[\![\, \mathsf{codeptr}(\mathtt{f},\mathtt{a}) \,]\!]_{\Psi} \triangleq \mathtt{f} \in dom(\Psi) \wedge \Psi(\mathtt{f}) = \mathtt{a}\,.$$

"*State* $\rightarrow$ *PropX*" does not have to be the only form of specification language used for certified assembly programming. For instance, the register file type used in TAL can be treated as a specification language. We can generalize the XCAP approach to support different specification languages [35]. Then we get the following generic framework:

$$\begin{array}{rrcl}
(\mathit{CHSpec}) & \Psi & \in & \mathit{Labels} \rightharpoonup \mathit{CdSpec} \\
(\mathit{CdSpec}) & \theta & \in & \ldots \\
(\mathit{Interp}) & [\![\,\_\,]\!] & \in & \mathit{CdSpec} \rightarrow (\mathit{CHSpec} \rightarrow \mathit{State} \rightarrow \mathsf{Prop})
\end{array}$$

where the code specification θ can be of different forms, as long as appropriate interpretations are defined. A code pointer $\mathtt{f}$ with specification θ is now formulated as:

$$\mathsf{codeptr}(\mathtt{f},\theta) \;\triangleq\; \lambda\Psi,\mathbb{S}.\ \mathtt{f} \in dom(\Psi) \wedge \Psi(\mathtt{f}) = \theta\,.$$

Although generic, this framework is not "open" because it only allows homogeneous program specification Ψ with a specific type of θ. If program modules are specified in different specification languages, the code pointer $\mathtt{f}_1$ specified in the specification language $\mathcal{L}_1$ is formulated as $\mathsf{codeptr}(\mathtt{f}_1,\theta_{\mathcal{L}_1})$, while code pointer $\mathtt{f}_2$ in $\mathcal{L}_2$ is specified as $\mathsf{codeptr}(\mathtt{f}_2,\theta_{\mathcal{L}_2})$. To make both codeptr definable, we need a heterogeneous program specification Ψ in OCAP.

3.2 OCAP Specifications

The first attempt to define the program specifications for OCAP is to take advantage of the support of dependent types in CiC and pack each code specification θ with its corresponding interpretation.

$$
\begin{array}{rcll}
(\mathit{LangTy}) & \mathcal{L} & ::= & (\mathit{CiC\ terms}) \ \in\ \mathsf{Type} \\
(\mathit{CdSpec}) & \theta & ::= & (\mathit{CiC\ terms}) \ \in\ \mathcal{L} \\
(\mathit{Assert}) & \mathsf{a} & \in & \mathit{CHSpec} \rightarrow \mathit{State} \rightarrow \mathsf{Prop} \\
(\mathit{Interp}) & [\![\,\_\,]\!]_{\mathcal{L}} & \in & \mathcal{L} \rightarrow \mathit{Assert} \\
(\mathit{OCdSpec}) & \pi & ::= & \langle \mathcal{L}, [\![\,\_\,]\!]_{\mathcal{L}}, \theta \rangle \ \in\ \Sigma X.(X \rightarrow \mathit{Assert}) * X \\
(\mathit{CHSpec}) & \Psi & \in & \mathit{Labels} \rightharpoonup \mathit{OCdSpec}
\end{array}
$$

As shown above, specifications in each specification language will be encoded in CiC as θ, whose type $\mathcal{L}$ is also defined in CiC. The interpretation $[\![\,\_\,]\!]_{\mathcal{L}}$ for the language $\mathcal{L}$ maps θ to the OCAP assertion a. The language-specific specification θ is lifted to an "open" specification π, which is a dependent package containing the language type $\mathcal{L}$, its interpretation function $[\![\,\_\,]\!]_{\mathcal{L}}$ and the specification θ. The heterogeneous program specification Ψ is simply defined as a partial mapping from code labels to the lifted specification π.

Unfortunately, this obvious solution introduces circularity again, because definitions of *CHSpec* and *OCdSpec* refer to each other. To break the circularity, we remove the interpretation from π and collect all the interpretations into an extra "language dictionary".

The OCAP Solution. The final definition of OCAP program specification constructs is shown in Figure 3.3. To embed a system into OCAP, we first assign a unique ID ρ to its specification language. Specifications in that language and their type are still represented as θ and $\mathcal{L}$. Both are CiC terms. The lifted specification π now contains the language ID ρ, the corresponding language type $\mathcal{L}$ and the specification θ. The program specification Ψ is a binary relation of code labels and lifted code specifications. We do not define Ψ as a partial mapping because the interface of modules may be specified in more than one specification language.

As explained above, the interpretation for language $\mathcal{L}$ maps specifications in $\mathcal{L}$ to as-

$$\begin{array}{rcll} (\mathit{LangID}) & \rho & ::= & n\ (\mathit{nat\ nums}) \\ (\mathit{LangTy}) & \mathcal{L} & ::= & (\mathit{CiC\ terms}) \quad \in \ \mathsf{Type} \\ (\mathit{CdSpec}) & \theta & ::= & (\mathit{CiC\ terms}) \quad \in \ \mathcal{L} \\ (\mathit{OCdSpec}) & \pi & ::= & \langle \rho, \mathcal{L}, \theta \rangle \qquad \in \ \mathit{LangID} * (\Sigma X.X) \\ (\mathit{CHSpec}) & \Psi & \in & 2^{\mathit{Labels} * \mathit{OCdSpec}} \\ (\mathit{Assert}) & \mathsf{a} & \in & \mathit{CHSpec} \to \mathit{State} \to \mathsf{Prop} \\ (\mathit{Interp}) & [\![\_]\!]_{\mathcal{L}} & \in & \mathcal{L} \to \mathit{Assert} \\ (\mathit{LangDict}) & \mathcal{D} & \in & \mathit{LangID} \rightharpoonup \Sigma X.(X \to \mathit{Assert}) \end{array}$$

Figure 3.3: Specification Constructs of OCAP

sertions a. To avoid circularity, we do not put the interpretation $[\![\_]\!]_{\mathcal{L}}$ in π. Instead, we collect the interpretations and put them in a language dictionary $\mathcal{D}$, which maps language IDs to dependent pairs containing the language type and the corresponding interpretation.

Given a lifted specification π, the following operation maps it to an assertion a:

$$[\![\![\langle \rho, \mathcal{L}, \theta \rangle]\!]\!]_{\mathcal{D}} \triangleq \lambda \Psi, \mathbb{S}.\ \exists [\![\_]\!]_{\mathcal{L}}.\ (\mathcal{D}(\rho) = \langle \mathcal{L},\ [\![\_]\!]_{\mathcal{L}} \rangle) \ \wedge\ ([\![\theta]\!]_{\mathcal{L}}\ \Psi\ \mathbb{S}). \tag{3.3}$$

It takes the language ID ρ and looks up the interpretation from $\mathcal{D}$. Then the interpretation is applied to the specification θ. If there is no interpretation found, the result is simply false.

We allow a specification language $\mathcal{L}$ to have more than one interpretation, each assigned a different language ID. That is why we use ρ instead of $\mathcal{L}$ to look up the interpretation from $\mathcal{D}$.

3.3 OCAP Inference Rules

Figure 3.4 shows OCAP inference rules. The PROG rule is similar to the one for CAP shown in Figure 3.2, but with several differences:

$\boxed{\mathcal{D};\Psi\vdash\mathbb{P}}$ **(*Well-formed program*)**

$$\frac{\mathcal{D};\Psi\vdash\mathbb{C}:\Psi \qquad (\mathsf{a}\ \Psi\ \mathbb{S}) \qquad \mathcal{D}\vdash\{\mathsf{a}\}\,\mathsf{pc}:\mathbb{C}[\mathsf{pc}]}{\mathcal{D};\Psi\vdash(\mathbb{C},\mathbb{S},\mathsf{pc})}\ (\text{PROG})$$

$\boxed{\mathcal{D};\Psi\vdash\mathbb{C}:\Psi'}$ **(*Well-formed code heap*)**

$$\frac{\text{for all } (\mathtt{f},\pi)\in\Psi'\text{:} \qquad \mathsf{a}=\langle[\![\,\pi\,]\!]_{\mathcal{D}}\rangle_{\Psi} \qquad \mathcal{D}\vdash\{\mathsf{a}\}\,\mathtt{f}:\ \mathbb{C}[\mathtt{f}]}{\mathcal{D};\Psi\vdash\mathbb{C}:\Psi'}\ (\text{CDHP})$$

$$\frac{\begin{array}{lll}\mathcal{D}_1;\Psi_1\vdash\mathbb{C}_1:\Psi_1' & \mathcal{D}_2;\Psi_2\vdash\mathbb{C}_2:\Psi_2' & \\ \mathsf{ITP}(\mathcal{D}_1,\Psi_1') & \mathsf{ITP}(\mathcal{D}_2,\Psi_2') & \mathcal{D}_1\sharp\mathcal{D}_2 \quad \mathbb{C}_1\sharp\mathbb{C}_2\end{array}}{\mathcal{D}_1\cup\mathcal{D}_2;\Psi_1\cup\Psi_2\vdash\mathbb{C}_1\cup\mathbb{C}_2:\Psi_1'\cup\Psi_2'}\ (\text{LINK*})$$

$\boxed{\mathcal{D}\vdash\{\mathsf{a}\}\,\mathtt{f}:\ \mathbb{I}}$ **(*Well-formed instruction sequence*)**

$$\frac{\mathsf{a}\Rightarrow\lambda\Psi',\mathbb{S}.\ \exists\pi'.(\mathsf{codeptr}(\mathtt{f}',\pi')\wedge[\![\,\pi'\,]\!]_{\mathcal{D}})\ \Psi'\ \mathbb{S}}{\mathcal{D}\vdash\{\mathsf{a}\}\,\mathtt{f}:\ \mathsf{j}\,\mathtt{f}'}\ (\text{J})$$

$$\frac{\mathsf{a}\Rightarrow\lambda\Psi',\mathbb{S}.\ \exists\pi'.(\mathsf{codeptr}(\mathbb{S}.\mathbb{R}(\mathtt{r}_s),\pi')\wedge[\![\,\pi'\,]\!]_{\mathcal{D}})\ \Psi'\ \mathbb{S}}{\mathcal{D}\vdash\{\mathsf{a}\}\,\mathtt{f}:\ \mathsf{jr}\,\mathtt{r}_s}\ (\text{JR})$$

$$\frac{\begin{array}{l}\mathsf{a}\Rightarrow\lambda\Psi',\mathbb{S}.\ \exists\pi'.\ (\mathsf{codeptr}(\mathtt{f}',\pi')\wedge[\![\,\pi'\,]\!]_{\mathcal{D}})\ \Psi'\ \hat{\mathbb{S}}\\ \qquad\qquad\text{where } \hat{\mathbb{S}}=(\mathbb{S}.\mathbb{H},\mathbb{S}.\mathbb{R}\{\mathtt{r}_{31}\leadsto\mathtt{f}{+}1\})\end{array}}{\mathcal{D}\vdash\{\mathsf{a}\}\,\mathtt{f}:\ \mathsf{jal}\ \mathtt{f}';\mathbb{I}}\ (\text{JAL})$$

$$\frac{\begin{array}{ll}\iota\in\{\mathsf{addu},\mathsf{addiu},\mathsf{lw},\mathsf{subu},\mathsf{sw}\} & \\ \mathcal{D}\vdash\{\mathsf{a}'\}\,\mathtt{f}{+}1:\ \mathbb{I} & \mathsf{a}\Rightarrow\lambda\Psi'.\ (\mathsf{a}'\ \Psi')\circ\mathsf{NextS}_{(\mathtt{f},\,\iota)}\end{array}}{\mathcal{D}\vdash\{\mathsf{a}\}\,\mathtt{f}:\ \iota;\ \mathbb{I}}\ (\text{SEQ})$$

$$\frac{\begin{array}{l}\mathcal{D}\vdash\{\mathsf{a}''\}\,\mathbb{I}\\ \mathsf{a}\Rightarrow\lambda\Psi',\mathbb{S}.\ (\mathbb{S}.\mathbb{R}(\mathtt{r}_s)\leq 0\rightarrow\mathsf{a}''\ \Psi'\ \mathbb{S})\\ \qquad\qquad\wedge\ (\mathbb{S}.\mathbb{R}(\mathtt{r}_s)>0\rightarrow\\ \qquad\qquad\qquad\exists\pi'.\ (\mathsf{codeptr}(\mathtt{f}',\pi')\wedge[\![\,\pi'\,]\!]_{\mathcal{D}})\ \Psi'\ \mathbb{S})\end{array}}{\mathcal{D}\vdash\{\mathsf{a}\}\,\mathtt{f}:\ \mathsf{bgtz}\ \mathtt{r}_s,\mathtt{f}';\ \mathbb{I}}\ (\text{BGTZ})$$

$$\frac{\begin{array}{l}\mathcal{D}\vdash\{\mathsf{a}''\}\,\mathbb{I}\\ \mathsf{a}\Rightarrow\lambda\Psi',\mathbb{S}.\ (\mathbb{S}.\mathbb{R}(\mathtt{r}_s)\neq\mathbb{S}.\mathbb{R}(\mathtt{r}_t)\rightarrow\mathsf{a}''\ \Psi'\ \mathbb{S})\\ \qquad\qquad\wedge\ (\mathbb{S}.\mathbb{R}(\mathtt{r}_s)=\mathbb{S}.\mathbb{R}(\mathtt{r}_t)\rightarrow\\ \qquad\qquad\qquad\exists\pi'.\ (\mathsf{codeptr}(\mathtt{f}',\pi')\wedge[\![\,\pi'\,]\!]_{\mathcal{D}})\ \Psi'\ \mathbb{S})\end{array}}{\mathcal{D}\vdash\{\mathsf{a}\}\,\mathtt{f}:\ \mathsf{beq}\ \mathtt{r}_s,\mathtt{r}_t,\mathtt{f}';\ \mathbb{I}}\ (\text{BEQ})$$

$$\frac{\mathsf{a}\Rightarrow\mathsf{a}' \qquad \mathcal{D}'\subseteq\mathcal{D} \qquad \mathcal{D}'\vdash\{\mathsf{a}'\}\,\mathtt{f}:\mathbb{I}}{\mathcal{D}\vdash\{\mathsf{a}\}\,\mathtt{f}:\mathbb{I}}\ (\text{WEAKEN*})$$

Figure 3.4: OCAP Inference Rules

- In addition to the program specification Ψ, OCAP requires a language dictionary $\mathcal{D}$ to interpret code specifications.
- The well-formedness of $\mathbb{C}$ is checked with respect to $\mathcal{D}$ and Ψ.
- The assertion $\mathtt{a}$ is now a predicate over code heap specifications and states. It holds over Ψ and the current state $\mathbb{S}$.
- We check the well-formedness of the current instruction sequences $\mathbb{C}[\mathtt{pc}]$ with respect to $\mathcal{D}$ and $\mathtt{a}$.

As in CAP, to certify programs using OCAP, we only need to prove that the invariant holds at the initial program $(\mathbb{C}, \mathbb{S}_0, \mathtt{pc}_0)$. The precondition $\mathtt{a}$ specifies the initial state $\mathbb{S}_0$. It takes Ψ as argument to specify embedded code pointers in $\mathbb{S}_0$, as explained before. The soundness of OCAP will guarantee that the invariant holds at each step of execution and that the invariant ensures program progress.

Well-formed code heaps. The CDHP rule checks that the specification asserted at each $\mathtt{f}$ in Ψ' ensures safe execution of the corresponding instruction sequence $\mathbb{C}[\mathtt{f}]$. As in CAP, the Ψ on the left hand side specifies the code to which each $\mathbb{C}[\mathtt{f}]$ may jump. Instead of using the specification $\Psi'(\mathtt{f})$ directly, we first map it to an assertion ($[\![\![\, \Psi'(\mathtt{f}) \,]\!]\!]_{\mathcal{D}}$) by applying the corresponding interpretation defined in $\mathcal{D}$. Then we do another lifting $\langle \_ \rangle_{\Psi}$, which is defined as:

$$\langle \mathtt{a} \rangle_{\Psi} \triangleq \left(\bigwedge_{(\mathtt{f},\pi)\in\Psi} \mathsf{codeptr}(\mathtt{f},\pi) \right) \wedge \mathtt{a}\,. \tag{3.4}$$

Here $\mathsf{codeptr}(\mathtt{f}, \pi)$ is defined as the following assertion:

$$\mathsf{codeptr}(\mathtt{f},\pi) \triangleq \lambda\Psi,\mathbb{S}.\ (\mathtt{f},\pi) \in \Psi\,. \tag{3.5}$$

In Equation 3.4, we also overload the conjunction connector "$\wedge$"for assertions:

$$\mathtt{a} \wedge \mathtt{a}' \triangleq \lambda\Psi,\mathbb{S}.\ \mathtt{a}\ \Psi\ \mathbb{S} \wedge \mathtt{a}'\ \Psi\ \mathbb{S}\,. \tag{3.6}$$

Therefore, the lifted assertion ($\langle [\![\, \Psi'(\mathtt{f}) \,]\!]_{\mathcal{D}} \rangle_{\Psi}$) carries the knowledge of the code pointers in Ψ, which may be reached from $\mathbb{C}[\mathtt{f}]$. When we check $\mathbb{C}[\mathtt{f}]$, we do not need to carry Ψ, but we need to carry $\mathcal{D}$ to interpret the specification π for each $\mathsf{codeptr}(\mathtt{f}, \pi)$.

Linking of modules. The $\mathbb{C}$ checked in the CDHP rule does not have to be the global code heap used in the PROG rule. Subsets $\mathbb{C}_i$ of the complete code heap can be certified with local interfaces $\mathcal{D}_i$, Ψ_i and Ψ_i'. Then they are linked using the admissible LINK rule. An admissible rule is a rule that can be proved as a lemma based on other rules. We use a star (*) in the name to distinguish admissible rules from normal rules. Below we will prove the admissibility of the LINK rule in Lemma 3.3. Here the compatibility of partial mappings f and g is defined as

$$f \sharp g \;\triangleq\; \forall x.\; x \in dom(f) \land x \in dom(g) \rightarrow f(x) = g(x)\,. \tag{3.7}$$

We use $\mathsf{ITP}(\mathcal{D}, \Psi)$ to represent that $\mathcal{D}$ contains all interpretations needed to interprete Ψ, *i.e.*,

$$\mathsf{ITP}(\mathcal{D}, \Psi) \;\triangleq\; \forall \mathtt{f}, \pi.\; (\mathtt{f}, \pi) \in \Psi \rightarrow (\pi.\rho \in dom(\mathcal{D})). \tag{3.8}$$

The LINK rule shows the openness of OCAP: $\mathbb{C}_1$ and $\mathbb{C}_2$ may be specified and certified in different verification systems with interpretations defined in $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively. Proofs constructed in foreign systems are converted to proofs of OCAP judgments $\mathcal{D}_i; \Psi_i \vdash \mathbb{C}_i : \Psi_i'$ at the time of linkage. We will demonstrate this in the following sections.

Well-formed instruction sequences. Rules for jump instructions (J, JR and JAL) are simple. They require that the target address be a valid code pointer with specification π', and that there be an interpretation for π' in $\mathcal{D}$. The interpretation of π' should hold at the resulting state of the jump. Here, we use $\mathsf{a} \Rightarrow \mathsf{a}'$ to represent the logical implication between OCAP assertions:

$$\mathsf{a} \Rightarrow \mathsf{a}' \;\triangleq\; \forall \Psi, \mathbb{S}.\; \mathsf{a}\;\Psi\;\mathbb{S} \rightarrow \mathsf{a}'\;\Psi\;\mathbb{S}. \tag{3.9}$$

The SEQ rule for sequential instructions is similar to the CAP SEQ rule. It requires no further explanation. The BGTZ rule and BEQ rule are like a simple combination of the J rule and the SEQ rule, which are straightforward to understand too.

The WEAKEN rule. The WEAKEN rule is also admissible in OCAP, whose admissibility is proved in Lemma 3.3. It is a normal rule in Hoare-style program logics, but plays an important role in OCAP to interface foreign verification systems. The instruction sequence $\mathbb{I}$ may have specifications θ and θ' in different foreign systems. Their interpretations are $\mathtt{a}$ and $\mathtt{a}'$, respectively. If the proof of $\mathcal{D}' \vdash \{\mathtt{a}'\}\,\mathtt{f} : \mathbb{I}$ is converted from proof constructed in the system where $\mathbb{I}$ is certified with specification θ', it can be called from the other system as long as $\mathtt{a}$ is stronger than $\mathtt{a}'$. The use of this rule will be shown in Section 6.1.

Lemma 3.3 (Admissibility of WEAKEN and LINK)

1. If $\mathcal{D}' \subseteq \mathcal{D}$, $\mathtt{a} \Rightarrow \mathtt{a}'$, and $\mathcal{D}' \vdash \{\mathtt{a}'\}\,\mathtt{f} : \mathbb{I}$, then $\mathcal{D} \vdash \{\mathtt{a}\}\,\mathtt{f} : \mathbb{I}$. That is, the WEAKEN rule is admissible.

2. If $\mathcal{D}_1; \Psi_1 \vdash \mathbb{C}_1 : \Psi'_1$, $\mathcal{D}_2; \Psi_2 \vdash \mathbb{C}_2 : \Psi'_2$, $\mathsf{ITP}(\mathcal{D}_1, \Psi'_1)$, $\mathsf{ITP}(\mathcal{D}_2, \Psi'_2)$, $\mathcal{D}_1 \sharp \mathcal{D}_2$ and $\mathbb{C}_1 \sharp \mathbb{C}_2$, then $\mathcal{D}_1 \cup \mathcal{D}_2; \Psi_1 \cup \Psi_2 \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi'_1 \cup \Psi'_2$. That is, the LINK rule is admissible.

Proof. The proof of 1 contains two steps. In the first step, we first prove $\mathcal{D}' \vdash \{\mathtt{a}\}\,\mathtt{f} : \mathbb{I}$. This can be done by induction over the structure of $\mathbb{I}$. Then we prove $\mathcal{D} \vdash \{\mathtt{a}\}\,\mathtt{f} : \mathbb{I}$, given $\mathcal{D}' \subseteq \mathcal{D}$ and $\mathcal{D}' \vdash \{\mathtt{a}\}\,\mathtt{f} : \mathbb{I}$. This, again, can be done by induction over the structure of $\mathbb{I}$. We also need to use Lemma 3.5.

The proof of 2 is trivial, given 1, Lemma 3.4 and the CDHP rule. □

Lemma 3.4 For all $\mathtt{a}$, if $\Psi \subseteq \Psi'$, then $\langle \mathtt{a} \rangle_{\Psi'} \Rightarrow \langle \mathtt{a} \rangle_{\Psi}$.

Proof. Trivial, by the definition shown in Equation 3.4. □

Lemma 3.5 For all π, if $\mathcal{D} \subseteq \mathcal{D}'$, then $[\![\, \pi \,]\!]_{\mathcal{D}} \Rightarrow [\![\, \pi \,]\!]_{\mathcal{D}'}$.

Proof. The proof is trivial by the definition shown in Equation 3.3. □

3.4 Soundness of OCAP

The soundness of OCAP inference rules is proved following the syntactic approach [99] to proving type soundness. We need to first prove the progress and preservation lemmas.

Lemma 3.6 (Progress)

If $\mathcal{D}; \Psi \vdash \mathbb{P}$, there exists $\mathbb{P}'$ such that $\mathbb{P} \longmapsto \mathbb{P}'$.

Proof. Suppose $\mathbb{P} = (\mathbb{C}, \mathbb{S}, \mathsf{pc})$. By inversion of the PROG rule, we have $\mathcal{D}; \Psi \vdash \mathbb{C} : \Psi$, $\mathtt{a}\ \Psi\ \mathbb{S}$ and $\mathcal{D} \vdash \{\mathtt{a}\}\, \mathsf{pc} :\ \mathbb{C}[\mathsf{pc}]$ for some $\mathtt{a}$. By inversion of all the well-formed instruction sequence rules and the definition of the operational semantics in Figure 2.3, we will see that the program $\mathbb{P}$ can execute one step, given $\mathtt{a}\ \Psi\ \mathbb{S}$. The only tricky part is to prove that, if $\mathbb{C}(\mathsf{pc})$ is a jumps (j, jr or jal) or a branches (beq or bgtz), the target address is in the domain of $\mathbb{C}$. This can be derived from the following Lemmas 3.8 and 3.9, and inversion of corresponding instruction rules. □

The following Preservation lemma is an extension of the one normally used in syntactic approach to soundness of type systems. It is important to help us prove the support of partial correctness in our soundness theorem. Before showing the lemma, we first introduce the following definition:

$$[\![\, \Psi(\mathtt{f})\,]\!]_{\mathcal{D}} \triangleq \lambda \Psi', \mathbb{S}.\ \exists \pi.\ ((\mathtt{f}, \pi) \in \Psi) \wedge [\![\, \pi_i\,]\!]_{\mathcal{D}}\ \Psi'\ \mathbb{S} \tag{3.10}$$

Lemma 3.7 (Preservation)

If $\mathcal{D}; \Psi \vdash (\mathbb{C}, \mathbb{S}, \mathsf{pc})$ and $(\mathbb{C}, \mathbb{S}, \mathsf{pc}) \longmapsto (\mathbb{C}, \mathbb{S}', \mathsf{pc}')$, then $\mathcal{D}; \Psi \vdash (\mathbb{C}, \mathbb{S}', \mathsf{pc}')$, and

1. if $\mathbb{C}(\mathsf{pc})$ is a jump instruction, *i.e.*, $\mathsf{j}\ \mathtt{f}$, $\mathsf{jal}\ \mathtt{f}$ or $\mathsf{jr}\ \mathtt{r}_s$, then $[\![\, \Psi(\mathsf{pc}')\,]\!]_{\mathcal{D}}\ \Psi\ \mathbb{S}'$;
2. if $\mathbb{C}(\mathsf{pc}) = \mathsf{bgtz}\ \mathtt{r}_s, \mathtt{f}$ and $\mathbb{S}.\mathbb{R}(\mathtt{r}_s) > 0$, then $[\![\, \Psi(\mathsf{pc}')\,]\!]_{\mathcal{D}}\ \Psi\ \mathbb{S}'$;
3. if $\mathbb{C}(\mathsf{pc}) = \mathsf{beq}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f}$ and $\mathbb{S}.\mathbb{R}(\mathtt{r}_s) = \mathbb{S}.\mathbb{R}(\mathtt{r}_t)$, then $[\![\, \Psi(\mathsf{pc}')\,]\!]_{\mathcal{D}}\ \Psi\ \mathbb{S}'$.

Proof. Given $\mathcal{D}; \Psi \vdash (\mathbb{C}, \mathbb{S}, \mathsf{pc})$, we know by an inversion of the PROG rule there exists an "$\mathtt{a}$" such that $\mathcal{D}, \Psi \vdash \mathbb{C} : \Psi$, $\mathtt{a}\ \Psi\ \mathbb{S}$ and $\mathcal{D} \vdash \{\mathtt{a}\}\, \mathsf{pc} :\ \mathbb{C}[\mathsf{pc}]$. To prove $\mathcal{D}; \Psi \vdash (\mathbb{C}, \mathbb{S}', \mathsf{pc}')$, we need to find an assertion "$\mathtt{a}'$" and prove the following three sub-goals: $\mathcal{D}, \Psi \vdash \mathbb{C} : \Psi$, $\mathtt{a}' \ \Psi \ \mathbb{S}'$ and $\mathcal{D} \vdash \{\mathtt{a}'\}\, \mathtt{pc}' : \ \mathbb{C}[\mathtt{pc}']$. We have got the first sub-goal. To prove the last two sub-goals, we analyze all possible cases of $\mathbb{C}[\mathtt{pc}]$:

- if $\mathbb{C}[\mathtt{pc}]$ starts with a normal sequential instruction, the second and the third sub-goals can be proved by $\mathcal{D} \vdash \{\mathtt{a}\}\, \mathtt{pc} : \ \mathbb{C}[\mathtt{pc}]$ and a simple inversion of the SEQ rule;

- if $\mathbb{C}[\mathtt{pc}]$ starts with a conditional branch instruction and the condition is false, we can prove the second and the third sub-goals by an inversion of the BEQ and BGTZ rule;

- if $\mathbb{C}[\mathtt{pc}]$ starts with a jump instruction, or it starts with a conditional branch instruction and the condition is true, we know by inversion of the corresponding rules that there exists π such that $(\mathsf{codeptr}(\mathtt{pc}', \pi) \wedge [\![\, \pi \,]\!]_{\mathcal{D}}) \ \Psi \ \mathbb{S}$. We let $\langle [\![\, \pi \,]\!]_{\mathcal{D}} \rangle_{\Psi}$ be the assertion $\mathtt{a}'$. So we can prove the second sub-goal by Lemma 3.10. The third sub-goal follows Lemma 3.11.

Now that we have proved $\mathcal{D}; \Psi \vdash (\mathbb{C}, \mathbb{S}', \mathtt{pc}')$, it is trivial to prove 1–5 given the fact that we use in the third bullet above $\langle [\![\, \pi \,]\!]_{\mathcal{D}} \rangle_{\Psi}$ as the assertion $\mathtt{a}'$ and proved $\mathtt{a}' \ \Psi \ \mathbb{S}'$. □

Lemma 3.8 If $\mathcal{D}; \Psi \vdash \mathbb{C} : \Psi'$, then, for all $(\mathtt{f}, \pi) \in \Psi'$, we have $\mathtt{f} \in \mathit{dom}(\mathbb{C})$.

Proof. Trivial, follows the CDHP rule. □

Lemma 3.9 If $\mathsf{codeptr}(\mathtt{f}, \pi) \ \Psi \ \mathbb{S}$ and $\mathcal{D}; \Psi \vdash \mathbb{C} : \Psi$, then $\mathtt{f} \in \mathit{dom}(\mathbb{C})$.

Proof. By the definition of codeptr in Equation 3.5 and Lemma 3.8. □

Lemma 3.10 For all assertion $\mathtt{a}$, we have $\forall \Psi, \mathbb{S}.\ \mathtt{a} \ \Psi \ \mathbb{S} \leftrightarrow \langle \mathtt{a} \rangle_{\Psi} \ \Psi \ \mathbb{S}$.

Proof. Trivially follows the definition shown in Equation 3.4. □

Lemma 3.11 If $\mathcal{D}; \Psi \vdash \mathbb{C} : \Psi'$ and $\mathsf{codeptr}(\mathtt{f}, \pi) \ \Psi' \ \mathbb{S}$, then $\mathcal{D} \vdash \{\mathtt{a}\}\, \mathtt{f} : \ \mathbb{C}[\mathtt{f}]$, where $\mathtt{a} = \langle [\![\, \pi \,]\!]_{\mathcal{D}} \rangle_{\Psi'}$.

Proof. By $\mathsf{codeptr}(\mathtt{f}, \pi)\ \ \Psi'\ \ \mathbb{S}$ and the definition of $\mathsf{codeptr}$ in Equation 3.5, we know $(\mathtt{f}, \pi) \in \Psi'$. Then the proof is trivial follows the CDHP rule, given $\mathcal{D}; \Psi \vdash \mathbb{C} : \Psi'$. □

The soundness of OCAP is formalized below. It not only shows that programs certified in OCAP will never get stuck, but also guarantees that assertions specified in Ψ for target addresses of jump and branch instructions actually hold when we reach corresponding program points. This essentially captures the partial correctness of certified programs.

Theorem 3.12 (OCAP-Soundness)
If $\mathcal{D}; \Psi \vdash (\mathbb{C}, \mathbb{S}, \mathsf{pc})$, for all natural number n there exist $\mathbb{S}'$ and pc' such that $(\mathbb{C}, \mathbb{S}, \mathsf{pc}) \longmapsto^{n} (\mathbb{C}, \mathbb{S}', \mathsf{pc}')$, and

1. if $\mathbb{C}(\mathsf{pc}') = \mathsf{j}\ \mathtt{f}$, then $[\![\, \Psi(\mathtt{f}) \,]\!]_{\mathcal{D}}\ \Psi\ \mathbb{S}'$;
2. if $\mathbb{C}(\mathsf{pc}') = \mathsf{jal}\ \mathtt{f}$, then $[\![\, \Psi(\mathtt{f}) \,]\!]_{\mathcal{D}}\ \Psi\ (\mathbb{S}'.\mathbb{H}, \mathbb{S}'.\mathbb{R}\{\mathtt{r}_{31} \rightsquigarrow \mathsf{pc}'{+}1\})$;
3. if $\mathbb{C}(\mathsf{pc}') = \mathsf{jr}\ \mathtt{r}_s$, then $[\![\, \Psi(\mathbb{S}'.\mathbb{R}(\mathtt{r}_s)) \,]\!]_{\mathcal{D}}\ \Psi\ \mathbb{S}'$;
4. if $\mathbb{C}(\mathsf{pc}') = \mathsf{bgtz}\ \mathtt{r}_s, \mathtt{f}$ and $\mathbb{S}'.\mathbb{R}(\mathtt{r}_s) > 0$, then $[\![\, \Psi(\mathtt{f}) \,]\!]_{\mathcal{D}}\ \Psi\ \mathbb{S}'$.

Proof. The proof is trivial by induction over n, and applying the Progress and Preservation lemmas at each step. □

Therefore, if the interpretation for a specification language captures the invariant enforced in the corresponding verification system, the soundness of OCAP ensures that the invariant holds when the modules certified in that system get executed.

OCAP's ability to support partial correctness of programs benefits from the way we specify codeptr, as proposed by Ni and Shao [75]. This makes OCAP a more expressive framework than the Princeton FPCC framework [6, 8, 94], where the step-indexed semantic model of codeptr makes it very difficult, if possible, to prove similar properties of programs.

$$\begin{array}{rl}
(\mathit{InitFlag}) & \varphi ::= 0 \mid 1 \\
(\mathit{WordTy}) & \tau ::= \alpha \mid \mathsf{int} \mid \forall[\Delta].\Gamma \mid \langle\tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle \mid \exists\alpha\!:\!\tau. \mid \mu\alpha.\,\tau \\
(\mathit{TyVarEnv}) & \Delta ::= \cdot \mid \alpha,\Delta \\
(\mathit{RfileTy}) & \Gamma ::= \{\mathtt{r}\rightsquigarrow\tau\}^* \\
(\mathit{CHType}) & \psi ::= \{(\mathtt{f},\ [\Delta].\,\Gamma)\}^* \\
(\mathit{DHType}) & \Phi ::= \{\mathtt{l}\rightsquigarrow\tau^{\varphi}\}^*
\end{array}$$

Figure 3.5: Type Definitions of TAL

3.5 Embedding TAL into OCAP

Given a specific verification system (henceforth called a foreign system) with the definition of $\mathcal{L}$ and a language ID ρ, it takes three steps to embed the system into OCAP: identifying the invariant enforced in the system; defining an interpretation for specifications θ (usually an encoding of the invariant into an assertion $\mathtt{a}$); and proving the soundness of the embedding by showing that inference rules in the foreign system can be proved as lemmas in OCAP based on the interpretation.

As an example to illustrate the expressiveness of OCAP, we show how to embed Typed Assembly Languages (TALs) [69, 27] into OCAP. Unlike traditional TALs based on abstract machines with primitive operations for memory allocation, we present a variation of TAL for our machine defined in Section 2.2.

3.5.1 TAL types and typing rules.

Figure 3.5 shows the definition of TAL types, including polymorphic code types, mutable references, existential types, and recursive types. Definitions for types are similar to the original TAL. Γ is the type for the register file. $\forall[\Delta].\Gamma$ is the polymorphic type for code pointers, which means the code pointer expects a register file of type Γ with type variables declared in Δ. The flag φ is used to mark whether memory cell has been initialized or not. $\langle\tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle$ is the type for a mutable reference pointing to a tuple in the heap. The fresh memory cells returned by memory allocation libraries will have types with flag 0.

The reader should keep in mind that this TAL is designed for TM, so there is no "heap values" as in the original TAL. Also, since we separate code heap and data heap in our TM, specifications for them are separated too. We use ψ for code heap type and Φ for data heap type.

We present TALtyping rules in Figures 3.6 and 3.7. The major judgements are shown below.

Judgement	Meaning
$\psi \vdash \mathbb{C} : \psi'$	*Well-formed code heap*
$\psi \vdash \{[\Delta].\Gamma\}\, \mathbb{I}$	*Well-formed instruction sequence*
$\vdash [\Delta].\, \Gamma \leq [\Delta'].\, \Gamma'$	*Sub-typing of register file types*

The typing rules are similar to the original TAL [69] and are not explained in details here. But we do not need a PROG rule to type check whole programs $\mathbb{P}$ because this TAL will be embedded in OCAP and only be used to type check code heaps, which may be a subset of the whole program code. Readers who are not familiar with TAL can view $[\Delta].\Gamma$ as assertions about states and the subtyping relation as logic implication. Then the rules in Figure 3.6 look very similar to CAP rules shown in Figure 3.2. Actually this is exactly how we embed TAL in OCAP below.

3.5.2 Interpretation and Embedding

As we can see, specifications θ of preconditions for code blocks in TAL are register file types $[\Delta].\Gamma$. We use $\mathcal{L}_{\mathrm{TAL}}$ to represent the meta-type of its CiC encoding. Then we define the mapping between the TAL code heap specification ψ and the OCAP code heap specification Ψ:

$$\llcorner \psi \lrcorner_\rho \triangleq \{(\mathtt{f},\ \langle \rho, \mathcal{L}_{\mathrm{TAL}}, [\Delta].\Gamma \rangle) \mid (\mathtt{f}, [\Delta].\Gamma) \in \psi\} \tag{3.11}$$

$$\ulcorner \Psi \urcorner^\rho \triangleq \{(\mathtt{f}, \theta) \mid (\mathtt{f},\ \langle \rho, \_, \theta \rangle) \in \Psi\} \tag{3.12}$$

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

$$\frac{\psi \vdash \{[\Delta].\,\Gamma\}\ \mathtt{f} :\ \mathbb{C}[\mathtt{f}] \qquad \text{for all } (\mathtt{f}, [\Delta].\,\Gamma) \in \psi'}{\psi \vdash \mathbb{C} : \psi'} \ (\text{CDHP})$$

$\boxed{\psi \vdash \{[\Delta].\,\Gamma\}\ \mathtt{f} :\ \mathbb{I}}$ **(*Well-formed Instruction Sequence*)**

$$\frac{(\mathtt{f}', [\Delta'].\,\Gamma') \in \psi \qquad \vdash [\Delta].\,\Gamma \leq [\Delta'].\,\Gamma'}{\psi \vdash \{[\Delta].\,\Gamma\}\ \mathtt{f} :\ \mathsf{j}\ \mathtt{f}'} \ (\text{J})$$

$$\frac{\begin{array}{c}(\mathtt{f}', [\Delta'].\,\Gamma') \in \psi \qquad (\mathtt{f}{+}1, [\Delta''].\,\Gamma'') \in \psi \\ \vdash [\Delta].\,\Gamma\{\mathtt{r}_{31} \rightsquigarrow \forall[\Delta''].\,\Gamma''\} \ \leq\ [\Delta'].\,\Gamma'\end{array}}{\psi \vdash \{[\Delta].\,\Gamma\}\ \mathtt{f} :\ \mathsf{jal}\ \mathtt{f}';\ \mathbb{I}} \ (\text{JAL})$$

$$\frac{\Gamma(\mathtt{r}_s) = \forall[\Delta'].\,\Gamma' \qquad \vdash [\Delta].\,\Gamma \leq [\Delta'].\,\Gamma'}{\psi \vdash \{[\Delta].\,\Gamma\}\ \mathtt{f} :\ \mathsf{jr}\ \mathtt{r}_s} \ (\text{JR})$$

$$\frac{\Gamma(\mathtt{r}_s) = \mathsf{int} \qquad \psi \vdash \{[\Delta].\,\Gamma\{\mathtt{r}_d \rightsquigarrow \mathsf{int}\}\}\ \mathtt{f}{+}1 :\ \mathbb{I}}{\psi \vdash \{[\Delta].\,\Gamma\}\ \mathtt{f} :\ \mathsf{addiu}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{w};\ \mathbb{I}} \ (\text{ADDI})$$

Figure 3.6: Selected TAL typing rules

$\boxed{\Delta \vdash \tau \qquad \vdash [\Delta].\,\Gamma \leq [\Delta'].\,\Gamma'}$

$$\frac{ftv(\tau) \subseteq \Delta}{\Delta \vdash \tau} \ (\text{TYPE})$$

$$\frac{\Gamma(\mathtt{r}) = \Gamma'(\mathtt{r}) \qquad \forall\, \mathtt{r} \in dom(\Gamma')}{\vdash [].\Gamma \leq [].\Gamma'} \ (\text{SUBT}) \qquad \frac{\Gamma(\mathtt{r}) = \forall[\alpha, \Delta'].\,\Gamma' \qquad \Delta \vdash \tau'}{\vdash [\Delta].\,\Gamma \leq [\Delta].\Gamma\{\mathtt{r} : \forall[\Delta'].\,[\tau'/\alpha]\Gamma'\}} \ (\text{TAPP})$$

$$\frac{\Gamma(\mathtt{r}) = [\tau'/\alpha]\tau \qquad \Delta \vdash \tau'}{\vdash [\Delta].\,\Gamma \leq [\Delta].\Gamma\{\mathtt{r} : \exists\alpha : \tau.\,\}} \ (\text{PACK}) \qquad \frac{\Gamma(\mathtt{r}) = \exists\alpha : \tau.}{\vdash [\Delta].\,\Gamma \leq [\alpha, \Delta].\Gamma\{\mathtt{r} : \tau\}} \ (\text{UNPACK})$$

$$\frac{\Gamma(\mathtt{r}) = [\mu\alpha.\,\tau/\alpha]\tau}{\vdash [\Delta].\,\Gamma \leq [\Delta].\Gamma\{\mathtt{r} : \mu\alpha.\,\tau\}} \ (\text{FOLD}) \qquad \frac{\Gamma(\mathtt{r}) = \mu\alpha.\,\tau}{\vdash [\Delta].\,\Gamma \leq [\Delta].\Gamma\{\mathtt{r} : [\mu\alpha.\,\tau/\alpha]\tau\}} \ (\text{UNFOLD})$$

Figure 3.7: TAL typing rules – II

$$\boxed{\psi \vdash \mathbb{S}\!:\![\Delta].\,\Gamma \quad \psi \vdash \mathbb{H}\!:\!\Phi \quad \psi;\Phi \vdash \mathbb{R}\!:\!\Gamma}$$

$$\frac{\cdot \vdash \tau_i \qquad \psi \vdash \mathbb{H}\!:\!\Phi \qquad \psi;\Phi \vdash \mathbb{R}\!:\![\tau_1,\ldots,\tau_n/\alpha_1,\ldots,\alpha_n]\Gamma}{\psi \vdash \mathbb{S}\!:\![\alpha_1,\ldots,\alpha_n].\Gamma}\ (\text{STATE})$$

$$\frac{\psi;\Phi \vdash \mathbb{H}(\mathtt{l})\!:\!\Phi(\mathtt{l}) \qquad \forall\,\mathtt{l}\!\in\! dom(\Phi)}{\psi \vdash \mathbb{H}\!:\!\Phi}\ (\text{HEAP}) \qquad \frac{\psi;\Phi \vdash \mathbb{R}(\mathtt{r})\!:\!\Gamma(\mathtt{r}) \qquad \forall\,\mathtt{r}\!\in\! dom(\Gamma)}{\psi;\Phi \vdash \mathbb{R}\!:\!\Gamma}\ (\text{RFILE})$$

$$\boxed{\psi;\Phi \vdash \mathtt{w}\!:\!\tau \quad \psi;\Phi \vdash \mathtt{w}\!:\!\tau^{\varphi} \quad \vdash \tau^{\varphi} \le \tau^{\varphi'}}$$

$$\frac{}{\psi;\Phi \vdash \mathtt{w}\!:\!\mathsf{int}}\ (\text{INT}) \qquad \frac{(\mathtt{f},[\Delta].\,\Gamma)\in\psi}{\psi;\Phi \vdash \mathtt{f}\!:\!\forall[\Delta].\,\Gamma}\ (\text{CODE})$$

$$\frac{\cdot \vdash \tau' \qquad \psi;\Phi \vdash \mathtt{f}\!:\!\forall[\alpha,\Delta].\,\Gamma}{\psi;\Phi \vdash \mathtt{f}\!:\!\forall[\Delta].\,[\tau'/\alpha]\Gamma}\ (\text{POLY}) \qquad \frac{\vdash \Phi(\mathtt{l}{+}i{-}1) \le \tau_i^{\varphi_i}}{\psi;\Phi \vdash \mathtt{l}\!:\!\langle\tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle}\ (\text{TUP})$$

$$\frac{\cdot \vdash \tau' \qquad \psi;\Phi \vdash \mathtt{w}\!:\![\tau'/\alpha]\tau}{\psi;\Phi \vdash \mathtt{w}\!:\!\exists\alpha\!:\!\tau.}\ (\text{EXT}) \qquad \frac{\psi;\Phi \vdash \mathtt{w}\!:\![\mu\alpha.\,\tau/\alpha]\tau}{\psi;\Phi \vdash \mathtt{w}\!:\!\mu\alpha.\,\tau}\ (\text{REC})$$

$$\frac{\psi;\Phi \vdash \mathtt{w}\!:\!\tau}{\psi;\Phi \vdash \mathtt{w}\!:\!\tau^{\varphi}}\ (\text{INIT}) \qquad \frac{}{\psi;\Phi \vdash \mathtt{w}\!:\!\tau^{0}}\ (\text{UNINIT})$$

$$\frac{}{\vdash \tau^{\varphi} \le \tau^{\varphi}}\ (\text{REFL}) \qquad \frac{}{\vdash \tau^{1} \le \tau^{0}}\ (\text{0-1})$$

Figure 3.8: TAL typing rules for well-formed states

The lifting function $\llcorner\psi\lrcorner_\rho$ assigns a language id ρ to TAL, and packs each code specification in ψ with ρ into an OCAP specification π. The sink function $\ulcorner\Psi\urcorner^\rho$ collects from OCAP's Ψ the specifications of code certified in language ρ, and constructs a language-specific code heap specification.

As mentioned before, to embed TAL into OCAP, we need to discover the invariant enforced in TAL. TAL type systems essentially guarantees that, at any step of execution, the program state is well-typed with respect to the code heap type ψ and some register file type $[\Delta].\Gamma$. Judgment for well-typed state is represented as $\psi \vdash \mathbb{S}\!:\![\Delta].\,\Gamma$. TAL's typing rules for well-formed states and auxiliary judgments are shown in Figure 3.8. Our interpretation of TAL specifications encodes this invariant.

$$\begin{aligned}[\![\, [\Delta].\,\Gamma\,]\!]^{(\rho,r)}_{\mathcal{L}_{\text{TAL}}} \triangleq \lambda\Psi,\mathbb{S}.\ \exists\mathbb{H}_1,\mathbb{H}_2.\ \mathbb{S}.\mathbb{H} = \mathbb{H}_1 \uplus \mathbb{H}_2\ \wedge \\ (\ulcorner\Psi\urcorner^{\rho} \vdash (\mathbb{H}_1,\mathbb{S}.\mathbb{R}) : [\Delta].\,\Gamma\ )\ \wedge\ r\ \Psi\ \mathbb{H}_2\,. \end{aligned} \tag{3.13}$$

Here we use $\uplus$ to represent the union of two partial maps, which is defined only if these two partial maps have disjoint domains. We introduce the following formal definitions over heaps, which will also be used in the rest part of the thesis.

$$\mathbb{H}\bot\mathbb{H}' \triangleq \mathit{dom}(\mathbb{H}) \cap \mathit{dom}(\mathbb{H}') = \emptyset \tag{3.14}$$

$$\mathbb{H} \uplus \mathbb{H}' \triangleq \begin{cases} \mathbb{H} \cup \mathbb{H}' & \text{if } \mathbb{H}\bot\mathbb{H}' \\ \textit{undefined} & \text{otherwise} \end{cases} \tag{3.15}$$

The interpretation for TAL's specification $[\Delta].\,\Gamma$ takes a special open parameter r, in addition to the language ID and the meta-type $\mathcal{L}_{\text{TAL}}$ of $[\Delta].\,\Gamma$ in Coq. The assertion r is a predicate over data heaps and also takes Ψ as an extra argument. We use this extra r to specify the invariant of data heap used by TAL's runtime systems, such as memory management routines and garbage collectors. The heap used by runtime is separated from the heap used by TAL, and is invisible to TAL code. Note r does not specify register files. Although this may limit its expressiveness, it should be sufficient for runtime code because usually runtime does not reserve registers. The interpretation essentially says that the data heap in $\mathbb{S}$ can be split into two disjoint parts; one part is used by TAL and is well-formed with respect to TAL's typing rules; the other part is used by runtime code and satisfies the invariant r. Careful readers may feel strange that we use the TAL judgment $(\ulcorner\Psi\urcorner^{\rho} \vdash (\mathbb{H}_1,\mathbb{S}.\mathbb{R}) : [\Delta].\,\Gamma)$ as a proposition in our definition. This is well-defined because the interpretation is defined in our meta-logic, in which TAL's judgments are defined as propositions.

3.5.3 Soundness of the embedding

Theorem 3.13 states the soundness of TAL rules and the interpretation for TAL specifications. It shows that, given the interpretation, TAL rules are derivable as lemmas in OCAP. The soundness is independent with the open parameter r.

Theorem 3.13 (TAL Soundness)
For all ρ and r, let $\mathcal{D} = \{\rho \leadsto \langle \mathcal{L}_{\mathrm{TAL}},\ [\![\_]\!]^{(\rho,r)}_{\mathcal{L}_{\mathrm{TAL}}} \rangle\}$.

1. if we have $\psi \vdash \{[\Delta].\,\Gamma\}\,\mathbb{I}$ in TAL, then the OCAP judgment $\mathcal{D} \vdash \{\langle \mathtt{a} \rangle_{\Psi}\}\,\mathbb{I}$ holds, where $\mathtt{a} = [\![\,[\Delta].\,\Gamma\,]\!]^{(\rho,r)}_{\mathcal{L}_{\mathrm{TAL}}}$ and $\Psi = \llcorner \psi \lrcorner_{\rho}$;
2. if we have $\psi \vdash \mathbb{C} : \psi'$ in TAL, then the OCAP judgment $\mathcal{D}; \llcorner \psi \lrcorner_{\rho} \vdash \mathbb{C} : \llcorner \psi' \lrcorner_{\rho}$ holds.

This theorem shows the soundness of our embedding of TAL into OCAP. Given a module $\mathbb{C}$ certified in TAL, we can apply this theorem to convert it into a certified module in OCAP. On the other hand, it can also be viewed as a different way to prove the soundness of TAL inference rules by reducing TAL rules to OCAP rules.

Lemma 3.14 is essential to prove the soundness theorem. It shows the TAL subtyping relation is sound with respect to the interpretation. The whole proof of Theorem 3.13 has been formalized in the Coq proof assistant [31].

Lemma 3.14 (Subtyping Soundness)
For any ρ, r, $[\Delta].\,\Gamma$ and $[\Delta'].\,\Gamma'$, let $\mathtt{a} = [\![\,[\Delta].\,\Gamma\,]\!]^{(\rho,r)}_{\mathcal{L}_{\mathrm{TAL}}}$ and $\mathtt{a}' = [\![\,[\Delta'].\,\Gamma'\,]\!]^{(\rho,r)}_{\mathcal{L}_{\mathrm{TAL}}}$. If $\vdash [\Delta].\,\Gamma \leq [\Delta'].\,\Gamma'$ in TAL, we have $\mathtt{a} \Rightarrow \mathtt{a}'$.

3.6 Discussions and Summary

The work on the OCAP framework is closely related to work on FPCC. We have briefly discussed about existing work on FPCC at the beginning of this chapter. In this section, we give a detailed comparison of the OCAP framework and related work.

3.6.1 Related Work

Semantic approaches to FPCC. The semantic approach to FPCC [7, 8, 94] builds semantic models for types. Based on type definitions, typing rules in TAL are proved as lemmas. Our work is similar to this approach in the sense that a uniform assertion is used in the OCAP framework. Interpretations are used to map foreign specifications to OCAP assertions. Based on the interpretation, inference rules of foreign systems are proved as OCAP lemmas.

However, our interpretation does not have to be a semantic model of foreign specifications. For instance, when we embed TAL into OCAP, we simply use TAL's syntactic state typing as the interpretation for register file types. This makes our interpretation easier to define than semantic models, such as indexed models for mutable references [3, 9], which are heavyweight to use.

OCAP also uses a different specification for embedded code pointers than the step-indexed semantic model [8, 94] used in the Princeton FPCC. Following our previous work on CAP systems, specification of embedded code pointers is interpreted as a code label specified in the code heap specification Ψ. This approach allows our framework to support partial correctness of programs with respect to its specifications, as shown in Theorem 3.12.

The step-indexed model is designed specifically for type safety. A code pointer $\mathtt{f}$ with precondition a will be defined as:

$$\mathsf{codeptr}(\mathtt{f}, \mathsf{a}) \triangleq \lambda k, \mathbb{C}, \mathbb{S}.\ \forall \mathbb{S}'. \forall j < k.\ \mathsf{a}\ j\ \mathbb{C}\ \mathbb{S}' \rightarrow \mathsf{Safe}(j, (\mathbb{C}, \mathbb{S}', \mathtt{f}))\,.$$

where a is an indexed predicate over the code heap and state, and $\mathsf{Safe}(n, \mathbb{P})$ means $\mathbb{P}$ can execute at least n steps. It is unclear how Theorem 3.12 could be proved if this model is used: when we do an indirect jump to a code pointer $\mathsf{codeptr}(\mathbb{R}(\mathtt{r}), \mathsf{a})$, we do not know the relationship between "a" and the loop invariant assigned to $\mathbb{R}(\mathtt{r})$ in program specification Ψ (unless we sacrifice the support of separate verification of modules), because the definition of codeptr is independent with Ψ.

Syntactic approaches to FPCC. The OCAP framework is quite different from the original syntactic approach [46, 28] to FPCC. In the syntactic approach, TALs are designed for a higher-level abstract machine with its own mechanized syntactic soundness proof. FPCC is constructed by proving bisimulation between type safe TAL programs and real machine code. In our framework, we allow users to certify machine code directly, but still at a higher abstraction level in TAL. The soundness of TAL is shown by proving TAL instruction rules as lemmas in OCAP. Runtime code for TAL is certified in a different system and is linked with TAL code in OCAP.

Hamid and Shao [45] shows how to interface XTAL with CAP. XTAL supports stubs which encapsulate interfaces of runtime library. Actual implementation of library is certified in CAP. Our work on linking TAL with runtime is similar to theirs, but with several differences. XTAL is also defined for a higher-level abstract machine. With stubs, the machine does not have a self-contained operational semantics. They present XTAL as a stand alone system with syntactic soundness proof. Our TAL is just a set of rules which is proved as OCAP lemmas under appropriate interpretations. It does not even have a top PROG rule for complete programs. In Hamid and Shao's framework, CAP serves two roles from our point of view: the underlying framework (like OCAP) and the system to certify runtime (like our use of SCAP). Both OCAP and SCAP have better support of modularity than CAP. By splitting the underlying framework and the system to certify runtime, our work is more general and conceptually clearer.

Previous work on CAP systems. CAP is first used in [101] to certify `malloc/free` libraries. The system used there does not have modular support of embedded code pointers. Ni and Shao [75] solved this problem in XCAP by defining a specification language with a built-in construct for code pointers. XCAP specifications are interpreted into a predicate taking Ψ as argument. This approach is extended in [35] to support single or fixed combinations of specification languages, which is not open and extensible. OCAP is built upon previous work, but it is the first framework we use to support interoperability of different systems in an extensible and systematic way. All our previous CAP systems can be trivially embedded in OCAP, as discussed in section 3.1.

The open verifier framework. Chang *et al.* proposed an open verifier for verifying untrusted code [19]. Their framework can be customized by embedding extension modules, which are executable verifiers implementing verification strategies in pre-existing systems. However, the paper does not show how multiple extension modules can coexist and collaborate in the framework. Especially, since their support of indirect jumps needs to know all the possible target addresses, it is unclear how they support separate verification of program modules using different extensions. Open Verifier emphasizes on implementation issues for practical proof construction, while our work explores the generality of FPCC frameworks. OCAP provides a formal basis with clear meta properties for interoperation between verification systems.

XCAP and TAL. Ni and Shao [76] showed a translation from TAL to XCAP+, an extension of XCAP [75] with recursive types and mutable reference types. There translation has a semantic flavor in the sense that primitive TAL types is translated into a predicate in XCAP+'s assertion language. Our embedding of TAL into OCAP is more like the syntactic approach taken by Hamid and Shao [45], which does not have an interpretation of primitive types and uses the type judgment as propositions instead.

3.6.2 Summary

In this chapter, we propose OCAP as an open framework for constructing foundational certified software packages. OCAP is the first framework which systematically supports interoperation of different verification systems. It lays a set of Hoare-style inference rules above the raw machine semantics, so that proofs can be constructed following these rules instead of directly using the mechanized meta-logic. As the foundational layer, the assertion language for OCAP is expressive enough to specify the invariants enforced in foreign verification systems. Soundness of these rules are proved in the Coq proof assistant with machine-checkable proofs [31], therefore these rules are not trusted. OCAP is modular, extensible and expressive, therefore it satisfies all the requirements mentioned above for an open framework. We have shown the embedding of a typed assembly language into OCAP. More applications will be shown in the next several chapters.

Chapter 4

Stack-Based Reasoning of Sequential Code

As in CAP, control flows at the assembly level are usually reasoned about following the continuation passing style (CPS) [5], because of the lack of abstractions. CPS treats all control flows, such as return addresses of functions, as first-class code pointers. Although general, it is too low-level and cannot be used to easily describe stack invariants preserved in the implementations of normal control abstractions.

Runtime stacks are critical components of any modern software—they are used to implement powerful control structures such as procedure call/return, tail call [92, 21], C-style `setjmp/longjmp` [61], stack cutting and unwinding (for handling exceptions) [20, 30, 85], coroutines [23], and thread context switch [47]. Correct implementation of these constructs is of utmost importance to the safety and reliability of many software systems.

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* [85]). Neither C nor C-- [85] 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 difficult 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, Figure 2.4 shows MIPS style assembly code compiled from the C code shown in Section 2.2. 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.

Under CPS, the code following `ct` (lines 6–9) in Figure 2.4 is treated not as part of the 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 gives a uniform model of all controls, but it is still hard to describe the above invariant about `f` and `ct`. Indeed, none of the existing PCC systems [73, 74, 22, 6, 46, 75] have successfully certified `setjmp/longjmp`, weak continuations, and general stack cutting and unwinding.

In this chapter, we propose SCAP (Stack-based CAP), a program logic that can expose and validate the invariants of stack-based control abstractions. We will 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. We also embed SCAP into the OCAP framework by proving SCAP inference rules as lemmas that can be derived from OCAP. This research is based on joint work I have done with Zhong Shao, Alexander Vaynberg, Sen Xiang, Zhaozhong Ni and Yu Guo [35, 33].

4.1 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* [65], 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) [85]. *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 [85] 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 [61] 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 [89] 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. We will study non-preemptive threads, which may be viewed as a generalization of coroutines, in the next chapter.

4.1.1 Reasoning about Control Abstractions

Traditional Hoare-logic [50] uses the pre- and postcondition as specifications for programs. Most work on Hoare-logic [10] reasons 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 [97, 11]; 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 [91] also observed two similar challenges for type-checking 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 [73]. 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" [68, 75]), 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 [71, 6, 75] to model sophisticated stack-based invariants.

4.1.2 The SCAP 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

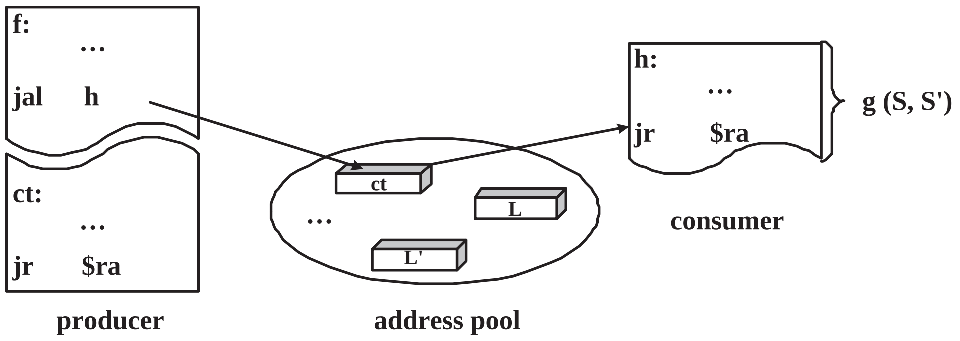


Figure 4.1: The Model for Code Pointers

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 4.1, the caller is the "producer" and the callee is the "consumer", while the return address is the continuation pointer.

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 chapter, we will describe the invariant for each control abstraction.

$$
\begin{array}{rlll}
(\textit{LocalSpec}) & \psi & ::= & \{(\mathtt{f}_1, (\mathtt{p}_1, \mathtt{g}_1)), \ldots, (\mathtt{f}_n, (\mathtt{p}_n, \mathtt{g}_n))\} \\
(\textit{StatePred}) & \mathtt{p} & \in & \textit{State} \rightarrow \mathsf{Prop} \\
(\textit{Guarantee}) & \mathtt{g} & \in & \textit{State} \rightarrow \textit{State} \rightarrow \mathsf{Prop}
\end{array}
$$

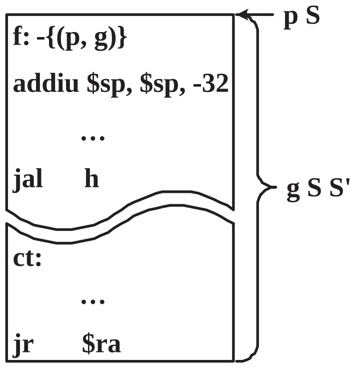
Figure 4.2: SCAP Specification Constructs

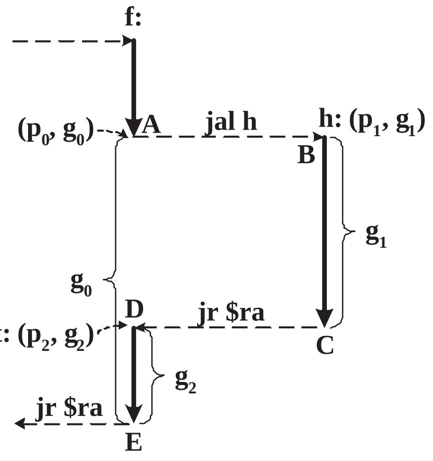
We also present a set of inference rules (as OCAP lemmas) to allow programmers to verify structureless assembly code with higher-level abstractions.

4.2 Stack-Based Reasoning for Function Call

SCAP is proposed to certify machine programs defined in Section 2.2, where the basic block of code is a sequence of instructions ending with a jump instruction. However, SCAP also uses the concept of "functions", which corresponds to functions in higher-level languages (*e.g.,* C functions). A *function* in our machine is a set of instruction sequences compiled down from a higher-level function. For instance, in Figure 2.4 (Section 2.2) the instruction sequences labeled by `f` and `ct` make up the machine code for the C function `f`. Note that "functions" in in our machine is purely a logical concept for reasoning. SCAP does not require any change of our machine program structures, such as labeling of instruction sequences with the corresponding function names. Also, one instruction sequence may logically belong to multiple functions.

The specification constructs of SCAP are shown in Figure 4.2. The SCAP specification ψ of the code heap is a collection of assertions associated with code blocks. Since we will embed SCAP into OCAP, we call ψ a "local specification" to distinguish it from the OCAP code heap specification Ψ. Similar to Ψ, ψ is a binary relation between code labels and specifications (in the form of $(\mathtt{p}, \mathtt{g})$ pairs) of code blocks. The assertion $\mathtt{p}$ is a predicate over a program state $\mathbb{S}$. As preconditions in normal Hoare logic, it specifies the requirement over the program state at the specified program point, which needs to ensure safe execution of the remaining code. The assertion $\mathtt{g}$ is a predicate over a pair of





(a) (b)

Figure 4.3: The Model for Function Call/Return in SCAP

states. It specifies a transition from one state (the first argument) to the other (the second argument). In SCAP, g is called a *guarantee*, which specifies the expected state transition made by the remaining part of the current function, *i.e.,* the behavior of the computation from the specified program point to the return point of the current function. Figure 4.3(a) shows the meaning of the specification $(\mathtt{p}, \mathtt{g})$ for the function `f` defined in Figure 2.4. Note that g may cover multiple instruction sequences. If a function has multiple return points, g governs all the traces from the current program point to any return point.

Figure 4.3(b) illustrates a function call of `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.,*

$$\forall \mathbb{H}, \mathbb{R}.\ \mathtt{p}_0\ (\mathbb{H}, \mathbb{R}) \rightarrow \mathtt{p}_1\ (\mathbb{H}, \mathbb{R}\{\mathtt{\$ra} \rightsquigarrow \mathtt{ct}\});$$

- after `h` returns, `f` can resume its execution from point `D`, *i.e.,*

$$\forall \mathbb{H}, \mathbb{R}, \mathbb{S}'.\ \mathtt{p}_0\ (\mathbb{H}, \mathbb{R}) \rightarrow \mathtt{g}_1\ (\mathbb{H}, \mathbb{R}\{\mathtt{\$ra} \rightsquigarrow \mathtt{ct}\})\ \mathbb{S}' \rightarrow \mathtt{p}_2\ \mathbb{S}';$$

- if the function `h` and the code segment `D-E` satisfy their specifications, the specification for `A-E` is satisfied, *i.e.*,

$$\forall \mathbb{H}, \mathbb{R}, \mathbb{S}', \mathbb{S}''.\ \mathtt{p}_0\ (\mathbb{H}, \mathbb{R}) \rightarrow \mathtt{g}_1\ (\mathbb{H}, \mathbb{R}\{\mathtt{\$ra} \rightsquigarrow \mathtt{ct}\})\ \mathbb{S}' \rightarrow \mathtt{g}_2\ \mathbb{S}'\ \mathbb{S}'' \rightarrow \mathtt{g}_0\ (\mathbb{H}, \mathbb{R})\ \mathbb{S}'';$$

- the function `h` must reinstate the return code pointer when it returns, *i.e.*,

$$\forall \mathbb{S}, \mathbb{S}'.\ \mathtt{g}_1\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathbb{S}.\mathbb{R}(\mathtt{\$ra}) = \mathbb{S}'.\mathbb{R}(\mathtt{\$ra}).$$

Above conditions are enforced by the CALL rule shown in Figure 4.4.

To check the well-formedness of an instruction sequence beginning with ι, the programmer needs to find an intermediate specification $(\mathtt{p}', \mathtt{g}')$, which serves both as the postcondition for ι and as the precondition for the remaining instruction sequence. As shown in the 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 ι; and
- if the remaining instruction sequence satisfies its guarantee $\mathtt{g}'$, the original instruction sequence satisfies $\mathtt{g}$.

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 $\mathtt{g}$ (*i.e.*, $\mathtt{g}\ \mathbb{S}_0\ \mathbb{S}_n$), we enforce the following chain of implication relations:

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

where each $\mathtt{g}_i$ 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.*, $\mathtt{g}_n\ \mathbb{S}_{n-1}\ \mathbb{S}_n$)

$\boxed{\psi \vdash \{(\mathtt{p},\mathtt{g})\}\,\mathtt{f} : \mathbb{I}}$ **(*Well-formed instruction sequence*)**

$$\frac{\begin{array}{l}(\mathtt{f}',(\mathtt{p}',\mathtt{g}')) \in \psi \qquad\qquad (\mathtt{f}{+}1,(\mathtt{p}'',\mathtt{g}'')) \in \psi \\ \forall \mathbb{H},\mathbb{R}.\ \mathtt{p}\ (\mathbb{H},\mathbb{R}) \rightarrow \mathtt{p}'\ (\mathbb{H},\mathbb{R}\{\$\mathsf{ra}{\leadsto}\mathtt{f}{+}1\}) \\ \forall \mathbb{H},\mathbb{R},\mathbb{S}'.\ \mathtt{p}\ (\mathbb{H},\mathbb{R}) \rightarrow \mathtt{g}'\ (\mathbb{H},\mathbb{R}\{\$\mathsf{ra}{\leadsto}\mathtt{f}{+}1\})\ \mathbb{S}' \rightarrow \\ \qquad\qquad (\mathtt{p}''\ \mathbb{S}' \wedge (\forall \mathbb{S}''.\ \mathtt{g}''\ \mathbb{S}'\ \mathbb{S}'' \rightarrow \mathtt{g}\ (\mathbb{H},\mathbb{R})\ \mathbb{S}'')) \\ \forall \mathbb{S},\mathbb{S}'.\ \mathtt{g}'\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathbb{S}.\mathbb{R}(\$\mathsf{ra}) = \mathbb{S}'.\mathbb{R}(\$\mathsf{ra})\end{array}}{\psi \vdash \{(\mathtt{p},\mathtt{g})\}\,\mathtt{f} : \mathsf{jal}\ \mathtt{f}'}\ (\textsc{call})$$

$$\frac{\begin{array}{l}\iota \in \{\mathsf{addu}, \mathsf{addiu}, \mathsf{lw}, \mathsf{subu}, \mathsf{sw}\} \\ \psi \vdash \{(\mathtt{p}',\mathtt{g}')\}\,\mathtt{f}{+}1 : \mathbb{I} \qquad \forall \mathbb{S}.\ \mathtt{p}\ \mathbb{S} \rightarrow \mathtt{p}'\ \mathsf{NextS}_{(\mathtt{f},\,\iota)}(\mathbb{S}) \\ \forall \mathbb{S},\mathbb{S}'.\ \mathtt{p}\ \mathbb{S} \rightarrow \mathtt{g}'\ \mathsf{NextS}_{(\mathtt{f},\,\iota)}(\mathbb{S})\ \mathbb{S}' \rightarrow \mathtt{g}\ \mathbb{S}\ \mathbb{S}'\end{array}}{\psi \vdash \{(\mathtt{p},\mathtt{g})\}\,\mathtt{f} : \iota;\mathbb{I}}\ (\textsc{scap-seq})$$

$$\frac{\forall \mathbb{S}.\ \mathtt{p}\ \mathbb{S} \rightarrow \mathtt{g}\ \mathbb{S}\ \mathbb{S}}{\psi \vdash \{(\mathtt{p},\mathtt{g})\}\,\mathtt{f} : \mathsf{jr}\ \$\mathsf{ra}}\ (\textsc{ret})$$

$$\frac{\begin{array}{l}(\mathtt{f}',(\mathtt{p}',\mathtt{g}')) \in \psi \\ \forall \mathbb{S}.\ \mathtt{p}\ \mathbb{S} \rightarrow \mathtt{p}'\ \mathbb{S} \quad\ \forall \mathbb{S},\mathbb{S}'.\ \mathtt{p}\ \mathbb{S} \rightarrow \mathtt{g}'\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathtt{g}\ \mathbb{S}\ \mathbb{S}'\end{array}}{\psi \vdash \{(\mathtt{p},\mathtt{g})\}\,\mathtt{f} : \mathsf{j}\ \mathtt{f}'}\ (\textsc{t-call})$$

$$\frac{\begin{array}{l}(\mathtt{f}',(\mathtt{p}'',\mathtt{g}'')) \in \psi \qquad\qquad \psi \vdash \{(\mathtt{p}',\mathtt{g}')\}\,\mathtt{f}{+}1 : \mathbb{I} \\ \forall \mathbb{S}.\ \mathtt{p}\ \mathbb{S} \rightarrow \mathbb{S}.\mathbb{R}(\mathtt{r}_s) \neq \mathbb{S}.\mathbb{R}(\mathtt{r}_t) \rightarrow (\mathtt{p}'\ \mathbb{S} \wedge (\forall \mathbb{S}'.\ \mathtt{g}'\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathtt{g}\ \mathbb{S}\ \mathbb{S}')) \\ \forall \mathbb{S}.\ \mathtt{p}\ \mathbb{S} \rightarrow \mathbb{S}.\mathbb{R}(\mathtt{r}_s) = \mathbb{S}.\mathbb{R}(\mathtt{r}_t) \rightarrow (\mathtt{p}''\ \mathbb{S} \wedge (\forall \mathbb{S}'.\ \mathtt{g}''\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathtt{g}\ \mathbb{S}\ \mathbb{S}'))\end{array}}{\psi \vdash \{(\mathtt{p},\mathtt{g})\}\,\mathtt{f} : \mathsf{beq}\ \mathtt{r}_s,\mathtt{r}_t,\mathtt{f}';\mathbb{I}}\ (\textsc{scap-beq})$$

$$\frac{\begin{array}{l}(\mathtt{f}',(\mathtt{p}'',\mathtt{g}'')) \in \psi \qquad \psi \vdash \{(\mathtt{p}',\mathtt{g}')\}\,\mathtt{f}{+}1 : \mathbb{I} \\ \forall \mathbb{S}.\ \mathtt{p}\ \mathbb{S} \rightarrow \mathbb{S}.\mathbb{R}(\mathtt{r}_s) \leq 0 \rightarrow (\mathtt{p}'\ \mathbb{S} \wedge (\forall \mathbb{S}'.\ \mathtt{g}\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathtt{g}\ \mathbb{S}\ \mathbb{S}')) \\ \forall \mathbb{S}.\ \mathtt{p}\ \mathbb{S} \rightarrow \mathbb{S}.\mathbb{R}(\mathtt{r}_s) > 0 \rightarrow (\mathtt{p}''\ \mathbb{S} \wedge (\forall \mathbb{S}'.\ \mathtt{g}''\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathtt{g}\ \mathbb{S}\ \mathbb{S}'))\end{array}}{\psi \vdash \{(\mathtt{p},\mathtt{g})\}\,\mathtt{f} : \mathsf{bgtz}\ \mathtt{r}_s,\mathtt{f}';\mathbb{I}}\ (\textsc{scap-bgtz})$$

$\boxed{\psi \vdash \mathbb{C} : \psi'}$ **(*Well-formed code heap*)**

$$\frac{\text{for all } (\mathtt{f},(\mathtt{p},\mathtt{g})) \in \psi' \text{:} \quad \psi \vdash \{(\mathtt{p},\mathtt{g})\}\,\mathtt{f} : \mathbb{C}[\mathtt{f}]}{\psi \vdash \mathbb{C} : \psi'}\ (\textsc{scap-cdhp})$$

Figure 4.4: SCAP Inference Rules

```
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 4.5: Factorial Functions in C

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 g $\mathbb{S}_0$ $\mathbb{S}_n$.

SCAP also supports tail function calls, where the callee reuses the caller's stack frame and the return address. 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.

The SCAP-CDHP rule checks the well-formedness of SCAP program modules. The code heap $\mathbb{C}$ is well-formed if and only if each instruction sequence is well-formed under SCAP.

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 4.5 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.

$$
\begin{aligned}
\mathsf{g}_0 &\triangleq \exists n.\ (\$\mathsf{a0}\!=\!n) \wedge (\$\mathsf{v0}'\!=\!n!) \wedge \mathsf{Rid}(\{\$\mathsf{gp}, \$\mathsf{sp}, \$\mathsf{fp}, \$\mathsf{ra}, \$\mathsf{s0}, \ldots, \$\mathsf{s7}\}) \\
&\qquad\qquad \wedge \mathsf{Hnid}(\{(\$\mathsf{sp} - 3\!\times\! n - 2), \ldots, \$\mathsf{sp}\}) \\
\mathsf{g}_1 &\triangleq \exists n.\ (\$\mathsf{a0}\!=\!n) \wedge (\$\mathsf{v0}'\!=\!n!) \wedge \mathsf{Rid}(\{\$\mathsf{gp}, \$\mathsf{s1}, \ldots, \$\mathsf{s7}\}) \wedge \mathsf{g}_{\mathrm{frm}} \\
&\qquad\qquad \wedge \mathsf{Hnid}(\{(\$\mathsf{sp} - 3\!\times\! n + 1), \ldots, \$\mathsf{sp}\}) \\
\mathsf{g}_3 &\triangleq (\$\mathsf{v0}' = \$\mathsf{v0} \times \$\mathsf{s0}) \wedge \mathsf{Rid}(\{\$\mathsf{gp}, \$\mathsf{s1}, \ldots, \$\mathsf{s7}\}) \wedge \mathsf{g}_{\mathrm{frm}} \wedge \mathsf{Hnid}(\emptyset) \\
\mathsf{g}_4 &\triangleq \mathsf{Rid}(\{\$\mathsf{gp}, \$\mathsf{v0}, \$\mathsf{s1}, \ldots, \$\mathsf{s7}\}) \wedge \mathsf{g}_{\mathrm{frm}} \wedge \mathsf{Hnid}(\emptyset)
\end{aligned}
$$

```
prolog:  -{(TRUE, g_0)}
         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, g_1)}
         bgtz   $a0, nonzero        ;n == 0
         addiu  $v0, $zero, 1       ;return 1
         j      epilog

nonzero: -{(p_1 ∧ ($a1 > 0), g_1)}
         addiu  $s0, $a0, 0         ;save n
         addiu  $a0, $a0, -1        ;n--
         jal    prolog, cont        ;fact(n)

cont:    -{($v0 = ($s0 - 1)!, g_3)}
         multu  $v0, $s0, $v0       ;return n*(n-1)!
         j      epilog

epilog:  -{($v0 = ($s0 - 1)!, g_3)}
         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 4.6: SCAP Factorial Example

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

```
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 4.7: SCAP Implementation of Tail Recursion

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

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

Figure 4.8: Macros for SCAP Examples

The compiled assembly code of these two functions is shown in Figure 4.6 and 4.7. 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 4.8 shows definitions of macros used in the code specifications. To simplify the presentation, we use the register name `r` and [`l`] to represent values contained in `r` and the memory location `l`, respectively. 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 4.6. The precondition defines no constraint on the value of `$ra`. The guarantee g_0 specifies the behavior of the function:

- the return value `$v0` is the factorial of the argument `$a0`;
- callee-save registers are not updated; and
- the memory, other than the stack frames, is 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.

In the second implementation (in Figure 4.7), the caller passes the address of a *stack variable* to the function `fact`. The tail recursion is optimized by reusing the stack frame

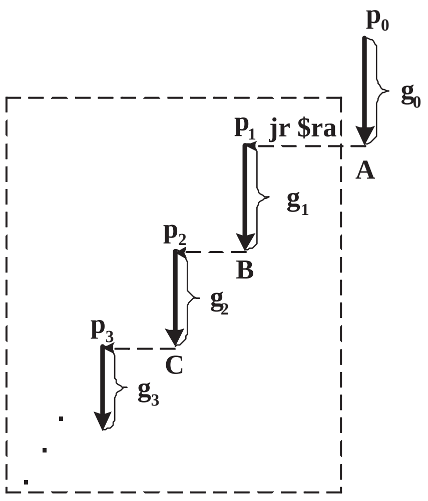


Figure 4.9: The Logical Control Stack

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 $ra. However, SCAP will reject any code trying to call `ld_vir`, because the g cannot satisfy the premises of the CALL rule.

4.4 The Stack Invariant

Figure 4.9 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.

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:

$$\begin{aligned}
&\mathrm{g}_0\ \mathbb{S}_0\ \mathbb{S}_1 \rightarrow \\
&\qquad ((\mathbb{S}_1.\mathbb{R}(\$\mathsf{ra}), (\mathrm{p}_1, \mathrm{g}_1)) \in \psi) \wedge \mathrm{p}_1\ \mathbb{S}_1; \\
&\mathrm{g}_0\ \mathbb{S}_0\ \mathbb{S}_1 \rightarrow \mathrm{g}_1\ \mathbb{S}_1\ \mathbb{S}_2 \rightarrow \\
&\qquad ((\mathbb{S}_2.\mathbb{R}(\$\mathsf{ra}), (\mathrm{p}_2, \mathrm{g}_2)) \in \psi) \wedge \mathrm{p}_2\ \mathbb{S}_2; \\
&\mathrm{g}_0\ \mathbb{S}_0\ \mathbb{S}_1 \rightarrow \mathrm{g}_1\ \mathbb{S}_1\ \mathbb{S}_2 \rightarrow \mathrm{g}_2\ \mathbb{S}_2\ \mathbb{S}_3 \rightarrow \\
&\qquad ((\mathbb{S}_3.\mathbb{R}(\$\mathsf{ra}), (\mathrm{p}_3, \mathrm{g}_3)) \in \psi) \wedge \mathrm{p}_3\ \mathbb{S}_3; \\
&\ldots
\end{aligned}$$

where ψ is the program specification, and each $\mathbb{S}_i$ $(i > 0)$ is implicitly quantified by universal quantification.

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

$$
\begin{array}{l}
\mathsf{wfst}(0, \mathsf{g}, \mathbb{S}, \psi) \quad\quad \triangleq \ \neg\exists\mathbb{S}'.\ \mathsf{g}\ \mathbb{S}\ \mathbb{S}' \\
\mathsf{wfst}(n{+}1, \mathsf{g}, \mathbb{S}, \psi) \triangleq \\
\quad \forall\mathbb{S}'.\ \mathsf{g}\ \mathbb{S}\ \mathbb{S}' \rightarrow \exists(\mathsf{p}', \mathsf{g}').\ (\mathbb{S}'.\mathbb{R}(\$\mathsf{ra}), (\mathsf{p}', \mathsf{g}')) \in \psi \wedge \mathsf{p}'\ \mathbb{S}' \wedge \mathsf{wfst}(n, \mathsf{g}', \mathbb{S}', \psi)
\end{array}
$$

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.*, $\neg\exists\mathbb{S}'.\ \mathsf{g}\ \mathbb{S}\ \mathbb{S}'$.

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:

$$
\mathsf{p}\ \mathbb{S} \wedge \exists n.\mathsf{wfst}(n, \mathsf{g}, \mathbb{S}, \psi)\,.
$$

Note here we do not need to know the exact depth n of the control stack.

To prove the soundness of SCAP, we need to prove that the invariant holds at every step of program execution. The stack invariant also explains why we can have such a simple RET rule, which "type-checks" the "jr \$ra" instruction without requiring that \$ra contain a valid code pointer.

4.5 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 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 [89].

4.5.1 SCAP-I

In SCAP, a function call is made by executing the jal instruction. The callee can only return to the "correct" return address, *i.e.,* the program counter of the next instruction following the corresponding jal instruction in the caller. This is enforced by the constraint $\forall \mathbb{S}, \mathbb{S}'.\ \mathsf{g}'\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathbb{S}.\mathbb{R}(\$\mathsf{ra}) = \mathbb{S}'.\mathbb{R}(\$\mathsf{ra})$ in the CALL rule. To allow the callee to return to multiple locations, we simply remove this constraint. We can also decouple the saving of the return address and the jump to the callee. The function call can be made by the normal jump instruction j `f`, as long as the return address is already saved before the function call. Now we have a more relaxed rule for function call:

$$\frac{\begin{array}{ll} \forall \mathbb{S}.\ \mathtt{p}\ \mathbb{S} \rightarrow \mathtt{p}'\ \mathbb{S} & (\mathtt{f}', (\mathtt{p}', \mathtt{g}')) \in \psi \\ \forall \mathbb{S}, \mathbb{S}'.\ \mathtt{p}\ \mathbb{S} \rightarrow \mathtt{g}'\ \mathbb{S}\ \mathbb{S}' \rightarrow & \\ \quad \exists \mathtt{p}'', \mathtt{g}''.\ ((\mathbb{S}'.\mathbb{R}(\$\mathsf{ra}), (\mathtt{p}'', \mathtt{g}'')) \in \psi) \wedge \mathtt{p}''\ \mathbb{S}' \wedge (\forall \mathbb{S}''.\ \mathtt{g}''\ \mathbb{S}'\ \mathbb{S}'' \rightarrow \mathtt{g}\ \mathbb{S}\ \mathbb{S}'') & \end{array}}{\psi \vdash \{(\mathtt{p}, \mathtt{g})\}\ \mathtt{f} : \mathsf{j}\ \mathtt{f}'} \ (\textsc{call-i})$$

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

Examples of Multi-Return Function Call

SCAP-I can certify the compiled C-- code with stack unwinding. C-- uses the primitive "`return <n/m>`" to allow a function to return to the $\mathtt{n}^{th}$ of `m` return continuations defined in the caller. A normal return is written as "`return <m/m>`", while `n` being less than `m` means an "abnormal" return. Correspondingly, at the call cite, the caller put the annotation such as "`also returns to k0, k1`", where continuations `k0` and `k1` are defined in

```
rev(bits32 x){                    cmp(bits32 x){
  cmp(x) also returns to k;         if (x = 0)
  return 0;                           return <0/1>;
                                    else
  continuation k:                     return <1/1>;
    return 1;                     }
}
```

Figure 4.10: C-- Code with Multi-Return Address

the same function as the call site.

In Figure 4.10 we show a simple C-- program which returns 1 if the argument is 0 and returns 0 otherwise. We illustrate in Figure 4.11 how to use SCAP-I to certify the compiled code. The precondition of the `rev` function is simply set to TRUE, while $\mathtt{g}_0$ specifies the relationship between the argument `$a0` and the return value `$v0`, and the preservation of callee save registers and memory except for the space for the stack frame. The precondition for the `cmp` function is TRUE, and the guarantee $\mathtt{g}_5$ says that the function returns to different addresses under different conditions.

At the point where `cmp` is called, we need to specify in the precondition $\mathtt{p}_1$ that both return addresses are valid code labels (*i.e.,* `ct1` and `k`). The guarantee $\mathtt{g}_1$ specifies the behavior of the remaining code under two different conditions, while $(\mathtt{p}_2, \mathtt{g}_2)$ and $(\mathtt{p}_3, \mathtt{g}_3)$ specify the two different return continuations. Interested readers can check that the specifications satisfy the constraint enforced by the CALL-I rule. Specifications for other code blocks are straightforward and are omitted here.

4.5.2 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

$$
\begin{array}{ll}
g_{\mathrm{frm}} & \triangleq \mathsf{\$sp}' = \mathsf{\$sp} + 2 \wedge \mathsf{\$fp}' = \mathsf{Frm}[0] \wedge \mathsf{\$ra}' = \mathsf{Frm}[1] \\
g_0 & \triangleq (\mathsf{\$a0} = 0 \rightarrow \mathsf{\$v0}' = 1) \wedge (\mathsf{\$a0} \neq 0 \rightarrow \mathsf{\$v0}' = 0) \\
 & \qquad \wedge \mathsf{Rid}(\{\mathsf{\$gp}, \mathsf{\$sp}, \mathsf{\$fp}, \mathsf{\$ra}, \mathsf{\$s0}, \ldots, \mathsf{\$s7}\}) \\
 & \qquad \wedge \mathsf{Hnid}(\{\mathsf{\$sp} - 1, \mathsf{\$sp}\}) \\
p_1 & \triangleq \mathsf{\$ra} = \mathsf{ct1} \wedge \mathsf{\$a1} = \mathrm{k} \\
g_1 & \triangleq (\mathsf{\$a0} = 0 \rightarrow \mathsf{\$v0}' = 1) \wedge (\mathsf{\$a0} \neq 0 \rightarrow \mathsf{\$v0}' = 0) \\
 & \qquad \wedge \mathsf{Rid}(\{\mathsf{\$gp}, \mathsf{\$s0}, \ldots, \mathsf{\$s7}\}) \wedge g_{\mathrm{frm}} \wedge \mathsf{Hnid}(\emptyset) \\
g_2 & \triangleq \mathsf{\$v0}' = 0 \wedge \mathsf{Rid}(\{\mathsf{\$gp}, \mathsf{\$s0}, \ldots, \mathsf{\$s7}\}) \wedge g_{\mathrm{frm}} \wedge \mathsf{Hnid}(\emptyset) \\
g_3 & \triangleq \mathsf{\$v0}' = 1 \wedge \mathsf{Rid}(\{\mathsf{\$gp}, \mathsf{\$s0}, \ldots, \mathsf{\$s7}\}) \wedge g_{\mathrm{frm}} \wedge \mathsf{Hnid}(\emptyset) \\
g_5 & \triangleq (\mathsf{\$a0} = 0 \rightarrow \mathsf{\$ra}' = \mathsf{\$a1}) \wedge (\mathsf{\$a0} \neq 0 \rightarrow \mathsf{\$ra}' = \mathsf{\$ra}) \\
 & \qquad \wedge \mathsf{Rid}(\{\mathsf{\$gp}, \mathsf{\$sp}, \mathsf{\$fp}, \mathsf{\$s0}, \ldots, \mathsf{\$s7}\}) \wedge \mathsf{Hnid}(\emptyset)
\end{array}
$$

```
rev:    -{(TRUE, g0)}
        addiu  $sp, $sp, -2       ;allocate frame
        sw     $fp, 2($sp)        ;save old $fp
        addiu  $fp, $sp, 2        ;new $fp
        sw     $ra, -1($fp)       ;save old $ra
        addiu  $ra, $zero, ct1    ;ret cont 1
        addiu  $a1, $zero, k      ;ret cont 0
        -{(p1, g1)}
        j      cmp                ;call cmp with k

ct1:    -{(TRUE, g2)}
        addiu  $v0, $zero, 0      ;$v0 = 0
        j      epilog

k:      -{(TRUE, g3)}
        addiu  $v0, $zero, 1      ;$v0 = 1
        j      epilog

epilog: -{(p4, g4)}
        lw     $ra, -1($fp)       ;restore $ra
        lw     $fp, 0($fp)        ;restore $fp
        addiu  $sp, $sp, 2        ;restore $sp
        jr     $ra                ;return

cmp:    -{(TRUE, g5)}
        beq    $a0, $zero, eqz
        jr     $ra                ;return <1/1>
eqz:    -{(p6, g6)}
        addu   $ra, $zero, $a1    ;return <0/1>
        jr     $ra
```

Figure 4.11: Assembly Code with Multi-Return Function Call

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.

$$\frac{\begin{array}{ll}\forall \mathbb{S}.\ \mathsf{p}\ \mathbb{S} \rightarrow \mathsf{p}'\ \mathbb{S} & (\mathtt{f}', (\mathsf{p}', \mathsf{g}')) \in \psi \\ \forall \mathbb{S}, \mathbb{S}'.\ \mathsf{p}\ \mathbb{S} \rightarrow \mathsf{g}'\ \mathbb{S}\ \mathbb{S}' \rightarrow & \\ \quad (\mathsf{g}\ \mathbb{S}\ \mathbb{S}' \vee & \\ \quad\ \exists \mathsf{p}'', \mathsf{g}''.\ (\mathbb{S}'.\mathbb{R}(\mathsf{\$ra}), (\mathsf{p}'', \mathsf{g}'')) \in \psi \wedge \mathsf{p}''\ \mathbb{S}' \wedge (\forall \mathbb{S}''.\ \mathsf{g}''\ \mathbb{S}'\ \mathbb{S}'' \rightarrow \mathsf{g}\ \mathbb{S}\ \mathbb{S}'')) & \end{array}}{\psi \vdash \{(\mathsf{p}, \mathsf{g})\}\ \mathtt{f} : \mathtt{j}\ \mathtt{f}'}\ (\textsc{call-ii})$$

In the CALL-II rule above, 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.

$$\begin{array}{l} \mathsf{wfst}'(0, \mathsf{g}, \mathbb{S}, \psi) \triangleq\ \neg\exists \mathbb{S}'.\ \mathsf{g}\ \mathbb{S}\ \mathbb{S}' \\ \mathsf{wfst}'(n{+}1, \mathsf{g}, \mathbb{S}, \psi) \triangleq \\ \quad \forall \mathbb{S}'. \mathsf{g}\ \mathbb{S}\ \mathbb{S}' \rightarrow \\ \qquad \exists \mathsf{p}', \mathsf{g}'.\ ((\mathbb{S}'.\mathbb{R}(\mathsf{\$ra}), (\mathsf{p}', \mathsf{g}')) \in \psi) \wedge \mathsf{p}'\ \mathbb{S}' \wedge \exists m \leq n.\ \mathsf{wfst}'(m, \mathsf{g}', \mathbb{S}', \psi) \end{array}$$

The rest of SCAP-II inference rules are the same with those in SCAP. When a function makes an indirect jump to a weak continuation instead of a normal return continuation in SCAP, we use the same RET rule in SCAP, as shown below. Here we give it a new name, JWC (jump to weak continuations), to show that $ra contains a weak continuation pointer

```
rev(bits32 x){                      cmp0(bits32 x, bits32 k){
  cmp0(x, k) also cuts to k;          cmp1(x, k);
  return 0;                         }

  continuation k:                   cmp1(bits32 x, bits32 k){
    return 1;                         if (x == 0) cut to k;
}                                     return;
                                    }
```

Figure 4.12: C-- Code with Weak-Continuations

$$
\begin{array}{lll}
p_0 & \triangleq & \mathsf{TRUE} \\
g_0 & \triangleq & (\$\mathsf{a0} = 0 \rightarrow \$\mathsf{v0}' = 1) \wedge (\$\mathsf{a0} \neq 0 \rightarrow \$\mathsf{v0}' = 0) \wedge \\
& & \quad \wedge \mathsf{Rid}(\{\$\mathsf{gp}, \$\mathsf{sp}, \$\mathsf{fp}, \$\mathsf{ra}, \$\mathsf{s0}, \ldots, \$\mathsf{s7}\}) \\
& & \quad \wedge \mathsf{Hnid}(\{\$\mathsf{sp} - 3, \ldots, \$\mathsf{sp}\}) \\
p_1 & \triangleq & (\$\mathsf{ra} = \mathtt{ct}) \wedge (\$\mathsf{a1} = \mathtt{k}) \wedge (\$\mathsf{a2} = \$\mathsf{fp}) \wedge (\$\mathsf{a3} = \$\mathsf{sp}) \\
g_1 & \triangleq & (\$\mathsf{a0} = 0 \rightarrow \$\mathsf{v0}' = 1) \wedge (\$\mathsf{a0} \neq 0 \rightarrow \$\mathsf{v0}' = 0) \wedge \\
& & \quad \wedge \mathsf{Rid}(\{\$\mathsf{gp}, \$\mathsf{s0}, \ldots, \$\mathsf{s7}\}) \wedge \mathsf{Hnid}(\{\$\mathsf{sp} - 1, \$\mathsf{sp}\}) \wedge g_{\mathrm{frm}} \\
g_c & \triangleq & (\$\mathsf{a0} = 0 \rightarrow (\$\mathsf{ra}' = \$\mathsf{a1}) \wedge (\$\mathsf{fp}' = \$\mathsf{a2}) \wedge (\$\mathsf{sp}' = \$\mathsf{a3}) \\
& & \qquad\qquad \wedge \mathsf{Rid}(\{\$\mathsf{gp}, \$\mathsf{s0}, \ldots, \$\mathsf{s7}\})) \\
& & \wedge (\$\mathsf{a0} \neq 0 \rightarrow \mathsf{Rid}(\{\$\mathsf{gp}, \$\mathsf{sp}, \$\mathsf{fp}, \$\mathsf{ra}, \$\mathsf{s0}, \ldots, \$\mathsf{s7}\}) \\
p_4 & \triangleq & \mathsf{TRUE} \\
g_4 & \triangleq & g_c \wedge \mathsf{Hnid}(\{\$\mathsf{sp} - 1, \$\mathsf{sp}\}) \\
p_5 & \triangleq & \mathsf{TRUE} \\
g_5 & \triangleq & g_c \wedge \mathsf{Hnid}(\emptyset)
\end{array}
$$

Figure 4.13: Specifications of Code in Figure 4.14

instead of the normal return address of the caller.

$$\frac{\forall \mathbb{S}.\ p\ \mathbb{S} \rightarrow g\ \mathbb{S}\ \mathbb{S}}{\psi \vdash \{(p, g)\}\ \mathtt{f} :\ \mathsf{jr}\ \$\mathsf{ra}}\ (\textsc{jwc})$$

Shivers and Fisher [89] use the "super tail recursive" function call to implement their MRLC, which is essentially multi-return function call with stack cutting. The implementation of MRLC can be certified using SCAP-II.

In Figure 4.12, we show a C-- program using weak continuations. The behavior of the function `rev` is similar to the one shown in Figure 4.10. If the argument is 0, the function `cmp1` may skip over its caller `cmp0` and cut to the stack of `rev`.

Figure 4.14 shows the compiled code and specifications. Specifications of the code is shown in Figure 4.13. To simplify the presentation, we pass the weak continuation k (which contains the return code pointer, the frame pointer and the stack pointer) via registers \$a1-\$a3. The specification $(\mathtt{p}_0, \mathtt{g}_0)$ for rev is very similar to the one shown in Figure 4.11. The specification at the call site of cmp0 is $(\mathtt{p}_1, \mathtt{g}_1)$. Specifications for functions cmp0 and cmp1 are given as $(\mathtt{p}_4, \mathtt{g}_4)$ and $(\mathtt{p}_5, \mathtt{g}_5)$, respectively. Notice that two different conditions are considered in $\mathtt{g}_4$ and $\mathtt{g}_5$, *i.e.,* the condition under which that the functions return normally and the condition under which the functions cut the stack. Specifications for other code blocks are omitted.

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

The code in Figure 4.15 shows a simple implementation of setjmp/longjmp functions and their specifications. Here we borrow the separation logic [87] 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 [101], separation logic primitives can be encoded in Coq and embedded in general predicates.

The precondition $\mathtt{p}_0$ of setjmp simply requires that the argument \$a0 point to a jmp_buf. It guarantees ($\mathtt{g}_0$) that the return value is 0; values of callee save registers, return code

```
rev:    -{(p0, g0)}
        addiu  $sp, $sp, -2          ;allocate frame
        sw     $fp, 2($sp)           ;save old $fp
        addiu  $fp, $sp, 2           ;new $fp
        sw     $ra, -1($fp)          ;save $ra
        addiu  $a1, $zero, k         ;set cut cont
        addiu  $a2, $fp, 0           ;save $fp
        addiu  $a3, $sp, 0           ;save $sp
        addiu  $ra, $zero, ct        ;set ret cont
        -{(p1, g1)}
        j      cmp0                  ;call cmp0

ct:     -{(p2, g2)}
        addiu  $v0, $zero, 0         ;return 0
        j      epilog

k:      -{(p3, g3)}
        addiu  $v0, $zero, 1         ;return 1
        j      epilog

cmp0:   -{(p4, g4)}
        addiu  $sp, $sp, -2          ;allocate frame
        sw     $fp, 2($sp)           ;save old $fp
        addiu  $fp, $sp, 2           ;new $fp
        sw     $ra, -1($fp)          ;save $ra
        addiu  $ra, $zero, epilog
        j      cmp1                  ;call cmp1

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

cutto:  -{(p6, g6)}
        addiu  $ra, $a1, 0           ;set $ra to k
        addiu  $fp, $a2, 0           ;restore k's $fp
        addiu  $sp, $a3, 0           ;restore k's $sp
        jr     $ra                   ;goto k

epilog: -{(p7, g7)}
        lw     $ra, -1($fp)          ;restore $ra
        lw     $fp, 0($fp)           ;restore $fp
        addiu  $sp, $sp, 2           ;restore $sp
        jr     $ra                   ;return
```

Figure 4.14: Example for Weak Continuation

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

```
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 4.15: 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 4.16: C Program Using `setjmp/longjmp`

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 4.16 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. It has the same behavior with the function shown in Figure 4.12, except that here we make a non-local jump by using the `setjmp/longjmp` instead of a weak continuation.

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 4.17 and 4.18. Here we reuse some macros defined previously in Figures 4.15 and 4.8.

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. The guarantee g_0 is similar to the one shown in Figure 4.14. It specifies the relationship between the

```
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 4.17: TM Code Using `setjmp`/`longjmp`

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

Figure 4.18: Specifications for Code in Figure 4.17

argument `$a0` and the return value `$v0'`, and the preservation of callee-save registers and memory (except for the space for stack frames). Specifications for function `cmp0` and `cmp1` are $(\mathtt{p}_4, \mathtt{g}_4)$ and $(\mathtt{p}_5, \mathtt{g}_5)$, respectively. They are similar to the ones given in Figure 4.14 too. However, it is a little tricky to specify the code labeled by `ct1`, which may be executed twice: the first time after the return from `setjmp` and the second time after the return from `longjmp`. We need to consider both cases in the specification $(\mathtt{p}_1, \mathtt{g}_1)$.

4.6 Embedding of SCAP into OCAP

In this section, we show how to embed SCAP into OCAP. Given the soundness of OCAP, this embedding also shows the soundness of our SCAP inference rules.

$$\begin{array}{llcl} (\mathit{SCAPTy}) & \mathcal{L}_{\text{SCAP}} & \triangleq & \mathit{StatePred} * \mathit{Guarantee} \\ (\mathit{CdSpec}) & \theta & ::= & (\mathtt{p}, \mathtt{g}) \quad \in \ \mathcal{L}_{\text{SCAP}} \end{array}$$

As shown above, the code specification θ in OCAP is instantiated into the pair of $\mathtt{p}$ and $\mathtt{g}$, *i.e.*, SCAP specification of code blocks. The meta-type of SCAP specification in CiC is then $\mathcal{L}_{\text{SCAP}}$.

We also assign a language ID ρ to SCAP. Then we use the lifting function $\llcorner\psi\lrcorner_\rho$ to convert the ψ in SCAP to OCAP's specification Ψ.

$$\llcorner\psi\lrcorner_\rho \triangleq \{(\mathtt{f},\ \langle\rho, \mathcal{L}_{\text{SCAP}}, (\mathtt{p}, \mathtt{g})\rangle) \mid (\mathtt{f},\ (\mathtt{p}, \mathtt{g})) \in \psi\}$$

The following interpretation function takes the SCAP specification $(\mathtt{p}, \mathtt{g})$ and transforms it into the assertion in OCAP.

$$[\![\,(\mathtt{p}, \mathtt{g})\,]\!]^{(\rho,\mathcal{D})}_{\mathcal{L}_{\text{SCAP}}} \ \triangleq\ \lambda\Psi, \mathbb{S}.\ \mathtt{p}\ \mathbb{S}\ \wedge\ \exists n.\mathsf{WFST}(n, \mathtt{g}, \mathbb{S}, \mathcal{D}, \Psi)$$

Here $\mathcal{D}$ is an open parameter which describes the verification systems used to verify the external world around SCAP code. The interpretation simply specifies the SCAP program invariants we have just shown, except that we reformulate the previous definition of wfst to adapt to OCAP specification Ψ.

$$
\begin{aligned}
&\mathsf{WFST}(0, \mathsf{g}, \mathbb{S}, \mathcal{D}, \Psi) \triangleq \\
&\qquad \forall \mathbb{S}'.\ \mathsf{g}\ \mathbb{S}\ \mathbb{S}' \rightarrow \exists \pi.\ (\mathsf{codeptr}(\mathbb{S}'.\mathbb{R}(\$\mathsf{ra}), \pi) \wedge [\![\![\, \pi \,]\!]\!]_{\mathcal{D}})\ \Psi\ \mathbb{S}' \\
&\mathsf{WFST}(n{+}1, \mathsf{g}, \mathbb{S}, \mathcal{D}, \Psi) \triangleq \\
&\qquad \forall \mathbb{S}'.\ \mathsf{g}\ \mathbb{S}\ \mathbb{S}' \rightarrow \exists \mathsf{p}', \mathsf{g}'.\ (\mathbb{S}'.\mathbb{R}(\$\mathsf{ra}),\ \langle \rho, \mathcal{L}_{\mathrm{SCAP}}, (\mathsf{p}', \mathsf{g}') \rangle) \in \Psi \\
&\qquad\qquad\qquad\qquad \wedge\ \mathsf{p}'\ \mathbb{S}'\ \wedge\ \exists m \leq n.\ \mathsf{WFST}(m, \mathsf{g}', \mathbb{S}', \mathcal{D}, \Psi).
\end{aligned}
$$

The definition of WFST is similar to wfst, but we look up code specifications from OCAP's Ψ. Since we are now in an open world, we allow SCAP code to return to the external world even if the depth of the SCAP stack is 0, as long as $\$\mathsf{ra}$ is a valid code pointer and the interpretation of its specification π is satisfied at the return point. The open parameter $\mathcal{D}$ is used here to interpret the specification π.

It is also important to note that we do not need ρ and $\mathcal{D}$ to use SCAP, although they are open parameters in the interpretation. When we certify code using SCAP, we only use SCAP rules to derive the well-formedness of instruction sequences (*i.e.*, $\psi \vdash \{(\mathtt{p}, \mathtt{g})\}\ \mathtt{f} :\ \mathbb{I}$) and code heaps (*i.e.*, $\psi \vdash \mathbb{C} : \psi'$) with respect to *SCAP* specification ψ. The interpretation is *not* used until we want to link the certified SCAP code with code certified in other logics. We instantiate ρ and $\mathcal{D}$ in each specific application scenario. Theorem 4.1 shows the soundness of SCAP rules and their embedding in OCAP, which is independent with these open parameters.

Theorem 4.1 (Soundness of the Embedding of SCAP)

Suppose ρ is the language ID assigned to SCAP. For all $\mathcal{D}$ describing foreign code and $\rho \notin dom(\mathcal{D})$, let $\mathcal{D}' = \mathcal{D}\{\rho \leadsto \langle \mathcal{L}_{\mathrm{SCAP}}, [\![\, \_ \,]\!]^{(\rho,\mathcal{D})}_{\mathcal{L}_{\mathrm{SCAP}}} \rangle\}$.

(1) If $\psi \vdash \{(\mathtt{p}, \mathtt{g})\}\ \mathtt{f} : \mathbb{I}$ in SCAP, we have $\mathcal{D}' \vdash \{\langle \mathtt{a} \rangle_{\Psi}\}\ \mathtt{f} : \mathbb{I}$ in OCAP, where $\Psi = \llcorner \psi \lrcorner_{\rho}$ and $\mathtt{a} = [\![\, (\mathtt{p}, \mathtt{g}) \,]\!]^{(\rho,\mathcal{D})}_{\mathcal{L}_{\mathrm{SCAP}}}$.

(2) If $\psi \vdash \mathbb{C}\!:\!\psi'$ in SCAP, we have $\mathcal{D}'; \llcorner\psi\lrcorner_\rho \vdash \mathbb{C}\!:\!\llcorner\psi'\lrcorner_\rho$ in OCAP.

Proof. To prove (1), we need to consider all instruction sequence rules in SCAP and show that their premises imply the premises of the corresponding rules in OCAP. Here we show the proof for the CALL rule and the RET rule in the lemmas below. Proofs for other rules are trivial.

Given the proof of (1), (2) trivially follows the formulation of the SCAP-CDHP rule in SCAP and the CDHP rule in OCAP.

The proof has also been formalized in Coq [31]. □

Lemma 4.2 (Stack Strengthen) For all $\mathcal{D}$, n, g, g′, $\mathbb{S}$, $\mathbb{S}'$ and Ψ, if $\mathsf{WFST}(n, \mathrm{g}, \mathbb{S}, \mathcal{D}, \Psi)$ and $\forall \mathbb{S}''. \mathrm{g}'\ \mathbb{S}'\ \mathbb{S}'' \rightarrow \mathrm{g}\ \mathbb{S}\ \mathbb{S}''$, we have $\mathsf{WFST}(n, \mathrm{g}', \mathbb{S}', \mathcal{D}, \Psi)$.

Proof. This trivially follows the definition of WFST. □

Lemma 4.3 (Call) For all $\mathcal{D}$, ψ, ρ, $\mathtt{f}$, $\mathtt{f}'$, let $\mathcal{D}' = \mathcal{D}\{\rho \leadsto \langle \mathcal{L}_{\mathrm{SCAP}}, [\![\, \_ \,]\!]^{(\rho,\mathcal{D})}_{\mathcal{L}_{\mathrm{SCAP}}} \rangle\}$ and $\Psi = \llcorner\psi\lrcorner_\rho$. Suppose $\rho \notin dom(\mathcal{D})$, $(\mathtt{f}', (\mathrm{p}', \mathrm{g}')) \in \psi$ and $(\mathtt{f}{+}1, (\mathrm{p}'', \mathrm{g}'')) \in \psi$. If

1. $\forall \mathbb{H}, \mathbb{R}.\ \mathtt{p}\ (\mathbb{H}, \mathbb{R}) \rightarrow \mathtt{p}'\ (\mathbb{H}, \mathbb{R}\{\mathtt{\$ra} \leadsto \mathtt{f}{+}1\})$

2. $\forall \mathbb{H}, \mathbb{R}, \mathbb{S}'.\ \mathtt{p}\ (\mathbb{H}, \mathbb{R}) \rightarrow \mathrm{g}'\ (\mathbb{H}, \mathbb{R}\{\mathtt{\$ra} \leadsto \mathtt{f}{+}1\})\ \mathbb{S}' \rightarrow (\mathrm{p}''\ \mathbb{S}' \wedge (\forall \mathbb{S}''.\ \mathrm{g}''\ \mathbb{S}'\ \mathbb{S}'' \rightarrow$
   $\mathtt{g}\ (\mathbb{H}, \mathbb{R})\ \mathbb{S}''))$;

3. $\forall \mathbb{S}, \mathbb{S}'.\ \mathrm{g}'\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathbb{S}.\mathbb{R}(\mathtt{\$ra}) = \mathbb{S}'.\mathbb{R}(\mathtt{\$ra})$;

we have

$$\forall \Psi', \mathbb{H}, \mathbb{R}.\ \langle \mathtt{a} \rangle_\Psi\ \Psi'\ (\mathbb{H}, \mathbb{R}) \rightarrow \exists \pi'.\ (\mathsf{codeptr}(\mathtt{f}', \pi') \wedge [\![\![\, \pi' \,]\!]\!]_{\mathcal{D}'})\ \Psi'\ (\mathbb{H}, \mathbb{R}\{\mathtt{\$ra} \leadsto \mathtt{f}{+}1\}),$$

where $\mathtt{a} = [\![\, (\mathtt{p}, \mathtt{g}) \,]\!]^{(\rho,\mathcal{D})}_{\mathcal{L}_{\mathrm{SCAP}}}$. (In short, the CALL rule in SCAP can be derived from the JAL rule in OCAP.)

Proof. We let $\pi' = (\rho, \mathcal{L}_{\mathrm{SCAP}}, (\mathrm{p}', \mathrm{g}'))$. Assume $\langle \mathtt{a} \rangle_\Psi\ \Psi'\ (\mathbb{H}, \mathbb{R})$ holds, therefore we know $\Psi \subseteq \Psi'$. Since $\Psi = \llcorner\psi\lrcorner_\rho$ and $\Psi \subseteq \Psi'$, it is obvious that $\mathsf{codeptr}(\mathtt{f}', \pi')\ \Psi'\ \mathbb{S}$ holds for all $\mathbb{S}$.

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

4. $\Psi \subseteq \Psi'$;

5. $\mathtt{p}\ (\mathbb{H}, \mathbb{R})$;

6. $\mathsf{WFST}(n, \mathtt{g}, (\mathbb{H}, \mathbb{R}), \mathcal{D}, \Psi')$;

we need to prove

a. $\mathtt{p}'\ (\mathbb{H}, \mathbb{R}\{\$\mathsf{ra} \rightsquigarrow \mathtt{f}+1\})$; and

b. $\mathsf{WFST}(n+1, \mathtt{g}', (\mathbb{H}, \mathbb{R}\{\$\mathsf{ra} \rightsquigarrow \mathtt{f}+1\}), \mathcal{D}, \Psi')$;

The proof of a is trivial (by 1 and 5). We are now focused on the proof of b.
Suppose $\mathtt{g}'\ (\mathbb{H}, \mathbb{R}\{\$\mathsf{ra} \rightsquigarrow \mathtt{f}+1\})\ \mathbb{S}$ for some $\mathbb{S}$,

- by 3 we know $\mathbb{S}.\mathbb{R}(\$\mathsf{ra}) = \mathtt{f}+1$, therefore $(\mathbb{S}.\mathbb{R}(\$\mathsf{ra}), (\rho, \mathcal{L}_{\text{SCAP}}, (\mathtt{p}'', \mathtt{g}''))) \in \Psi'$;

- by 5 and 2 we know $\mathtt{p}''\mathbb{S}$;

- by 5 and 2 we know $\forall \mathbb{S}'.\ \mathtt{g}''\ \mathbb{S}'\mathbb{S} \rightarrow \mathtt{g}\ (\mathbb{H}, \mathbb{R})\ \mathbb{S}$. Therefore, by 6 and Lemma 4.2 we get $\mathsf{WFST}(n, \mathtt{g}'', \mathbb{S}, \mathcal{D}, \Psi')$.

Then, by the definition of WFST we get $\mathsf{WFST}(n+1, \mathtt{g}', (\mathbb{H}, \mathbb{R}\{\$\mathsf{ra} \rightsquigarrow \mathtt{f}+1\}), \mathcal{D}, \Psi')$. $\Box$

Lemma 4.4 (Return) For all $\mathcal{D}$, ψ, ρ, $\mathtt{f}$, $\mathtt{f}'$, let $\mathcal{D}' = \mathcal{D}\{\rho \rightsquigarrow \langle \mathcal{L}_{\text{SCAP}}, [\![\ \_\ ]\!]^{(\rho,\mathcal{D})}_{\mathcal{L}_{\text{SCAP}}} \rangle\}$ and $\Psi = \llcorner \psi \lrcorner_\rho$. Suppose $\rho \notin dom(\mathcal{D})$. If $\forall \mathbb{S}. \mathtt{p}\ \mathbb{S} \rightarrow \mathtt{g}\ \mathbb{S}\ \mathbb{S}$, then for all Ψ, $\mathbb{H}$ and $\mathbb{R}$, we have

$$\forall \Psi', \mathbb{S}.\ \langle \mathsf{a} \rangle_\Psi\ \Psi'\ \mathbb{S} \rightarrow\!\!\rightarrow \exists \pi'.\ (\mathsf{codeptr}(\mathbb{S}.\mathbb{R}(\$\mathsf{ra}), \pi') \wedge [\![\ \pi'\ ]\!]_{\mathcal{D}'})\ \Psi'\ \mathbb{S}\,,$$

where $\mathsf{a} = [\![\ (\mathtt{p}, \mathtt{g})\ ]\!]^{(\rho,\mathcal{D})}_{\mathcal{L}_{\text{SCAP}}}$. That is, the RET rule in SCAP can be derived from an instantiation of the JR rule in OCAP, where $\mathtt{r}_s$ is instantiated to $\mathsf{\$ra}$.

Proof. Suppose $\langle [\![\ (\mathtt{p}, \mathtt{g})\ ]\!]^{(\rho,\mathcal{D})}_{\mathcal{L}_{\text{SCAP}}} \rangle_\Psi\ \Psi'\ \mathbb{S}$ for some $\mathbb{S}$. We know that

1. $\mathtt{p}\ \mathbb{S}$;

2. $\mathsf{g}\ \mathbb{S}\ \mathbb{S}$ (by our assumption $\forall \mathbb{S}.\mathsf{p}\ \mathbb{S} \rightarrow \mathsf{g}\ \mathbb{S}\ \mathbb{S}$); and

3. $\mathsf{WFST}(n, \mathsf{g}, \mathbb{S}, \mathcal{D}, \Psi')$ for some n.

If $n = 0$, it is trivial to prove $\exists \pi'.\ (\mathsf{codeptr}(\mathbb{S}.\mathbb{R}(\$\mathsf{ra}), \pi') \wedge [\![\, \pi' \,]\!]_{\mathcal{D}})\ \ \Psi'\ \ \mathbb{S}$ by 2, 3 and the definition of WFST. We know $\mathcal{D} \subseteq \mathcal{D}'$ because $\rho \notin \mathit{dom}(\mathcal{D})$. Therefore we know $\exists \pi'.\ (\mathsf{codeptr}(\mathbb{S}.\mathbb{R}(\$\mathsf{ra}), \pi') \wedge [\![\, \pi' \,]\!]_{\mathcal{D}'})\ \Psi'\ \mathbb{S}$ by Lemma 3.5.

If $n > 0$, by 2, 3 and the definition of WFST we know there exists $(\mathsf{p}', \mathsf{g}')$ such that

4. $(\mathbb{S}.\mathbb{R}(\$\mathsf{ra})\ (\rho, \mathcal{L}_{\text{SCAP}}, (\mathsf{p}', \mathsf{g}'))) \in \mathit{dom}(\Psi)$;

5. $\mathsf{p}'\ (\mathbb{H}, \mathbb{R})$; and

6. $\mathsf{WFST}(m, \mathsf{g}', (\mathbb{H}, \mathbb{R}), \Psi)$ for some $m < n$.

Let $\pi' = (\rho, \mathcal{L}_{\text{SCAP}}, (\mathsf{p}', \mathsf{g}'))$. By the definition of the interpretation function, we know $(\mathsf{codeptr}(\mathbb{S}.\mathbb{R}(\$\mathsf{ra}), \pi') \wedge [\![\, \pi' \,]\!]_{\mathcal{D}'})\ \Psi'\ \mathbb{S}$. □

4.7 Discussions and Summary

Continuing over the related work discussed in Section 4.1.1, STAL [68] and its variations [26, 96] 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 [26] 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 [26], 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 4.7, 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 [11] 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 uses the indexed model [8] to prove the soundness of his system, while we prove the soundness of SCAP by enforcing the stack invariant. 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.* [12] 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 4.1), 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.* [2, 58] 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.

State Relations as Program Specifications. SCAP is not the first to use relations between two states as program specifications. The rely-guarantee method [59], TLA [63], and VDM [60] all use state relations to specify programs. However, the guarantee g used in SCAP is different from those used in previous systems. Generalizing the idea of local guarantee [102], SCAP uses 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 g matches precisely the VDM postcondition, but intermediate 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 g's always refers to the (same) state at the exit point. We use these intermediate 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 [103] on machine code verification can also support stack-based procedure call and return. His correctness theorem for each subroutine resembles our guarantee 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.

Summary In this chapter we propose SCAP 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, and multi-return function call. For each control abstraction, we have formalized its invariants and showed how to certify its implementation. SCAP is proposed as instances of the OCAP framework. The complete soundness proof and a full verification of several examples have been formalized in the Coq proof assistant [24].

Chapter 5

Concurrent Code with Dynamic Thread Creation

We have seen SCAP for stack-based reasoning of sequential control abstractions in Chapter 4. However, most real-world systems use some form of concurrency in their core software. To build fully certified system software, it is crucial to certify low-level concurrent programs.

Yu and Shao [102] proposed a certified formal framework (known as CCAP) for specifying and reasoning about general properties of non-preemptive concurrent programs at the assembly level. They applied the invariant-based proof technique for verifying general safety properties and the rely-guarantee methodology [59] for decomposition. Their thread model, however, is rather restrictive in that no threads can be created or terminated dynamically and no sharing of code is allowed between threads; both of these features are widely supported and used in mainstream programming languages such as C, Java, and Concurrent ML [86].

It is difficult to reason about concurrent code with dynamic thread creation and termination because it implies a changing thread *environment* (*i.e.,* the collection of all live threads in the system other than the thread under concern). Such dynamic environment cannot be tracked during static verification, yet we still must somehow reason about it.

For example, we must ensure that a newly created thread does not interfere with existing live threads, but at the same time we do not want to enforce non-interference for threads that have no overlap in their lifetime. Using one copy of code to create multiple threads also complicates program specification.

Existing work on verification of concurrent programs almost exclusively uses high-level calculi (*e.g.*, CSP [52], CCS [66], TLA [63] and Concurrent Separation Logic [78, 14]). Also, existing work on the rely-guarantee methodology for shared-memory concurrency only supports properly nested concurrent code in the form of cobegin $P_1 \| \ldots \| P_n$ coend (which is a language construct for parallel composition where code blocks $P_1, \ldots, P_n$ execute in parallel and all terminate at the coend point). They do not support dynamic thread creation and termination.

Modularity is also important to make verification scale. Existing work on the rely-guarantee methodology supports thread modularity, *i.e.*, different threads can be verified separately without looking into other threads' code. However, they do not support code reuse very well. In CCAP, if a procedure is called in more than one thread, it must be verified multiple times using different specifications, one for each calling thread. We want a procedure to be specified and verified once so it can be reused for different threads.

In this chapter, we present a new program logic for $\underline{c}$ertified $\underline{m}$ulti-threaded $\underline{a}$ssembly $\underline{p}$rogramming (CMAP). It is derived from the research I have done jointly with Zhong Shao [34]. Instead of using the machine formalized in Section 2.2, we present CMAP based on an abstract machine with built-in support of threads. The machine supports dynamic thread creation with argument passing (the "fork" operation) as well as termination (the "exit" operation). Thread "join" can also be implemented in the machine language using synchronizations. The "fork/join" thread model is more general than "cobegin/coend" in that it supports unbounded dynamic thread creation, which poses new challenges for verification. CMAP is the first program logic that applies the rely-guarantee method to certify assembly code with unbounded dynamic thread creation. In particular, it unifies the concepts of a thread's assumption/guarantee and its environment's guarantee/assumption,

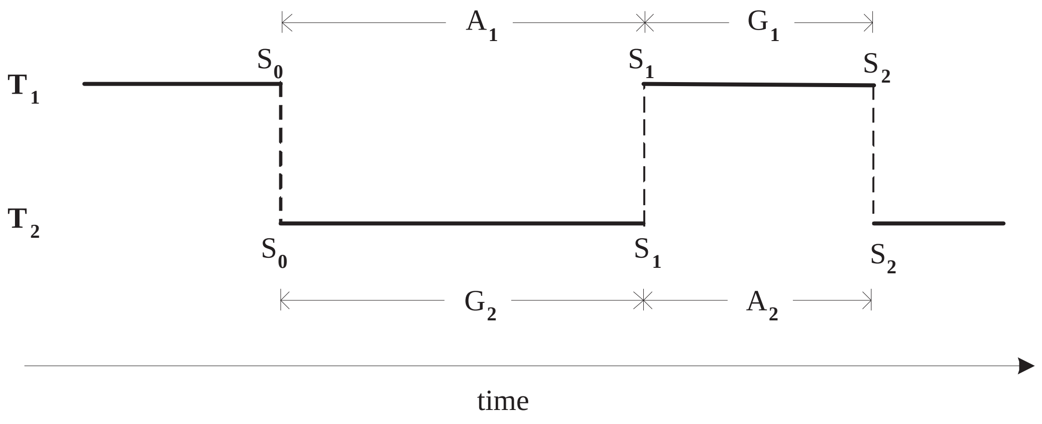


Figure 5.1: Rely-guarantee-based reasoning

and allows a thread to change its assumption/guarantee to track the composition of its dynamically changing environment.

Based on the rely-guarantee method, CMAP supports thread modular reasoning. Moreover, it merges code heaps of different threads into one uniform code heap, and allow code segments to be specified independently of the threads in which they are used. Code segments can be specified and certified once but used in multiple threads. Therefore, CMAP has good support of code (and proof) sharing between threads.

Some practical issues are also solved in CMAP, including multiple "incarnation" of thread code, thread argument passing, saving and resuming of thread local data at the time of context switching, *etc.*. These issues are important for realistic multi-threaded programming, but have never been discussed in existing work.

5.1 Background and Challenges

In this section, we give an overview of the rely-guarantee method, and explain the challenges to support dynamic thread creations.

5.1.1 Rely-Guarantee-Based Reasoning

The rely-guarantee (R-G) proof method [59] is one of the best-studied approaches to the compositional verification of shared-memory concurrent programs. Under the R-G paradigm, every thread is associated with a pair (A, G), with the meaning that if the environment (*i.e.*, the collection of all of the rest threads) satisfies the assumption A, the thread will meet its guarantee G to the environment. In the shared-memory model, the assumption A of a thread describes what atomic transitions may be performed by other threads, while the guarantee G of a thread must hold on every atomic transition of the thread. They are typically modeled as predicates on a pair of states, which are often called *actions*.

For instance, in Figure 5.1 we have two interleaving threads T_1 and T_2. T_1's assumption A_1 adds constraints on the transition (S_0, S_1) made by the environment (T_2 in this case), while G_1 describes the transition (S_1, S_2), assuming the environment's transition satisfies A_1. Similarly A_2 describes (S_1, S_2) and G_2 describes (S_0, S_1).

We need two steps to reason about a concurrent program consisting of T_1 and T_2. First, we check that there is no interference between threads, i.e., that each thread's assumption can be satisfied by its environment. In our example, non-interference is satisfied as long as $G_1 \Rightarrow A_2$ (a shorthand for $\forall S, S'.G_1(S, S') \Rightarrow A_2(S, S')$), and $G_2 \Rightarrow A_1$. Second, we check that T_1 and T_2 do not lie, that is, they satisfy their guarantee as long as their assumption is satisfied. As we can see, the first step only uses the specification of each thread, while the second step can be carried out independently without looking at other threads' code. This is how the R-G paradigm achieves thread-modularity.

5.1.2 R-G in Non-Preemptive Thread Model

CMAP adopts a non-preemptive thread model, in which threads yield control voluntarily with a yield instruction, as shown in Figure 5.2. The preemptive model can be regarded as a special case of the non-preemptive one, in which an explicit yield is used at every program point. Also, on real machines, programs might run in both preemptive and non-

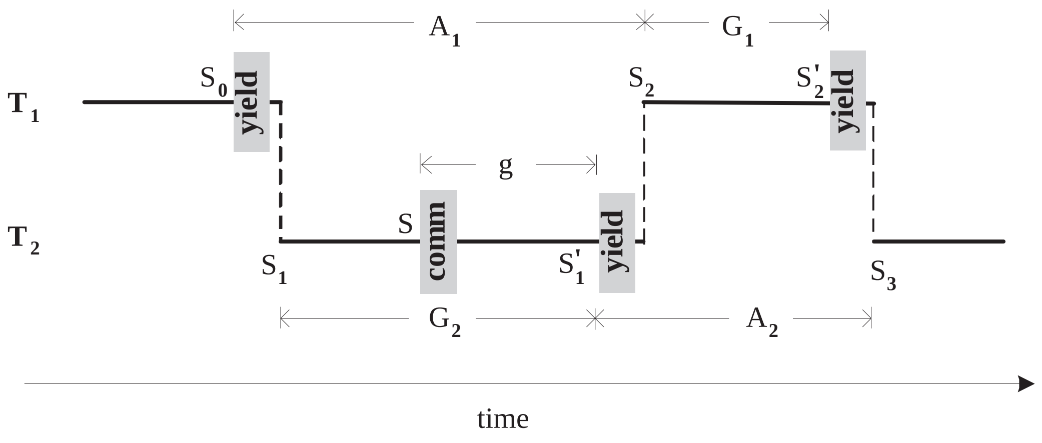


Figure 5.2: R-G in a non-preemptive setting

preemptive settings: preemption is usually implemented using interrupts; a program can disable the interrupt to get into non-preemptive setting.

An "atomic" transition in a non-preemptive setting then corresponds to a sequence of instructions between two yields. For instance, in Figure 5.2 the state pair (S_2, S_2') corresponds to an atomic transition of thread T_1. A difficulty in modeling concurrency in such a setting is that the effect of an "atomic" transition cannot be completely captured until the end. For example, in Figure 5.2, the transition (S_1, S_1') should satisfy G_2. But when we reach the intermediate state S, we have no idea of what the whole transition (*i.e.*, (S_1, S_1')) will be. At this point, neither (S_1, S) nor (S, S_1') need satisfy G_2. Instead, it may rely on the remaining commands (the commands between comm and yield, including comm) to complete an adequate state transition. In CCAP [102], a "local guarantee" g is introduced for every program point to capture further state changes that must be made by the following commands before it is safe for the current thread to yield control. For instance, the local guarantee g attached to comm in Figure 5.2 describes the transition (S, S_1').

5.1.3 Challenges for Dynamic Thread Creation

To prove safety properties of multi-threaded programs, the key problem is to enforce the invariant that *all executing threads must not interfere with each other*. As mentioned in Sec-

Variables:
$\texttt{nat}[100]\ \texttt{data};$

Initially:
$\texttt{data}[i] = n_i, 0 \le i < 100$

```
main1 :
  data[0] := f(0);
  fork(chld, 0);
    ⋮
  data[99] := f(99);
  fork(chld, 99);
  ...
```

```
main2 :
  nat i := 0;
  while(i < 100){
    data[i] := f(i);
    fork(chld, i);
    i := i + 1;
  }
  ...
```

```
void chld(int x){
  data[x] := g(x, data[x]);
}
```

Figure 5.3: Loop: high-level program

tion 5.1.1, threads *do not interfere* (or they satisfy the *non-interference* property) only if each thread's assumption is implied by the guarantee of all other threads. We will formalize the Non-Interference property in Section 5.3.3. For languages that do not support dynamic thread creation, the code for each thread corresponds to exactly one executing thread. Using the rely-guarantee method, we can assign an assumption and guarantee to each thread code and enforce non-interference by checking all of these assumptions and guarantees, as is done in [102] and [38]. However, the following example shows that this simple approach cannot support dynamic thread creation and multiple "incarnation" of the thread code.

In Figure 5.3, the high-level pseudo code (using C-like syntax) shows the code for (two versions of) a main thread and child threads. The main thread initializes 100 pieces of data using some function `f`, and distributes them to 100 child threads that will work on their own data (by applying a function `g`) in parallel. The `fork` function creates a child thread that will execute the function pointed to by the first argument. The second argument of `fork` is passed to the function as argument. The thread `main1` does this in sequential code

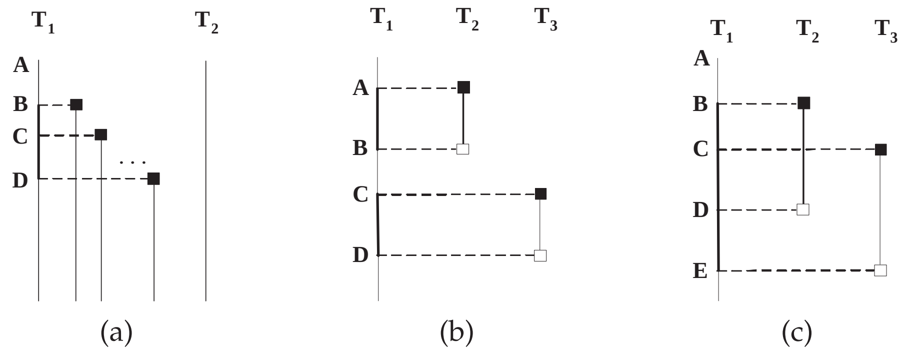


Figure 5.4: Interleaving of threads

while `main2` uses code with a loop. We assume that the high level code runs in preemptive mode. In other words, there is an implicit `yield` at any program point.

It is easy to see that both versions are "well-behaved" as long as the function `g` has no side effects, and all other threads in the rest of the system do not update the array of `data`. However, the simple approach used in [102] and [38] even cannot provide a specification for such trivial code.

1. Figure 5.4 (a) illustrates the execution of `main1` (time goes downwards). When doing data initialization (at stage A-B, meaning from point A to point B), the main thread needs to assume that no other threads in the environment (say, T_2) can change the array of `data`. However, the composition of the main thread's environment changes after a child thread is created. The assumption used at A-B is no longer appropriate for this new environment since the first child thread will write to `data`[0]. And the environment will keep changing with the execution of the main thread. How can we specify the `main1` thread to support such a dynamic thread environment?

   One possible approach is that the main thread relaxes its assumption to make exceptions for its child threads. However, it is hard to specify the parent-child relationship. Another approach is to use something like the program counter in the assumption and guarantee to indicate the phase of computation. This means the specification of the main thread is sensitive to the implementation. Also the program structure of the main thread has to be exposed to the specification of the child threads, which compromises modularity. The worst thing is that this approach simply won't work for the version `main2`.

2. Since multiple child threads are created, we must make sure that there is no interference between these children. It is easy for the above example since we can let the assumption and guarantee of the `chld` code be parameterized by its argument, and require $G_i \Rightarrow A_j$ given $i \neq j$. However, this approach cannot be generalized for threads that have dummy arguments and their behavior does not depend on their arguments at all. In this case $G_i \equiv G_j$ and $A_i \equiv A_j$ for any i and j. Then requiring $G_i \Rightarrow A_j$ is equivalent to requiring $G_i \Rightarrow A_i$, which cannot be true in general, given the meaning of assumptions and guarantees described in section 5.1.1. Do we need to distinguish these two kinds of threads and treat them differently? And how do we distinguish them?

3. Another issue introduced by dynamic thread creation, but not shown in this example program, is that the lifetimes of some threads may not overlap. In the case shown in Figure 5.4 (b), the lifetimes of T_2 and T_3 do not overlap and we should not statically enforce non-interference between them. Again, how can we specify and check the interleaving of threads, which can be as complex as shown in Figure 5.4 (c)?

In the next section we will show how these issues are resolved in the development of CMAP.

5.2 The CMAP Approach

In the rest of this chapter, to distinguish the executing thread and the thread code, we call the dynamically running thread the "dynamic thread" and the thread code the "static thread". In Figure 5.3 the function `chld` is the static child thread, from which 100 dynamic child threads are activated.

As explained in Section 5.1.3, the approach that requiring non-interference of static threads is too rigid to support dynamic thread creation. Our approach, instead, enforces the thread non-interference in a “lazy” way. We maintain a dynamic thread queue which contains all of the active threads in the system. When a new thread is created, it is added to the dynamic thread queue. A thread is removed from the queue when its execution terminates. We also require that, when specifying the program, each static thread be assigned an assumption/guarantee pair. However, we do not check for non-interference between static thread specifications. Instead, each dynamic thread is also assigned an assumption and guarantee at the time of creation, which is an instantiation of the corresponding static thread specification with the thread argument. We require that dynamic threads do not interfere with each other, which can be checked by inspecting their specifications.

This approach is very flexible in that each dynamic thread does not have to stick to one specification during its lifetime. When its environment changes, its specification can be changed accordingly. As long as the new specification does not introduce interference with other existing dynamic threads, and the subsequent behavior of this thread satisfies the new specification, the whole system is still interference-free. In this way, we can deal with the changing environment resulting from dynamic thread creation and termination. Problem 1 in Section 5.1.3 can be solved now.

If the lifetimes of two threads do not overlap, they will not show up in the system at the same time. Therefore we do not need to check for interference at all. Also, since each dynamic thread has its own specification, we no longer care about the specification of the corresponding static thread. Therefore problems 2 and 3 shown in Section 5.1.3 are no longer an issue in our approach.

5.2.1 Typing The Dynamic Thread Queue

We define the dynamic thread queue $\mathbb{Q}$ as a set of thread identifiers $\mathtt{t}_i$, and the assignment Θ of assumption/guarantee pairs to dynamic threads as a partial mapping from $\mathtt{t}_i$ to (A_i, G_i) [1]. The queue $\mathbb{Q}$ is "well-typed" with regard to Θ if:

- $\mathbb{Q} = dom(\Theta)$, where $dom(\Theta)$ is the domain of Θ;
- threads in $\mathbb{Q}$ do not interfere, *i.e.*, $\forall \mathtt{t}_i, \mathtt{t}_j . \mathtt{t}_i \neq \mathtt{t}_j \rightarrow (G_i \Rightarrow A_j)$; and
- each dynamic thread $\mathtt{t}_i$ is "well-behaved" with regard to (A_i, G_i), *i.e.*, if A_i is satisfied by the environment, $\mathtt{t}_i$'s execution does not get stuck and satisfies G_i.

Therefore, the invariant we need to maintain is that during the execution of the program, for the queue $\mathbb{Q}$ at each step there exists a Θ such that $\mathbb{Q}$ is well-typed with regard to Θ. In fact, we do not require Θ to be part of the program specification. We only need to ensure that there *exists* such a Θ at each step, which may be changing.

The content of the thread queue keeps changing, so how can we track the set of threads in the queue by a static inspection of the program? Here we follow the approach used for type-checking the dynamic data heap [69], which is dynamically updated by the store instruction and extended by the alloc instruction. We can ensure our invariant holds as long as the following conditions are satisfied:

- At the initial state (when the program starts to run) we can find a Θ to type-check the initial $\mathbb{Q}$. Usually the initial $\mathbb{Q}$ only contains the main thread, which will start to execute its first instruction, so we can simply assign the assumption/guarantee in the specification of the static main thread to the dynamic main thread.
- For each instruction in the program, assume that before the execution of the instruction there is a Θ such that $\mathbb{Q}$ is well typed. Then as long as certain constraints are satisfied to execute the instruction, there must exist a Θ' that can type check the

[1] This is a temporary formulation to illustrate the basic idea. We will use different definitions in the formal development of CMAP in Section 5.3.

resulting $\mathbb{Q}'$. For most instructions which do not change the content of the thread queue, this condition can be trivially satisfied. We are only interested in the "fork" and "exit" operation which will change the content of $\mathbb{Q}$.

For the "exit" instruction, the second condition can also be satisfied by the following lemma which can be trivially proven.

Lemma 5.1 (Thread Deletion)
If $\mathbb{Q}$ is well-typed with regard to Θ, then, for all $\mathtt{t} \in \mathbb{Q}$, we know $\mathbb{Q} \setminus \{\mathtt{t}\}$ is well-typed with regard to $\Theta \setminus \{\mathtt{t}\}$.

For the "fork" instruction, things are trickier. We need to ensure that the new child thread does not interfere with threads in the parent thread's environment. We also require that the parent thread does not interfere with the child thread. The following lemma ensures the first requirement.

Lemma 5.2 (Queue Extension I)
Suppose $\mathbb{Q}$ is well-typed with regard to Θ and the current executing thread is $\mathtt{t}$. If

- $\Theta(\mathtt{t}) = (A, G)$;
- a new thread $\mathtt{t}'$ is created by $\mathtt{t}$;
- (A', G') is the instantiation of the corresponding static thread specification by the thread argument;
- $A \Rightarrow A'$ and $G' \Rightarrow G$;

then $\mathbb{Q}' \cup \{\mathtt{t}'\}$ is well-typed with regard to $\Theta'\{\mathtt{t}' \leadsto (A', G')\}$, where $\mathbb{Q}' = \mathbb{Q} \setminus \{\mathtt{t}\}$ and $\Theta' = \Theta \setminus \{\mathtt{t}\}$.

Here $\mathbb{Q}'$ is the environment of the current thread $\mathtt{t}$. Since $\mathtt{t}$ does not interfere with its environment (because $\mathbb{Q}$ is well-typed), we know that its assumption A is an approximation of what the environment can guarantee (G_e), and similarly that G is an approximation of the environment's assumption (A_e). By this interpretation, we can unify the concepts of the current running thread's assumption/guarantee with its environment's guarantee/assumption. To ensure the new thread $\mathtt{t}'$ does not interfere with $\mathtt{t}$'s environment, we need $G' \Rightarrow A_e$ and $G_e \Rightarrow A'$, which can be derived from $G' \Rightarrow G$ and $A \Rightarrow A'$.

Still, we need to ensure that thread $\mathtt{t}$ does not interfere with $\mathtt{t}'$. As mentioned above, A and G are approximations of G_e and A_e, respectively. Since the environment is extended with the child thread, the guarantee G_e' for the new environment is $G_e \vee G'$ and the assumption for the new environment A_e' is $A_e \wedge A'$. We want to change A and G correspondingly to reflect the environment change. First, the following lemma says that the specification of a dynamic thread can be changed during its lifetime.

Lemma 5.3 (Queue Update)

Suppose $\mathbb{Q}$ is well-typed with regard to Θ and that the current executing thread is $\mathtt{t}$. If

- $\Theta(\mathtt{t}) = (A, G)$;
- $G'' \Rightarrow G$ and $A \Rightarrow A''$;
- the subsequent behavior of the current thread satisfies (A'', G'');

then $\mathbb{Q}$ is well-typed with regard to $\Theta\{\mathtt{t} \leadsto (A'', G'')\}$.

Now we can change the specification of the parent thread $\mathtt{t}$ to let it reflect the change of the environment.

Lemma 5.4 (Queue Extension II)

Suppose $\mathbb{Q}$ is well-typed with regard to Θ and the current executing thread is $\mathtt{t}$. If

- $\Theta(\mathtt{t}) = (A, G)$;
- a new thread $\mathtt{t}'$ is created by $\mathtt{t}$;
- (A', G') is the instantiation of the corresponding static thread specification by the thread argument;
- $(A \Rightarrow A') \wedge (G' \Rightarrow G)$;

- the remainder behavior of the thread t also satisfies $(A \vee G', G \wedge A')$;

then $\mathbb{Q} \cup \{\mathtt{t}'\}$ is well-typed with regard to $\Theta\{\mathtt{t}' \leadsto (A', G'), \mathtt{t} \leadsto (A \vee G', G \wedge A')\}$.

If t later creates another thread t″, because the specification of t already reflects the existence of t′, by Lemma 5.2 we know that t″ will not interfere with t′ as long as its specification satisfies the constraints. Therefore we do not need to explicitly check that t′ and t″ are activated from the same static thread or that multiple activations of a static thread do not interfere with each other.

These lemmas are used to prove the soundness of CMAP. They are somewhat similar to the heap update and heap extension lemmas used in TAL's soundness proof [69]. People familiar with the traditional rely-guarantee method may feel this is nothing but the parallel composition rule used to support nested cobegin/coend. However, by combining the invariant-based proof technique used by type systems and the traditional rely-guarantee method, we can now verify multi-threaded assembly program with a more flexible program structure than the cobegin/coend structure. In particular, programs which are not properly nested, as shown in Figure 5.4(c) and the `main2` program in Figure 5.3, can be supported in our system. This is one of the most important contributions of this paper.

5.2.2 Parameterized Assumption/Guarantee

The assumptions and guarantees are interfaces between threads, which should only talk about shared resources. As we allow multiple activations of static threads, the behavior of a dynamic thread may depend on its arguments, which is the thread's private data. Therefore, to specify a static thread, the assumption and guarantee need to be parameterized over the thread argument.

In our thread model, the flat memory space is shared by all threads, and as in most operating systems, the register file is saved at the moment of context switch. Therefore the register file is thread-private data. The thread argument is stored in a dedicated register.

Rather than letting the assumption and guarantee be parameterized over the thread

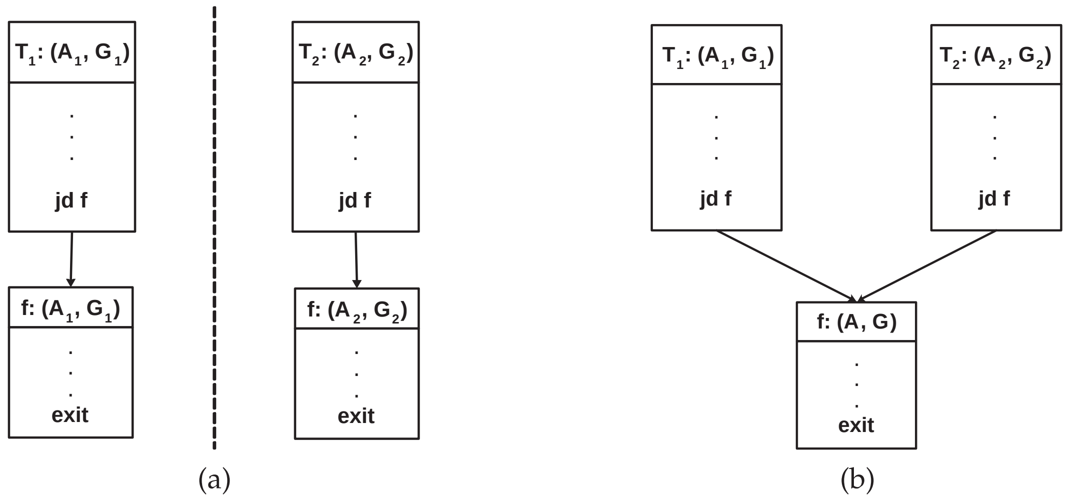


Figure 5.5: Code sharing between different threads

argument, we let them be parameterized over the whole register file. This makes our specification language very expressive. For instance, we allow the dynamic thread to change its specification during its lifetime to reflect change in the environment. If the thread has private data that tracks the composition of the environment, and its specification is parameterized by such data, then its specification automatically changes with the change of the data, which in turn results from the change of the thread environment. This is the key technique we use to support unbounded dynamic thread creation, as shown in the program `main2` in Figure 5.3.

5.2.3 Support of Modular Verification

The rely-guarantee method supports thread modularity well, *i.e.,* code of one thread can be verified independently without inspecting other threads' code. However, it does not have good support for code reuse. In CCAP, each thread has its own code heap and there is no sharing of code between threads. As Figure 5.5(a) shows, if an instruction sequence is used by multiple threads, it has multiple copies in different threads' code heaps, each copy verified with regard to the specifications of these threads.

Based on the rely-guarantee method, the thread modularity is also supported in our system. In addition, using our "lazy" checking of thread non-interference, and by the queue update lemma, we can allow instruction sequences to be specified independently of their calling thread, thus achieving better modularity.

As shown in Figure 5.5(b), we assign an assumption/guarantee pair to the specification of each instruction sequence, and require the instruction sequence be well-behaved with regard to its own assumption/guarantee. Similar to threads, the instruction sequence is well-behaved if, when it is executed by a dynamic thread, its execution is safe and satisfies its guarantee, as long as other dynamic threads in the environment satisfy the assumption. The instruction sequence only needs to be verified once with respect to its own specification, and can be executed by different threads as long as certain constraints are satisfied.

Intuitively, it is safe for a dynamic thread t with specification (A_i, G_i) to execute the instruction sequence labeled by f as long as executing it does not require a stronger assumption than A_i, nor does it violate the guarantee G_i. Therefore, if the specification of f is (A, G), t can call f as long as $A_i \Rightarrow A$ and $G \Rightarrow G_i$. The intuition is backed up by our queue update lemma.

5.3 The CMAP Logic

In this section, we present the CMAP logic based on an abstract machine, which extends the machine shown in Section 2.2 with a thread queue $\mathbb{Q}$. In the next Chapter, we will discuss the embedding of the CMAP logic into the OCAP framework for our original machine model.

5.3.1 The Abstract Machine

As shown in Figure 5.6, the machine has 16 general purpose registers and two special registers. The rt register records the thread id (t) of the currently executing thread. The ra register is used to pass argument when a new thread is created.

$$
\begin{array}{rrl}
(\mathit{Program}) & \mathbb{P} & ::= (\mathbb{C}, \mathbb{S}, \mathbb{Q}, \mathsf{pc}) \\
(\mathit{CodeHeap}) & \mathbb{C} & ::= \{\mathtt{f}_0 \leadsto \iota_0, \ldots, \mathtt{f}_n \leadsto \iota_n\} \\
(\mathit{State}) & \mathbb{S} & ::= (\mathbb{H}, \mathbb{R}) \\
(\mathit{Memory}) & \mathbb{H} & ::= \{\mathtt{l}_0 \leadsto \mathtt{w}_0, \ldots, \mathtt{l}_n \leadsto \mathtt{w}_n\} \\
(\mathit{RegFile}) & \mathbb{R} & ::= \{\mathtt{r}_0 \leadsto \mathtt{w}_0, \ldots, \mathtt{r}_{15} \leadsto \mathtt{w}_{15}, \mathsf{rt} \leadsto \mathtt{w}, \mathsf{ra} \leadsto \mathtt{w}\} \\
(\mathit{Register}) & \mathtt{r} & ::= \mathtt{r}_0 \mid \ldots \mid \mathtt{r}_{15} \mid \mathsf{rt} \mid \mathsf{ra} \\
(\mathit{Labels}) & \mathtt{f}, \mathtt{l} & ::= n \;\; (\mathit{nat\ nums}) \\
(\mathit{Word}) & \mathtt{w} & ::= i \;\; (\mathit{integers}) \\
(\mathit{TQueue}) & \mathbb{Q} & ::= \{\mathtt{t}_0 \leadsto (\mathbb{R}_0, \mathsf{pc}_0), \ldots, \mathtt{t}_n \leadsto (\mathbb{R}_n, \mathsf{pc}_n)\} \\
(\mathit{ThrdID}) & \mathtt{t} & ::= n \;\; (\mathit{nat\ nums}) \\
(\mathit{Instr}) & \iota & ::= \mathsf{yield} \mid \mathsf{fork}\ \mathtt{f}, \mathtt{r}_s \mid \mathsf{addu}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \mid \mathsf{addiu}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{w} \mid \mathsf{beq}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f} \\
 & & \quad \mid \mathsf{bgtz}\ \mathtt{r}_s, \mathtt{f} \mid \mathsf{lw}\ \mathtt{r}_d, \mathtt{w}(\mathtt{r}_s) \mid \mathsf{subu}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \mid \mathsf{sw}\ \mathtt{r}_t, \mathtt{w}(\mathtt{r}_s) \mid \mathsf{j}\ \mathtt{f} \mid \mathsf{exit} \\
(\mathit{InstrSeq}) & \mathbb{I} & ::= \iota \mid \iota; \mathbb{I}
\end{array}
$$

Figure 5.6: The abstract machine

We model the queue $\mathbb{Q}$ of ready threads as a partial mapping from the thread id to an execution context of a thread. The thread id $\mathtt{t}$ is a natural number generated randomly at run time when the thread is created. The thread execution context includes the snapshot of the register file and the program point where the thread will resume its execution. Note that $\mathbb{Q}$ does not contain the current executing thread, which is different from the dynamic thread queue used in Section 5.2.1.

In addition to the normal instructions we have seen in the original machine, we add primitives fork, exit and yield to support multi-threaded programming, which can be viewed as system calls to a thread library. We do not have a join instruction because thread join can be implemented in this machine using synchronization. Operational semantics of this abstract machine is shown in Figure 5.7. Here the functions $\mathsf{NextS}_{(\mathsf{pc},\iota)}(\_)$ and $\mathsf{NextPC}_{(\iota,\mathbb{S})}(\_)$ give the next state and program counter after executing the instruction ι at pc and state $\mathbb{S}$. The definitions are the same with those defined in Chapter 2.2 and are omitted here.

The primitive fork $\mathtt{f}, \mathtt{r}_s$ creates a new thread that will starts executing from $\mathtt{f}$, and passes the value in $\mathtt{r}_s$ to it as the argument. The new thread will be assigned a fresh thread id and placed in the thread queue waiting for execution. The current thread continues with the subsequent instructions.

$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{Q}, \mathsf{pc}) \longmapsto \mathbb{P}'$	
if $\mathbb{C}(\mathsf{pc}) =$	then $\mathbb{P}' =$
fork $\mathtt{f}, \mathtt{r}_s$	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{Q}\{\mathtt{t} \rightsquigarrow (\mathbb{R}', \mathtt{f})\}, \mathsf{pc}{+}1)$ if $\mathtt{t} \notin dom(\mathbb{Q}), \mathtt{t} \neq \mathbb{R}(\mathsf{rt})$, and $\mathbb{R}' = \{\mathtt{r}_0 \rightsquigarrow \_, \ldots, \mathtt{r}_{15} \rightsquigarrow \_, \mathsf{rt} \rightsquigarrow \mathtt{t}, \mathsf{ra} \rightsquigarrow \mathbb{R}(\mathtt{r}_s)\}$
yield	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}'), \mathbb{Q}' \setminus \{\mathtt{t}\}, \mathsf{pc}')$ where $\mathbb{Q}' = \mathbb{Q}\{\mathbb{R}(\mathsf{rt}) \rightsquigarrow (\mathbb{R}, \mathsf{pc}{+}1)\}, \mathtt{t} \in dom(\mathbb{Q}')$ and $\mathbb{Q}'(\mathtt{t}) = (\mathbb{R}', \mathsf{pc}')$
exit	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}'), \mathbb{Q} \setminus \{\mathtt{t}\}, \mathsf{pc}')$ where $\mathtt{t} \in dom(\mathbb{Q})$ and $\mathbb{Q}(\mathtt{t}) = (\mathbb{R}', \mathsf{pc}')$
exit	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{Q}, \mathsf{pc})$ if $\mathbb{Q} = \emptyset$
other ι	$(\mathbb{C}, \mathbb{S}', \mathbb{Q}, \mathsf{pc}')$ where $\mathbb{S}' = \mathsf{NextS}_{(\mathsf{pc},\iota)}(\mathbb{S})$ and $\mathsf{pc}' = \mathsf{NextPC}_{(\iota,\mathbb{S})}(\mathsf{pc})$

Figure 5.7: Operational Semantics for the CMAP Machine

At a yield instruction, the current thread will give up the control of the machine. Its execution context is stored in $\mathbb{Q}$. The scheduler will pick one thread non-deterministically from the thread queue (which might be the yielding thread itself), restore its execution context, and execute it.

The exit instruction terminates the execution of the current thread and non-deterministically selects a thread from the thread queue. If the thread queue is empty, the currently running thread is the last thread in the system. The machine reaches a stuttering state, *i.e.,* the next program configuration is always the same with the current one.

5.3.2 Program Specifications

The specification constructs of CMAP are defined in Figure 5.8. As in SCAP, the specification ψ for the code heap contains pairs of code labels and the specifications θ for the corresponding code blocks. The code block specification θ is a quadruple $(\mathrm{p}, \hat{\mathrm{g}}, \mathbb{A}, \mathbb{G})$. The assertion p is the precondition required to execute the code block. The local guarantee $\hat{\mathrm{g}}$, as explained in Section 5.1.2, describes a valid state transition – it is safe for the current thread to yield control only after making a state transition specified by $\hat{\mathrm{g}}$. We also assign a pair of $\mathbb{A}$ and $\mathbb{G}$ to each code block as part of its specification. As explained in Section 5.2.3,

(*CdSpec*)	θ	$::=$	$(\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}, \mathbb{G})$
(*CdHpSpec*)	ψ	$::=$	$\{(\mathtt{f}_0,\, \theta_0), \ldots, (\mathtt{f}_n,\, \theta_n)\}$
(*StatePred*)	$\mathtt{p}$	$\in$	$\mathit{State} \rightarrow \mathsf{Prop}$
(*Assumption*)	$\mathbb{A}$	$\in$	$\mathit{RegFile} \rightarrow \mathit{Memory} \rightarrow \mathit{Memory} \rightarrow \mathsf{Prop}$
(*Guarantee*)	$\mathbb{G}, \hat{\mathtt{g}}$	$\in$	$\mathit{RegFile} \rightarrow \mathit{Memory} \rightarrow \mathit{Memory} \rightarrow \mathsf{Prop}$
(*QSpec*)	Θ	$::=$	$\{\mathtt{t}_0 \leadsto (\mathtt{p}_0, \mathbb{A}_0, \mathbb{G}_0), \ldots, \mathtt{t}_k \leadsto (\mathtt{p}_k, \mathbb{A}_k, \mathbb{G}_k)\}$

Figure 5.8: Specification Constructs of CMAP

the code block can be certified with respect to its own $\mathbb{A}$ and $\mathbb{G}$ without knowing which thread executes it. Here the $\mathbb{A}$ and $\mathbb{G}$ reflect knowledge of the dynamic thread environment at the time the instruction sequence is executed. Assumptions and guarantees (*e.g.,* $\mathbb{A}$, $\mathbb{G}$ and $\hat{\mathtt{g}}$) are CiC terms with type *RegFile* $\rightarrow$ *Memory* $\rightarrow$ *Memory* $\rightarrow$ Prop, *i.e.,* predicates over a register file and two data heaps. Assumptions and guarantees specify the behavior of threads by describing the change of shared memory. As mentioned in Section 5.2.2, they are parameterized over the register file, which contains the private data of threads.

We also define the specification Θ of active threads in the thread queue $\mathbb{Q}$. Within Θ, each triple $(\mathtt{p}, \mathbb{A}, \mathbb{G})$ specifies a dynamic thread at the point when it yields the control or it is just created. The assertion $\mathtt{p}$ gives the constraint of the state when the thread gets control back to execute its remaining instructions. The assumption and guarantee used by the thread at the yield point are given by $\mathbb{A}$ and $\mathbb{G}$. As we said in Section 5.2.1, the $\mathbb{A}$ and $\mathbb{G}$ of each dynamic thread may change during the lifetime of the thread. Notice that Θ is not part of the program specification. It is used only in the soundness proof. also note that we do not need the local guarantee $\hat{\mathtt{g}}$ in specifications for threads in $\mathbb{Q}$, because they need to fulfill the guarantee $\mathbb{G}$ and use $\mathbb{G}$ as the initial local guarantee when they are scheduled to resume execution.

5.3.3 Inference Rules

Well-formed programs. The following PROG rule shows the invariants that need to be maintained during program transitions.

$$\frac{\begin{array}{l}\mathbb{S} = (\mathbb{H}, \mathbb{R}) \quad \mathtt{t} = \mathbb{R}(\mathsf{rt}) \\ \psi \vdash \mathbb{C}\!:\!\psi \qquad \mathtt{p}\ \mathbb{S} \qquad \psi \vdash \{(\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}, \mathbb{G})\}\ \mathsf{pc} :\ \mathbb{C}[\mathsf{pc}] \\ \psi; \Theta; \hat{\mathtt{g}} \vdash \{(\mathbb{C}, \mathbb{S})\}\mathbb{Q} \quad \mathtt{NI}(\Theta\{\mathtt{t} \rightsquigarrow (\mathtt{p}, \mathbb{A}, \mathbb{G})\}, \mathbb{Q}\{\mathtt{t} \rightsquigarrow (\mathbb{R}, \mathsf{pc})\})\end{array}}{\psi \vdash (\mathbb{C}, \mathbb{S}, \mathbb{Q}, \mathsf{pc})}\ (\text{PROG})$$

The PROG rule requires that there be a quadruple $(\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}, \mathbb{G})$ specifying the current code block. Also there exists a Θ that specifies threads in $\mathbb{Q}$. As in CAP, we require that the code heap be well-formed with respect to ψ, the current program state satisfy $\mathtt{p}$, and that it be safe to execute the code block $\mathbb{C}[\mathsf{pc}]$ given the specification ψ and $(\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}, \mathbb{G})$. Here $\mathbb{C}[\mathsf{pc}]$ is defined as:

$$\mathbb{C}[\mathtt{f}] = \begin{cases} \mathbb{C}(\mathtt{f}) & \text{if } \mathbb{C}(\mathtt{f}) = \mathsf{j}\ \mathtt{f}' \text{ or } \mathbb{C}(\mathtt{f}) = \mathsf{exit} \\ \iota; \mathbb{C}[\mathtt{f}+1] & \text{otherwise.} \end{cases}$$

In addition, we check that $\mathbb{Q}$ satisfies the specification Θ, given the code heap, the current state and the local guarantee for the remaining state transition the current thread needs to make before it yields. Also we enforce that assumptions and guarantees for the current thread and threads in $\mathbb{Q}$ implies non-interference.

The non-interference assertion $\mathtt{NI}(\Theta, \mathbb{Q})$ requires that each dynamic thread be compatible with all the other. It is formally defined as:

$$\begin{aligned}\mathtt{NI}(\Theta, \mathbb{Q}) \triangleq\ & \forall \mathtt{t}_i, \mathtt{t}_j \in \mathit{dom}(\Theta). \forall \mathbb{H}, \mathbb{H}'. \\ & \mathtt{t}_i \neq \mathtt{t}_j \rightarrow \mathbb{G}_i\ \mathbb{R}_i\ \mathbb{H}\ \mathbb{H}' \rightarrow \mathbb{A}_j\ \mathbb{R}_j\ \mathbb{H}\ \mathbb{H}' \end{aligned} \tag{5.1}$$

where $(\_, \mathbb{A}_i, \mathbb{G}_i) = \Theta(\mathtt{t}_i)$, $(\_, \mathbb{A}_j, \mathbb{G}_j) = \Theta(\mathtt{t}_j)$, $(\mathtt{r}_i, \_) = \mathbb{Q}(\mathtt{t}_i)$ and $(\mathtt{r}_j, \_) = \mathbb{Q}(\mathtt{t}_j)$.

Well-formed dynamic threads. The DTHRDS rule ensures each dynamic thread in $\mathbb{Q}$ is in good shape with respect to the specification Θ, the current program state $\mathbb{S}$, and the transition $\hat{\text{g}}$ that the current thread need do before other threads can take control.

$$\frac{\begin{array}{l}(\mathbb{R}_k, \texttt{pc}_k) = \mathbb{Q}(\texttt{t}_k), \quad (\texttt{p}_k, \mathbb{A}_k, \mathbb{G}_k) = \Theta(\texttt{t}_k), \quad \text{for all } \texttt{t}_k \in dom(\mathbb{Q}) \\ \forall \mathbb{H}, \mathbb{H}'. \ \texttt{p}_k \ (\mathbb{H}, \mathbb{R}_k) \rightarrow \mathbb{A}_k \ \mathbb{R}_k \ \mathbb{H} \ \mathbb{H}' \rightarrow \texttt{p}_k \ (\mathbb{H}', \mathbb{R}_k) \\ \forall \mathbb{H}'. \ \hat{\text{g}} \ \mathbb{R} \ \mathbb{H} \ \mathbb{H}' \rightarrow \texttt{p}_k \ (\mathbb{H}', \mathbb{R}_k) \\ \psi \vdash \{(\texttt{p}_k, \mathbb{G}_k, \mathbb{A}_k, \mathbb{G}_k)\} \ \texttt{pc}_k : \ \mathbb{C}[\texttt{pc}_k]\end{array}}{\psi; \Theta; \hat{\text{g}} \vdash \{(\mathbb{C}, (\mathbb{H}, \mathbb{R}))\}\mathbb{Q}} \text{ (DTHRDS)}$$

The first line gives the specification of each thread when it reaches a yield point, and its execution context. The premise in line 2 requires that if it is safe for a thread to take control at certain state, it should be also safe to do so after any state transition satisfying its assumption. Note that the state transition will not affect the thread-private data in $\texttt{r}_k$. The next premise describes the relationship between the local guarantee of the current thread and the preconditions of other threads. For any transitions starting from the current state $(\mathbb{H}, \texttt{r})$, as long as it satisfies $\hat{\text{g}}$, it should be safe for other threads to take control at the result state. The last line requires that for each thread its remainder instruction sequence must be well formed. Note we use $\mathbb{G}_k$ as the local guarantee because after yield, a thread starts a new "atomic transition" described by its global guarantee.

Well-formed code heap. The CDHP rule is similar to the well-formed code heap rules in CAP or SCAP: a code heap is well formed if every instruction sequence is well-formed with respect to its corresponding specification in ψ.

$$\frac{\text{for all } (\texttt{f}, \theta) \in \psi' : \qquad \psi \vdash \{\theta\} \ \texttt{f} : \ \mathbb{C}(\texttt{f})}{\psi \vdash \mathbb{C} : \psi'} \text{ (CDHP)}$$

Thread creation. The FORK rule describes constraints on new thread creation, which enforces the non-interference between the new thread and existing threads.

$$\frac{\begin{array}{l}(\mathtt{f}', (\mathtt{p}', \mathbb{G}', \mathbb{A}', \mathbb{G}')) \in \psi \\ \mathbb{A} \Rightarrow \mathbb{A}'' \quad \mathbb{G}'' \Rightarrow \mathbb{G} \quad (\mathbb{A} \vee \mathbb{G}'') \Rightarrow \mathbb{A}' \quad \mathbb{G}' \Rightarrow (\mathbb{G} \wedge \mathbb{A}'') \\ \forall \mathbb{R}, \mathbb{H}, \mathbb{R}', \mathbb{H}'. \ (\mathbb{R}(\mathsf{rt}) \neq \mathbb{R}'(\mathsf{rt}) \wedge \mathbb{R}(\mathtt{r}_s) = \mathbb{R}'(\mathsf{ra})) \rightarrow \mathtt{p}\ (\mathbb{H}, \mathbb{R}) \rightarrow \hat{\mathtt{g}}\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}' \rightarrow \mathtt{p}'\ (\mathbb{H}', \mathbb{R}') \\ \forall \mathbb{R}, \mathbb{H}, \mathbb{H}'. \ \mathtt{p}'\ (\mathbb{H}, \mathbb{R}) \rightarrow \mathbb{A}'\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}' \rightarrow \mathtt{p}'\ (\mathbb{H}, \mathbb{R}') \\ \psi \vdash \{(\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}'', \mathbb{G}'')\}\ \mathtt{f}{+}1 : \mathbb{I}\end{array}}{\psi \vdash \{(\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}, \mathbb{G})\}\ \mathtt{f} : \ \mathsf{fork}\ \mathtt{f}', \mathtt{r}_s; \mathbb{I}} \ (\textsc{fork})$$

As explained in Section 5.2.1, the parent thread can change its specification to reflect the change of the environment. To maintain the non-interference invariant, constraints between specifications of the parent and child threads have to be satisfied, as described in Lemma 5.4. Here we enforce these constraints by premises in line 2, where $(\mathbb{A} \vee \mathbb{G}'') \Rightarrow \mathbb{A}'$ is the shorthand for:

$$\begin{array}{r}\forall \mathbb{H}, \mathbb{H}', \mathbb{H}'', \mathbb{R}, \mathbb{R}'. \ \mathtt{p}\ (\mathbb{H}, \mathbb{R}) \rightarrow \mathbb{R}(\mathsf{rt}) \neq \mathbb{R}'(\mathsf{rt}) \rightarrow \mathbb{R}(\mathtt{r}_s) = \mathbb{R}'(\mathsf{ra}) \\ \rightarrow (\mathbb{A} \vee \mathbb{G}'')\ \mathbb{R}\ \mathbb{H}'\ \mathbb{H}'' \rightarrow \mathbb{A}'\ \mathbb{R}'\ \mathbb{H}'\ \mathbb{H}'',\end{array}$$

and $\mathbb{G}' \Rightarrow (\mathbb{G} \wedge \mathbb{A}'')$ for:

$$\begin{array}{r}\forall \mathbb{H}, \mathbb{H}', \mathbb{H}'', \mathbb{R}, \mathbb{R}'. \ \mathtt{p}\ (\mathbb{H}, \mathbb{R}) \rightarrow \mathbb{R}(\mathsf{rt}) \neq \mathbb{R}'(\mathsf{rt}) \rightarrow \mathbb{R}(\mathtt{r}_s) = \mathbb{R}'(\mathsf{ra}) \\ \rightarrow \mathbb{G}'\ \mathbb{R}'\ \mathbb{H}'\ \mathbb{H}'' \rightarrow (\mathbb{G} \wedge \mathbb{A}'')\ \mathbb{R}\ \mathbb{H}'\ \mathbb{H}''.\end{array}$$

Above non-interference checks use the extra knowledge that:

- the new thread id is different with its parent's, *i.e.*, $\mathbb{R}(\mathsf{rt}) \neq \mathbb{R}'(\mathsf{rt})$;
- the argument of the new thread comes from the parent's register $\mathtt{r}_s$, *i.e.*, $\mathbb{R}(\mathtt{r}_s) = \mathbb{R}'(\mathsf{ra})$;
- the parent's register file satisfies the precondition, *i.e.*, $\mathtt{p}\ (\mathbb{H}, \mathbb{R})$.

In most cases, the programmer can just pick $(\mathbb{A} \vee \widehat{\mathbb{G}'})$ and $(\mathbb{G} \wedge \widehat{\mathbb{A}'})$ as $\mathbb{A}''$ and $\mathbb{G}''$ respectively, where $\widehat{\mathbb{G}'}$ and $\widehat{\mathbb{A}'}$ are instantiations of $\mathbb{G}'$ and $\mathbb{A}'$ using the value of the child's argument.

The premise in line 3 says that after the current thread completes the transition described by $\hat{\mathtt{g}}$, it should be safe for the new thread to take control with its new register file $(\mathbb{R}')$, whose relationship between the parents register file $\mathbb{R}$ is satisfied.

The next premise requires that the precondition $\mathtt{p}'$ for the new thread is stable with respect to its assumption $\mathbb{A}'$. Therefore, the new thread can be scheduled to run after any number of state transitions, as long as each of them satisfies $\mathbb{A}'$.

The last premise checks the well-formedness of the remainder instruction sequence. Since the fork instruction does not change states, we need not change the precondition $\mathtt{p}$ and $\hat{\mathtt{g}}$.

Yielding and termination.

$$\frac{\begin{array}{l}\forall \mathbb{R}, \mathbb{H}, \mathbb{H}'.\ \mathtt{p}\ (\mathbb{H}, \mathbb{R}) \to \mathbb{A}\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}' \to \mathtt{p}\ (\mathbb{H}, \mathbb{R}') \quad \forall \mathbb{R}, \mathbb{H}.\ \mathtt{p}\ (\mathbb{H}, \mathbb{R}) \to \hat{\mathtt{g}}\ \mathbb{R}\ \mathbb{H}\ \mathbb{H} \\ \psi \vdash \{(\mathtt{p}, \mathbb{G}, \mathbb{A}, \mathbb{G})\}\ \mathtt{f}{+}1 : \ \mathbb{I}\end{array}}{\psi \vdash \{(\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}, \mathbb{G})\}\ \mathtt{f} : \ \mathsf{yield}; \mathbb{I}} \ (\text{YIELD})$$

The YIELD rule requires that it is safe for the yielding thread to take back control after any state transition satisfying the assumption $\mathbb{A}$. Also the current thread cannot yield until it completes the required state transition, *i.e.,* an identity transition satisfies the local guarantee $\hat{\mathtt{g}}$. Lastly, one must certify the remainder instruction sequence with the local guarantee reset to $\mathbb{G}$.

$$\frac{\forall \mathbb{R}, \mathbb{H}.\ \mathtt{p}\ (\mathbb{H}, \mathbb{R}) \to \hat{\mathtt{g}}\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}}{\psi \vdash \{(\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}, \mathbb{G})\}\ \mathtt{f} : \ \mathsf{exit}; \mathbb{I}} \ (\text{EXIT})$$

The EXIT rule is simple: it is safe for the current thread to terminate its execution only after it finishes the required transition described by $\hat{\mathtt{g}}$, which is an identity transition. The rest of the current block following exit will not be executed, so we do not need to check its well-formedness.

Other instructions.

$$\frac{\begin{array}{l}\iota \in \{\mathsf{addu}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t,\ \mathsf{addiu}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{w},\ \mathsf{subu}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t,\ \mathsf{lw}\ \mathtt{r}_d, \mathtt{w}(\mathtt{r}_s),\ \mathsf{sw}\ \mathtt{r}_t, \mathtt{w}(\mathtt{r}_s)\}, \mathtt{r}_d \notin \{\mathsf{rt}, \mathsf{ra}\} \\ \mathtt{p} \Rightarrow \mathtt{p}' \circ \mathsf{NextS}_{(\mathtt{f},\iota)} \qquad \forall \mathbb{S}, \mathbb{S}'.\ \mathtt{p}\ \mathbb{S} \to \hat{\mathtt{g}}'\ \mathsf{NextS}_{(\mathtt{f},\iota)}(\mathbb{S})\ \mathbb{S}' \to \hat{\mathtt{g}}\ \mathbb{S}\ \mathbb{S}' \\ \mathbb{A} \Rightarrow \mathbb{A}' \circ \mathsf{NextS}_{(\mathtt{f},\iota)} \qquad \mathbb{G}' \circ \mathsf{NextS}_{(\mathtt{f},\iota)} \Rightarrow \mathbb{G} \qquad \psi \vdash \{(\mathtt{p}', \hat{\mathtt{g}}', \mathbb{A}', \mathbb{G}')\}\ \mathtt{f}{+}1 : \mathbb{I}\end{array}}{\psi \vdash \{(\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}, \mathbb{G})\}\ \mathtt{f} :\ \iota; \mathbb{I}} \ (\textsc{simp})$$

The SIMP rule covers the verification of instruction sequences starting with a simple command such as an arithmetic instruction or a memory operation instruction. We require that the program not update registers rt and ra. In these cases, one must find an intermediate precondition $(\mathtt{p}', \hat{\mathtt{g}}', \mathbb{A}', \mathbb{G}')$ under which the remainder instruction sequence $\mathbb{I}$ is well-formed.

The intermediate assertion $\mathtt{p}'$ must hold on the updated machine state, and the intermediate guarantee $\hat{\mathtt{g}}'$ applied to the updated machine state must be no weaker than the current guarantee $\hat{\mathtt{g}}$ applied to the current state. Recall the definition of $\mathtt{p} \Rightarrow \mathtt{p}' \circ \mathsf{NextS}_{(\mathtt{f},\iota)}$ in Equaiton 3.1 (Section 3.1.1).

Since $\mathbb{A}$ and $\mathbb{G}$ are parameterized over $\mathbb{R}$, which will be changed by the instruction ι, we may change $\mathbb{A}$ and $\mathbb{G}$ to ensure the assumption does not become stronger and the guarantee does not become weaker. Here we use $\mathbb{A} \Rightarrow \mathbb{A}' \circ \mathsf{NextS}_{(\mathtt{f},\iota)}$ as a short hand for:

$$\begin{array}{l}\forall \mathbb{R}_0, \mathbb{R}_1, \mathbb{H}_0, \mathbb{H}_1.\ \mathtt{p}\ (\mathbb{H}_0, \mathbb{R}_0) \wedge (\mathsf{NextS}_{(\mathtt{f},\iota)}(\mathbb{H}_0, \mathbb{R}_0) = (\mathbb{H}_1, \mathbb{R}_1)) \\ \qquad\qquad \to \forall \mathbb{H}, \mathbb{H}'.\ \mathbb{A}\ \mathbb{R}_0\ \mathbb{H}\ \mathbb{H}' \to \mathbb{A}'\ \mathbb{R}_1\ \mathbb{H}\ \mathbb{H}'\,.\end{array}$$

The definition for $\mathbb{G}' \circ \mathsf{NextS}_{(\mathtt{f},\iota)} \Rightarrow \mathbb{G}$ is similar.

$$\frac{\begin{array}{l}(\mathtt{f}',\ (\mathtt{p}',\hat{\mathtt{g}}',\mathbb{A}',\mathbb{G}'))\in\psi\\ \mathtt{p}\Rightarrow\mathtt{p}'\quad \mathbb{A}\Rightarrow\mathbb{A}'\quad \mathbb{G}'\Rightarrow\mathbb{G}\\ \forall\mathbb{R},\mathbb{H},\mathbb{H}'.\ \mathtt{p}\ (\mathbb{H},\mathbb{R})\rightarrow\hat{\mathtt{g}}'\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}'\rightarrow\hat{\mathtt{g}}\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}'\end{array}}{\psi\vdash\{(\mathtt{p},\hat{\mathtt{g}},\mathbb{A},\mathbb{G})\}\,\mathtt{f}:\ \mathtt{j}\ \mathtt{f}'}\ \text{(J)}$$

The J rule checks the specification of the target instruction sequence. As mentioned before, each instruction sequence can have its own specification, independent of the thread that will jump to it. It is safe for a thread to execute an instruction sequence as long as executing the instruction sequence does not require a stronger assumption than the thread's assumption $\mathbb{A}$, nor does it break the guarantee $\mathbb{G}$ of the thread.

$$\frac{\begin{array}{l}(\mathtt{f}',(\mathtt{p}',\hat{\mathtt{g}}',\mathbb{A}',\mathbb{G}'))\in\psi\quad \mathbb{A}\Rightarrow\mathbb{A}'\quad \mathbb{G}'\Rightarrow\mathbb{G}\\ \forall\mathbb{S}.\ \mathtt{p}\ \mathbb{S}\rightarrow(\mathbb{S}.\mathbb{R}(\mathtt{r}_s)>0)\rightarrow\mathtt{p}'\ \mathbb{S}\\ \forall\mathbb{R},\mathbb{H},\mathbb{H}'.\ \mathtt{p}\ (\mathbb{H},\mathbb{R})\rightarrow(\mathbb{R}(\mathtt{r}_s)>0)\rightarrow\hat{\mathtt{g}}'\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}'\rightarrow\hat{\mathtt{g}}\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}'\\ \forall\mathbb{S}.\ \mathtt{p}\ \mathbb{S}\rightarrow(\mathbb{S}.\mathbb{R}(\mathtt{r}_s)\leq 0)\rightarrow\mathtt{p}''\ \mathbb{S}\\ \forall\mathbb{R},\mathbb{H},\mathbb{H}'.\ \mathtt{p}\ (\mathbb{H},\mathbb{R})\rightarrow(\mathbb{R}(\mathtt{r}_s)\leq 0)\rightarrow\hat{\mathtt{g}}''\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}'\rightarrow\hat{\mathtt{g}}\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}'\\ \psi\vdash\{(\mathtt{p}'',\hat{\mathtt{g}}'',\mathbb{A},\mathbb{G})\}\,\mathtt{f}{+}1:\ \mathbb{I}\end{array}}{\psi\vdash\{(\mathtt{p},\hat{\mathtt{g}},\mathbb{A},\mathbb{G})\}\,\mathtt{f}:\ \mathsf{bgtz}\ \mathtt{r}_s,\mathtt{f}';\mathbb{I}}\ \text{(BGTZ)}$$

$$\frac{\begin{array}{l}(\mathtt{f}',(\mathtt{p}',\hat{\mathtt{g}}',\mathbb{A}',\mathbb{G}'))\in\psi\quad \mathbb{A}\Rightarrow\mathbb{A}'\quad \mathbb{G}'\Rightarrow\mathbb{G}\\ \forall\mathbb{S}.\ \mathtt{p}\ \mathbb{S}\rightarrow(\mathbb{S}.\mathbb{R}(\mathtt{r}_s)=\mathbb{S}.\mathbb{R}(\mathtt{r}_t))\rightarrow\mathtt{p}'\ \mathbb{S}\\ \forall\mathbb{R},\mathbb{H},\mathbb{H}'.\ \mathtt{p}\ (\mathbb{H},\mathbb{R})\rightarrow(\mathbb{R}(\mathtt{r}_s)=\mathbb{R}(\mathtt{r}_t))\rightarrow\hat{\mathtt{g}}'\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}'\rightarrow\hat{\mathtt{g}}\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}'\\ \forall\mathbb{S}.\ \mathtt{p}\ \mathbb{S}\rightarrow(\mathbb{S}.\mathbb{R}(\mathtt{r}_s)\neq\mathbb{S}.\mathbb{R}(\mathtt{r}_t))\rightarrow\mathtt{p}''\ \mathbb{S}\\ \forall\mathbb{R},\mathbb{H},\mathbb{H}'.\ \mathtt{p}\ (\mathbb{H},\mathbb{R})\rightarrow(\mathbb{R}(\mathtt{r}_s)\neq\mathbb{R}(\mathtt{r}_t))\rightarrow\hat{\mathtt{g}}''\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}'\rightarrow\hat{\mathtt{g}}\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}'\\ \psi\vdash\{(\mathtt{p}'',\hat{\mathtt{g}}'',\mathbb{A},\mathbb{G})\}\,\mathtt{f}{+}1:\ \mathbb{I}\end{array}}{\psi\vdash\{(\mathtt{p},\hat{\mathtt{g}},\mathbb{A},\mathbb{G})\}\,\mathtt{f}:\ \mathsf{beq}\ \mathtt{r}_s,\mathtt{r}_t,\mathtt{f}';\mathbb{I}}\ \text{(BEQ)}$$

Rules for conditional branching instructions are similar to the J rule, which are straightforward to understand.

5.3.4 Soundness of CMAP

The soundness of CMAP inference rules with respect to the operational semantics of the machine is established following the syntactic approach of proving type soundness [99]. From the "progress" and "preservation" lemmas, we can guarantee that, given a well-formed program under compatible assumptions and guarantees, the current instruction sequence will be able to execute without getting "stuck". As before, we define $\mathbb{P} \longmapsto^n \mathbb{P}'$ as the relation of n-step program transitions. The soundness of CMAP is formally stated as Theorem 5.7.

Lemma 5.5 (Progress)
For all ψ and $\mathbb{P}$, if $\psi \vdash \mathbb{P}$, then there exists $\mathbb{P}'$ such that $\mathbb{P} \longmapsto \mathbb{P}'$.

Proof. We prove the lemma by case analysis of all ι at pc. If ι is one of add, sub, movi, fork, exit, or yield instructions, the program can always make a step by the definition of the operational semantics. If ι is a ld or st instruction, the side conditions for making a step, as defined by the operational semantics, are established by the SIMP rule. If ι is bgt, beq, or jd, the operational semantics may fetch a code block from the code heap; such a code block exists by the inversion of the CDHP rule. $\square$

Lemma 5.6 (Preservation)
For all ψ, $\mathbb{P}$ and $\mathbb{P}'$, if $\psi \vdash \mathbb{P}$ and $\mathbb{P} \longmapsto \mathbb{P}'$, then we have $\psi \vdash \mathbb{P}'$.

kwProof. Suppose $\mathbb{P} = (\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{Q}, \mathsf{pc})$. Let $\mathsf{t} = \mathbb{R}(\mathsf{rt})$. By the assumption $\psi \vdash \mathbb{P}$ and the inversion of the PROG rule, we know that there exist p, $\hat{\mathsf{g}}$, $\mathbb{A}$, $\mathbb{G}$ and Θ such that

1. $\psi \vdash \mathbb{C} : \psi$;

2. $\mathsf{p}\ \mathbb{S}$;

3. $\psi \vdash \{(\mathsf{p}, \hat{\mathsf{g}}, \mathbb{A}, \mathbb{G})\}\ \mathsf{pc} :\ \mathbb{C}[\mathsf{pc}]$;

4. $\psi; \Theta; \hat{\mathsf{g}} \vdash \{(\mathbb{C}, \mathbb{S})\}\mathbb{Q}$;

5. $\mathtt{NI}(\Theta\{\mathtt{t} \leadsto (\mathtt{p}, \mathbb{A}, \mathbb{G})\}, \mathbb{Q}\{\mathtt{t} \leadsto (\mathbb{R}, \mathsf{pc})\})$ (see Section 5.3.3 for the definition of NI).

We prove the lemma by case analysis over $\mathbb{C}[\mathsf{pc}]$. Here we only give the detailed proof of cases where $\mathbb{C}[\mathsf{pc}]$ starts with fork or yield. Proofs for the rest of cases are trivial.

Case $\mathbb{C}[\mathsf{pc}] = \mathsf{fork}\ \mathtt{f}, \mathtt{r}_s; \mathbb{I}$.

By the operational semantics, we know that $\mathbb{P}' = (\mathbb{C}, \mathbb{S}, \mathbb{Q}\{\mathtt{t}' \leadsto (\mathbb{R}', \mathtt{f})\}, \mathsf{pc}+1)$, where $\mathtt{t}' \notin dom(\mathbb{Q})$, $\mathtt{t}' \neq \mathtt{t}$, and $\mathbb{R}' = \{\mathtt{r}_0 \leadsto \_, \ldots, \mathtt{r}_{15} \leadsto \_, \mathsf{rt} \leadsto \mathtt{t}, \mathsf{ra} \leadsto \mathbb{R}(\mathtt{r}_s)\}$. According to 3 and the inversion of the FORK rule, we know that there exist $\mathtt{p}'$, $\mathbb{A}'$, $\mathbb{G}'$, $\mathbb{A}''$ and $\mathbb{G}''$ such that

f.1 $(\mathtt{f}, (\mathtt{p}', \mathbb{G}', \mathbb{A}', \mathbb{G}')) \in \psi$;

f.2 $\mathbb{A} \Rightarrow \mathbb{A}''$, $\mathbb{G}'' \Rightarrow \mathbb{G}$, $(\mathbb{A} \vee \mathbb{G}'') \Rightarrow \mathbb{A}'$, and $\mathbb{G}' \Rightarrow (\mathbb{G} \wedge \mathbb{A}'')$;

f.3 $\forall \mathbb{R}, \mathbb{H}, \mathbb{R}', \mathbb{H}'.\ (\mathbb{R}(\mathsf{rt}) \neq \mathbb{R}'(\mathsf{rt}) \wedge \mathbb{R}(\mathtt{r}_s) = \mathbb{R}'(\mathsf{ra})) \rightarrow \mathtt{p}\ (\mathbb{H}, \mathbb{R}) \rightarrow \hat{\mathtt{g}}\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}' \rightarrow \mathtt{p}'\ (\mathbb{H}', \mathbb{R}')$;

f.4 $\psi \vdash \{(\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}'', \mathbb{G}'')\}\ \mathsf{pc}+1 : \mathbb{I}$;

f.5 $\forall \mathbb{R}, \mathbb{H}, \mathbb{H}'.\ \mathtt{p}'\ (\mathbb{H}, \mathbb{R}) \rightarrow \mathbb{A}'\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}' \rightarrow \mathtt{p}'\ (\mathbb{H}, \mathbb{R}')$

Then we let $\Theta' = \Theta\{\mathtt{t}' \leadsto (\mathtt{p}', \mathbb{G}', \mathbb{A}', \mathbb{G}')\}$.

According to the PROG rule, to prove $\psi \vdash \mathbb{P}'$, we need to prove the following:

- $\psi \vdash \mathbb{C} : \psi$, which trivially follows 1. Since $\mathbb{C}$ never changes, we will omit the proof of it in the following cases;
- $\mathtt{p}\ \mathbb{S}$, which follows 2;
- $\psi \vdash \{(\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}'', \mathbb{G}'')\}\ \mathsf{pc}+1 :\ \mathbb{I}$, which follows f.4.
- $\psi; \Theta'; \hat{\mathtt{g}} \vdash \{(\mathbb{C}, \mathbb{S})\}\mathbb{Q}\{\mathtt{t}' \leadsto (\mathbb{R}', \mathtt{f})\}$. By 4 and the inversion of the DTHRDS rule, we need to check the following for the new thread $\mathtt{t}'$:
  - $\forall \mathbb{H}', \mathbb{H}''.\mathtt{p}'\ (\mathbb{H}', \mathtt{r}') \rightarrow \mathbb{A}'\ \mathtt{r}'\ \mathbb{H}'\ \mathbb{H}'' \rightarrow \mathtt{p}'\ (\mathbb{H}'', \mathtt{r}')$. By f.5.
  - $\psi \vdash \{(\mathtt{p}', \mathbb{G}', \mathbb{A}', \mathbb{G}')\}\ \mathtt{f} :\ \mathbb{C}[\mathtt{f}]$. By 1 and f.1.

– $\forall \mathbb{H}'.\ \hat{\mathtt{g}}\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}' \rightarrow \mathtt{p}'\ (\mathbb{H}', \mathbb{R}')$. By 2, and f.3.

- $\mathtt{NI}(\Theta'\{\mathtt{t} \rightsquigarrow (\mathtt{p}, \mathbb{A}, \mathbb{G})\}, \mathbb{Q}'\{\mathtt{t} \rightsquigarrow (\mathbb{R}, \mathsf{pc})\})$, where $\mathbb{Q}' = \mathbb{Q}\{\mathtt{t}' \rightsquigarrow (\mathbb{R}', \mathtt{f})\}$. By 2, 5, f.2, and Lemma 5.4.

Case $\mathbb{C}[\mathsf{pc}] = \mathsf{yield}; \mathbb{I}$.

By the operational semantics, we know that $\mathbb{P}' = (\mathbb{C}, (\mathbb{H}, \mathbb{R}'), \mathbb{Q}' \setminus \{\mathtt{t}'\}, \mathsf{pc}')$, where $\mathbb{Q}' = \mathbb{Q}\{\mathtt{t} \rightsquigarrow (\mathbb{R}, \mathsf{pc}{+}1)\}, \mathtt{t}' \in dom(\mathbb{Q}')$ and $(\mathbb{R}', \mathsf{pc}') = \mathbb{Q}'(\mathtt{t}')$.

If $\mathtt{t}' = \mathtt{t}$, the yield instruction is like a no-op instruction. The proof is trivial. So we just consider the case that $\mathtt{t}' \in dom(\mathbb{Q})$ and $(\mathtt{r}', \iota') = \mathbb{Q}(\mathtt{t}')$.

By 4 we know that there exist $\mathtt{p}'$, $\mathtt{g}'$ and $\mathbb{G}'$ such that $(\mathtt{p}', \mathbb{A}', \mathbb{G}') = \Theta(\mathtt{t}')$. We let $\Theta' = (\Theta\{\mathtt{t} \rightsquigarrow (\mathtt{p}, \mathbb{A}, \mathbb{G})\}) \setminus \{\mathtt{t}'\}$. According to the PROG rule, to prove $lspec \vdash \mathbb{P}'$, we need to prove:

- $\mathtt{p}\ (\mathbb{H}, \mathtt{r}')$.

  By 4 and the inversion of the DTHRDS rule, we know that $\forall \mathbb{H}'.\ \hat{\mathtt{g}}\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}' \rightarrow \mathtt{p}'\ (\mathbb{H}', \mathbb{R}')$, also by 2, 3 and the inversion of the YIELD rule we get $\hat{\mathtt{g}}\ \ \mathbb{R}\ \mathbb{H}\ \mathbb{H}$. Therefore we can prove $\mathtt{p}'\ (\mathbb{H}, \mathtt{r}')$.

- $\psi \vdash \{(\mathtt{p}', \mathbb{G}', \mathbb{A}', \mathbb{G}')\}\ \mathbb{C}[\mathsf{pc}']$. By 4 and the inversion of the DTHRDS rule.

- $\psi; \Theta'; \mathbb{G}' \vdash \{(\mathbb{C}, (\mathbb{H}, \mathtt{r}'))\}\mathbb{Q}' \setminus \{\mathtt{t}'\}$ (where $\mathbb{Q}' = \mathbb{Q}\{\mathtt{t} \rightsquigarrow (\mathbb{R}, \mathsf{pc}{+}1)\}$ and $\mathtt{t}' \neq \mathtt{t}$). To prove this, we only check the following for the yielding thread:

  – $\forall \mathbb{H}, \mathbb{H}'.\ \mathtt{p}\ (\mathbb{H}, \mathbb{R}) \rightarrow \mathbb{A}\ \mathbb{R}\ \mathbb{H}\ \mathbb{H}' \rightarrow \mathtt{p}\ (\mathbb{H}', \mathbb{R})$. This follows 3 and the inversion of the YIELD rule.

  – $\psi \vdash \{(\mathtt{p}, \mathbb{G}, \mathbb{A}, \mathbb{G})\}\ \mathsf{pc}{+}1 :\ \mathbb{I}$. This follows 3 and the inversion of the YIELD rule.

  – $\forall \mathbb{H}'.\ \mathbb{G}'\ \mathbb{R}'\ \mathbb{H}\ \mathbb{H}' \rightarrow \mathtt{p}\ (\mathbb{H}', \mathbb{R})$. By 5 we know $\forall \mathbb{H}', \mathbb{H}''.\mathbb{G}'\ \mathtt{r}'\ \mathbb{H}'\ \mathbb{H}'' \rightarrow \mathbb{A}\ \mathtt{r}\ \mathbb{H}'\ \mathbb{H}''$. Therefore, we only need prove: $\forall \mathbb{H}''.\mathbb{A}\ \mathbb{H}\ \mathbb{H}'' \rightarrow \mathtt{p}\ (\mathbb{H}'', \mathtt{r})$. By 3, the inversion of the YIELD rule, and 2 we get it.

- $\mathtt{NI}(\Theta'\{\mathtt{t}'\leadsto(\mathtt{p}',\mathbb{A}',\mathbb{G}')\},\mathbb{Q}\{\mathtt{t}\leadsto\mathtt{r},\mathsf{pc}+1)\})$, which follows 5.

  Note that $\mathbb{Q}\{\mathtt{t}\leadsto\mathtt{r},\mathsf{pc}+1)\}) = (\mathbb{Q}'\setminus\{\mathtt{t}'\})\{\mathtt{t}'\leadsto(\mathtt{r}',\mathsf{pc}')\}$, where $\mathbb{Q}' = \mathbb{Q}\{\mathtt{t}\leadsto(\mathbb{R},\mathsf{pc}+1)\}$ and $(\mathbb{R}',\mathsf{pc}') = \mathbb{Q}'(\mathtt{t}')$.

Case other cases, the proof is trivial. The whole proof is formalized in Coq [31]. □

From the progress and preservation lemmas, we can prove that execution of $\mathbb{P}$ would never get stuck as long as $\mathbb{P}$ satisfy the global invariant formalized by the PROG rule, *i.e.,* $\psi \vdash \mathbb{P}$. *More importantly*, the global invariant holds over every intermediate machine configuration during execution, from which we can derive rich properties of programs. In the following soundness theorem, we show the support of partial correctness. In addition to non-stuckness, the soundness theorem says that the assertions specified in ψ holds when the corresponding code labels are reached through jump or branch instructions.

Theorem 5.7 (Soundness)

For all n, ψ and $\mathbb{P}$, if $\psi \vdash \mathbb{P}$, then there exists $\mathbb{P}'$ such that $\mathbb{P} \longmapsto^n \mathbb{P}'$. Also, suppose $\mathbb{P}' = (\mathbb{C},\mathbb{S},\mathbb{Q},\mathsf{pc})$, we have

(1) if $\mathbb{C}(\mathsf{pc}) = \mathsf{j}\ \mathtt{f}$, then there exist $\mathtt{p}$, $\hat{\mathtt{g}}$, $\mathbb{A}$ and $\mathbb{G}$ such that $(\mathtt{f},(\mathtt{p},\hat{\mathtt{g}},\mathbb{A},\mathbb{G})) \in \psi$ and $\mathtt{p}\ \mathbb{S}$;

(2) if $\mathbb{C}(\mathsf{pc}) = \mathsf{beq}\ \mathtt{r}_s,\mathtt{r}_d,\mathtt{f}$ and $\mathbb{S}.\mathbb{R}(\mathtt{r}_s) = \mathbb{S}.\mathbb{R}(\mathtt{r}_d)$, then there exist $\mathtt{p}$, $\hat{\mathtt{g}}$, $\mathbb{A}$ and $\mathbb{G}$ such that $(\mathtt{f},(\mathtt{p},\hat{\mathtt{g}},\mathbb{A},\mathbb{G})) \in \psi$ and $\mathtt{p}\ \mathbb{S}$;

(3) if $\mathbb{C}(\mathsf{pc}) = \mathsf{bgtz}\ \mathtt{r}_s,\mathtt{f}$ and $\mathbb{S}.\mathbb{R}(\mathtt{r}_s) > 0$, then there exist $\mathtt{p}$, $\hat{\mathtt{g}}$, $\mathbb{A}$ and $\mathbb{G}$ such that $(\mathtt{f},(\mathtt{p},\hat{\mathtt{g}},\mathbb{A},\mathbb{G})) \in \psi$ and $\mathtt{p}\ \mathbb{S}$.

Proof. The proof is very simple. By doing induction over n and applying the progress and preservation lemmas, we can prove a conclusion stronger than non-stuckness, *i.e.,* we can prove $\psi \vdash \mathbb{P}'$ in addition to the existence of $\mathbb{P}'$. Then propositions (1), (2) and (3) become trivial by inversion of the PROG rule. □

5.4 Examples

5.4.1 Unbounded Dynamic Thread Creation

In Figure 5.3 we showed a small program main2 which spawns child threads within a while loop. This kind of unbounded dynamic thread creation cannot be supported using the cobegin/coend structure. We show how such a program is specified and verified using our logic. To simplify the specification, we trivialize the function f and g and let $\mathtt{f}(i) = i$ and $\mathtt{g}(x, \_) = x + 1$.

We assume the high-level program works in a preemptive mode. Figure 5.9 shows the CMAP implementation, where yield instructions are inserted to simulate the preemption. This also illustrates that our logic is general enough to simulate the preemptive thread model.

To certify the safety property of the program, the programmer needs to find a specification for the code heap. In Figure 5.9 we show definitions of assertions that are used to specify the program. Proof sketches are also inserted in the program. Following our convention set in Chapter 4, we use the register name $r and [l] to represent values contained in r and the memory location l, respectively. We also use primed representations $r′ and [l]′ to represent values in the resulting state of a state transition (*i.e.,* the second argument of $\hat{\mathsf{g}}$, $\mathbb{A}$ and $\mathbb{G}$).

The following formulae show the specifications of code heap, and the initial memory and program counter pc.

$$
\begin{aligned}
\psi \;\triangleq \{\; & (\mathtt{main}, (\mathsf{True}, \mathbb{G}_0, \mathbb{A}_0, \mathbb{G}_0)\;),\\
& (\mathtt{chld}, (\mathsf{p}, \mathbb{G}, \mathbb{A}, \mathbb{G})\;),\\
& (\mathtt{loop}, (\mathsf{p}', \mathbb{G}_1, \mathbb{A}_1, \mathbb{G}_1)\;),\\
& (\mathtt{cont}, (\mathsf{p}_3, \mathbb{G}_3, \mathbb{A}_3, \mathbb{G}_3)\;)\\
\}\; &
\end{aligned}
$$

$\mathsf{p} \triangleq 0 \le \texttt{\$ra} < 100 \qquad \mathbb{A} \triangleq [\texttt{data}+\texttt{\$ra}] = [\texttt{data}+\texttt{\$ra}]'$
$\mathbb{G} \triangleq \forall i.((0 \le i < 100) \wedge (i \ne \texttt{\$ra})) \to ([\texttt{data}+i] = [\texttt{data}+i]')$

$\mathbb{A}_0 \triangleq \forall i.(0 \le i < 100) \to ([\texttt{data}+i] = [\texttt{data}+i]') \qquad \mathbb{G}_0 \triangleq \mathsf{True}$

$\mathbb{A}_1 \triangleq \forall i.((0 \le i < 100) \wedge (i \ge \texttt{\$r}_1)) \to ([\texttt{data}+i] = [\texttt{data}+i]')$
$\mathbb{G}_1 \triangleq \forall i.((0 \le i < 100) \wedge (i < \texttt{\$r}_1)) \to ([\texttt{data}+i] = [\texttt{data}+i]')$
$\mathbb{A}_2 \triangleq \forall i.((0 \le i < 100) \wedge (i > \texttt{\$r}_1)) \to ([\texttt{data}+i] = [\texttt{data}+i]')$
$\mathbb{G}_2 \triangleq \forall i.((0 \le i < 100) \wedge (i \le \texttt{\$r}_1)) \to ([\texttt{data}+i] = [\texttt{data}+i]')$

$\mathbb{A}_3 \triangleq \mathsf{True} \qquad \mathbb{G}_3 \triangleq \forall i.(0 \le i < 100) \to ([\texttt{data}+i] = [\texttt{data}+i]')$
$\mathsf{p}' \triangleq (0 \le \texttt{\$r}_1 < 100) \wedge (\texttt{\$r}_2 = 1) \wedge (\texttt{\$r}_3 = 100) \qquad \mathsf{p}_3 \triangleq \texttt{\$r}_1 = 100$

```
main : −{(True, G_0, A_0, G_0)}
         addiu $r_1, $r_0, 0
         addiu $r_2, $r_0, 1
         addiu $r_3, $r_0, 100
         j loop
loop : −{(p', G_1, A_1, G_1)}
         beq $r_1, $r_3, cont
         yield
         sw $r_1, data($r_1)
         yield
         fork chld, $r_1
         −{(p', G_1, A_2, G_2)}
         yield
         addu $r_1, $r_1, $r_2
         −{(p', G_1, A_1, G_1)}
         yield
         j loop
cont : −{(p_3, G_3, A_3, G_3)}
         ...

chld : −{(p, G, A, G)}
         addiu $r_1, $r_0, data
         yield
         addu $r_1, $r_1, $ra
         yield
         lw $r_2, 0($r_1)
         yield
         addiu $r_3, $r_0, 1
         yield
         addu $r_2, $r_2, $r_3
         yield
         sw $r_2, 0($r_1)
         exit
```

Figure 5.9: Loop: the CMAP program

$$\text{Initial } \mathbb{H} \triangleq \{\texttt{data} \leadsto \_, \ldots, \texttt{data} + 99 \leadsto \_\}$$

$$\text{Initial } \texttt{pc} \triangleq \texttt{main}$$

For each child thread, it is natural to assume that no other threads will touch its share of the data entry and guarantee that other threads' data entry will not be changed, as specified by $\mathbb{A}$ and $\mathbb{G}$. For the main thread, it assumes at the beginning that no other threads in the environment will change any of the data entries ($\mathbb{A}_0$).

Specifying the loop body is not easy. We need to find a loop invariant $(\texttt{p}', \mathbb{G}_1, \mathbb{A}_1, \mathbb{G}_1)$ to attach to the code label `loop`. At first glance, $\mathbb{A}_1$ and $\mathbb{G}_1$ can be defined the same as $\mathbb{A}_3$ and $\mathbb{G}_3$, respectively. However, this does not work because our FORK rule requires $\mathbb{G} \Rightarrow \mathbb{G}_1$, which cannot be satisfied. Instead, our $\mathbb{A}_1$ and $\mathbb{G}_1$ are polymorphic over the loop index $\texttt{r}_0$, which reflects the composition of the changing thread environments. At the point that a new child thread is forked but the value of $\texttt{r}_0$ has not been changed to reflect the environment change, we explicitly change the assumption and guarantee to $\mathbb{A}_2$ and $\mathbb{G}_2$. When the value of $\texttt{r}_0$ is increased, we cast $\mathbb{A}_2$ and $\mathbb{G}_2$ back to $\mathbb{A}_1$ and $\mathbb{G}_1$.

5.4.2 The Readers-Writers Problem

Our logic is general enough to specify and verify general properties of concurrent programs. In this section, we give a simple solution of the readers-writers problem and show that there is no race conditions. This example also shows how P/V operations and lock primitives can be implemented and specified in CMAP.

Figure 5.10 shows the C-like pseudo code for readers and writers. Note that this simple code just ensures that there is no race conditions. It does not ensure fairness. Verification of liveness properties is part of our future work. We assume that 100 readers and writers will be created by a main thread. The main thread and its specification will be very similar to the main program shown in the previous section, so we omit it here and just focus on the readers and writers code. The array of `rf` and `wf` are not necessary for the implementation. They are introduced as auxiliary variables just for specification and verification purpose.

```
Variables :
  int[100] rf, wf;
  int cnt, writ, l, v;
Initially :
  rf[i] = wf[i] = 0, 0 ≤ i < 100;
  cnt = 0 ∧ writ = 1 ∧ l = 0;


writer(int x){
  while(true){
    P(writ);

    wf[x] := 1;

    write v ...

    wf[x] := 0;

    V(writ);
  }
}
```

```
reader(int x){
  while(true){
    lock_acq(l);
      if (cnt = 0){
        P(writ);
      }
      cnt := cnt + 1;
      rf[x] := 1;
    lock_rel(l);

    read v ...

    lock_acq(l);
      rf[x] := 0;
      cnt := cnt − 1;
      if(cnt = 0){
        V(writ);
      }
    lock_rel(l);
}}
```

Figure 5.10: Readers & writers : the high-level program

```
acq(f) : −{}
        yield
        addiu $r_1, $r_0, l
        lw $r_2, 0($r_1)
        bgtz $r_2, acq
        sw $rt, 0($r_1)
        j f

rel(f) : −{}
        yield
        sw $r_0, l($r_0)
        j f
```

```
p_writ(f) : −{}
        yield
        addiu $r_1, $r_0, writ
        lw $r_2, 0($r_1)
        beq $r_2, $r_0, p_writ
        sw $r_0, 0($r_1)
        j f

v_writ(f) : −{}
        addiu $r_1, $r_0, 1
        addiu $r_2, $r_0, writ
        sw $r_1, 0($r_2)
        j f
```

Figure 5.11: Lock & semaphore primitives

```
reader :  $-\{(\mathrm{p}_1, \hat{\mathrm{g}}, \mathbb{A}_\mathrm{r}, \mathbb{G}_\mathrm{r})\}$
          JD acq(cont_1)

cont_1 :  $-\{(\mathrm{p}_2, \hat{\mathrm{g}}, \mathbb{A}_\mathrm{r}, \mathbb{G}_\mathrm{r})\}$
          yield
          lw $\$\mathrm{r}_2$, cnt($\$\mathrm{r}_0$)
          yield
          beq $\$\mathrm{r}_2$, $\$\mathrm{r}_0$, getw
          j inc_cnt

getw :    $-\{(\mathrm{p}_3, \hat{\mathrm{g}}, \mathbb{A}_\mathrm{r}, \mathbb{G}_\mathrm{r})\}$
          JD p_writ(inc_cnt)

inc_cnt : $-\{(\mathrm{p}_4, \hat{\mathrm{g}}, \mathbb{A}_\mathrm{r}, \mathbb{G}_\mathrm{r})\}$
          yield
          lw $\$\mathrm{r}_2$, cnt($\$\mathrm{r}_0$)
          yield
          addiu $\$\mathrm{r}_3$, $\$\mathrm{r}_0$, 1
          addu $\$\mathrm{r}_2$, $\$\mathrm{r}_2$, $\$\mathrm{r}_3$
          sw $\$\mathrm{r}_2$, cnt($\$\mathrm{r}_0$)
          yield
          addiu $\$\mathrm{r}_1$, $\$\mathrm{r}_0$, 1
          sw $\$\mathrm{r}_1$, rf($ra)
          $-\{(\mathrm{p}_5, \hat{\mathrm{g}}, \mathbb{A}_r, \mathbb{G}_r)\}$
          JD rel(cont_2)

cont_2 :  $-\{(\mathrm{p}_6, \hat{\mathrm{g}}, \mathbb{A}_\mathrm{r}, \mathbb{G}_\mathrm{r})\}$
          yield
          ...
          yield
          $-\{(\mathrm{p}_6, \hat{\mathrm{g}}, \mathbb{A}_r, \mathbb{G}_r)\}$
          JD acq(cont_3)

cont_3 :  $-\{(\mathrm{p}_7, \hat{\mathrm{g}}, \mathbb{A}_\mathrm{r}, \mathbb{G}_\mathrm{r})\}$
          yield
          sw $\$\mathrm{r}_0$, rf($ra)
          yield
          lw $\$\mathrm{r}_2$, cnt($\$\mathrm{r}_0$)
          yield
          addiu $\$\mathrm{r}_3$, $\$\mathrm{r}_0$, 1
          subu $\$\mathrm{r}_2$, $\$\mathrm{r}_2$, $\$\mathrm{r}_3$
          sw $\$\mathrm{r}_2$, cnt($\$\mathrm{r}_0$)
          yield
          beq $\$\mathrm{r}_2$, $\$\mathrm{r}_0$, relw
          $-\{(\mathrm{p}_8, \hat{\mathrm{g}}, \mathbb{A}_r, \mathbb{G}_r)\}$
          JD rel(reader)

relw :    $-\{(\mathrm{p}_9, \hat{\mathrm{g}}, \mathbb{A}_\mathrm{r}, \mathbb{G}_\mathrm{r})\}$
          yield
          JD v_writ(cont_4)
cont_4 :  $-\{(\mathrm{p}_8, \hat{\mathrm{g}}, \mathbb{A}_\mathrm{r}, \mathbb{G}_\mathrm{r})\}$
          JD rel(reader)


writer :  $-\{(\mathrm{p}_{10}, \hat{\mathrm{g}}, \mathbb{A}_\mathrm{w}, \mathbb{G}_\mathrm{w})\}$
          JD p_writ(cont_5)

cont_5 :  $-\{(\mathrm{p}_{11}, \hat{\mathrm{g}}_1, \mathbb{A}_\mathrm{w}, \mathbb{G}_\mathrm{w})\}$
          addiu $\$\mathrm{r}_1$, $\$\mathrm{r}_0$, 1
          sw $\$\mathrm{r}_1$, wf($ra)
          yield
          ...
          yield
          sw $\$\mathrm{r}_0$, wf($ra)
          $-\{(\mathrm{p}_{11}, \hat{\mathrm{g}}_2, \mathbb{A}_\mathrm{w}, \mathbb{G}_\mathrm{w})\}$
          JD v_writ(writer)
```

Figure 5.12: Readers & writers : the CMAP program

$inv_1 \triangleq ([\mathtt{l}] = 0) \rightarrow (\sum_i [\mathtt{rf}+i]) = [\mathtt{cnt}] \qquad inv_2 \triangleq (\sum_i [\mathtt{wf}+i]) \leq 1$
$inv_3 \triangleq ((\sum_i [\mathtt{wf}+i]) = 1) \rightarrow ([\mathtt{cnt}]=0 \wedge [\mathtt{writ}]=0)$
$inv_4 \triangleq ([\mathtt{writ}] = 1) \rightarrow ([\mathtt{cnt}] = 0) \qquad \mathit{Inv} \triangleq inv_1 \wedge inv_2 \wedge inv_3 \wedge inv_4$

$idr \triangleq \forall i.\ [\mathtt{rf}+i] = [\mathtt{rf}+i]'$
$idr_1(i) \triangleq \forall j.\ (i \neq j) \rightarrow ([\mathtt{rf}+j] = [\mathtt{rf}+j]') \qquad idr_2(i) \triangleq [\mathtt{rf}+i] = [\mathtt{rf}+i]'$
$idw \triangleq \forall i.\ [\mathtt{wf}+i] = [\mathtt{wf}+i]'$
$idw_1(i) \triangleq \forall j.\ (i \neq j) \rightarrow ([\mathtt{wf}+j] = [\mathtt{wf}+j]') \qquad idw_2(i) \triangleq [\mathtt{wf}+i] = [\mathtt{wf}+i]'$

$\mathbb{A}_r \triangleq idr_2([\mathsf{\$ra}]) \wedge (([\mathtt{rf}+\mathsf{\$ra}] = 1) \rightarrow [\mathtt{v}] = [\mathtt{v}]') \wedge ([\mathtt{l}] = \mathsf{\$rt} \rightarrow [\mathtt{cnt}] = [\mathtt{cnt}]')$
$\mathbb{G}_r \triangleq idw \wedge idr_1(\mathsf{\$ra}) \wedge ([\mathtt{v}] = [\mathtt{v}]') \wedge ([\mathtt{l}] \notin \{\mathsf{\$rt}, 0\} \rightarrow [\mathtt{cnt}] = [\mathtt{cnt}]')$

$\mathbb{A}_w \triangleq idw_2(\mathsf{\$ra}) \wedge (([\mathtt{wf}+\mathsf{\$ra}] = 1) \rightarrow [\mathtt{v}] = [\mathtt{v}]')$
$\mathbb{G}_w \triangleq idr \wedge idw_1(\mathsf{\$ra}) \wedge (([\mathtt{writ}]=0 \wedge [\mathtt{wf}+\mathsf{\$ra}]=0) \rightarrow [\mathtt{v}] = [\mathtt{v}]')$

$\hat{\mathsf{g}} \triangleq \lambda \mathbb{R}, \mathbb{H}, \mathbb{H}'.\ \mathbb{H} = \mathbb{H}'$
$\hat{\mathsf{g}}_1 \triangleq \lambda \mathbb{R}, \mathbb{H}, \mathbb{H}'.\ \mathbb{H}\{\mathtt{wf}+\mathsf{\$ra} \rightsquigarrow 1\} = \mathbb{H}'$
$\hat{\mathsf{g}}_2 \triangleq \lambda \mathbb{R}, \mathbb{H}, \mathbb{H}'.\ \mathbb{H}\{\mathtt{writ} \rightsquigarrow 1\} = \mathbb{H}'$
$\mathsf{p}_1 \triangleq ([\mathtt{rf}+\mathsf{\$ra}] = 0) \wedge ([\mathtt{l}] \neq \mathsf{\$rt}) \wedge \mathit{Inv}$
$\mathsf{p}_2 \triangleq ([\mathtt{rf}+\mathsf{\$ra}] = 0) \wedge ([\mathtt{l}] = \mathsf{\$rt}) \wedge \mathit{Inv}$
$\mathsf{p}_3 \triangleq \mathsf{p}_2 \wedge ([\mathtt{cnt}] = 0) \wedge \mathit{Inv}$
$\mathsf{p}_4 \triangleq \mathsf{p}_2 \wedge ([\mathtt{writ}] = 0) \wedge \mathit{Inv}$
$\mathsf{p}_5 \triangleq ([\mathtt{rf}+\mathsf{\$ra}] = 1) \wedge ([\mathtt{l}] = \mathsf{\$rt}) \wedge \mathit{Inv}$
$\mathsf{p}_6 \triangleq ([\mathtt{rf}+\mathsf{\$ra}] = 1) \wedge ([\mathtt{l}] \neq \mathsf{\$rt}) \wedge ([\mathtt{cnt}] > 0) \wedge \mathit{Inv}$
$\mathsf{p}_7 \triangleq ([\mathtt{rf}+\mathsf{\$ra}] = 1) \wedge ([\mathtt{l}] = \mathsf{\$rt}) \wedge ([\mathtt{cnt}] > 0) \wedge \mathit{Inv}$
$\mathsf{p}_8 \triangleq ([\mathtt{l}] = \mathsf{\$rt}) \wedge ([\mathtt{rf}+\mathsf{\$ra}] = 0) \wedge \mathit{Inv}$
$\mathsf{p}_9 \triangleq ([\mathtt{l}] = \mathsf{\$rt}) \wedge ([\mathtt{rf}+\mathsf{\$ra}] = 0) \wedge ([\mathtt{writ}] = 0) \wedge ([\mathtt{cnt}] = 0) \wedge \mathit{Inv}$
$\mathsf{p}_{10} \triangleq ([\mathtt{wf}+\mathsf{\$ra}] = 0) \wedge \mathit{Inv}$
$\mathsf{p}_{11} \triangleq ([\mathtt{wf}+\mathsf{\$ra}] = 0) \wedge ([\mathtt{writ}] = 0) \wedge ([\mathtt{cnt}] = 0) \wedge \mathit{Inv}$

Figure 5.13: Program specifications

Figure 5.12 shows the CMAP implementation of the high-level pseudo code. Yielding is inserted at all the intervals of the atomic operations of the high-level program. The lock primitives and P/V operations, as shown in Figure 5.11, are implemented as program macros parameterized by the return label. They will be instantiated and inlined in the proper position of the code. We introduce a pseudo instruction JD to represent the inlining of the macro.

We define the global invariant and reader/writer's assumptions and guarantees in Figure 5.13. The code heap specifications are embedded in the code as annotations, as shown in Figure 5.12. Specifications of lock primitives and P/V operations are given at places they are inlined. Definitions of assertions and local guarantees used in code heap specifications are shown in Figure 5.13.

The assertion inv_1 says that out of the critical section protected by the lock, the value of the counter `cnt` is always consistent and reflects the number of the readers that can read the value of `v`; inv_2 says at one time there is at most one writer that can change the value of `v`; while inv_3 and inv_4 states the relationship between the counter `cnt`, the semaphore variable `writ` and the actual number of writers that can write the data, which must be maintained to ensure the mutual exclusive access between readers and writers. The program invariant *Inv*, which is the conjunction of the four, must be satisfied at any step of the execution.

The assumptions and guarantees of readers and writers are parameterized by their thread id (in rt) and thread argument (in ra). The reader assumes ($\mathbb{A}_r$) that if it has the right to read the data `v` (*i.e.*, $\mathtt{rf}[\mathsf{ra}] = 1$), nobody can change the data. Also, if it owns the lock, nobody else can change the value of the counter. Finally, it assumes its auxiliary variable $\mathtt{rf}[\mathsf{ra}]$ will never be changed by others. Readers guarantee ($\mathbb{G}_r$) that they will never change the shared data `v` and auxiliary variables of other threads, and it will not revise the counter unless it owns the lock. Writers' assumption ($\mathbb{A}_w$) and guarantee ($\mathbb{G}_w$) are defined in a similar way to ensure the non-interference.

```
Variables:
  nat m, n;
  nat[2] flag;
                                    void gcd(int x){
Main :                                while (m ≠ n){
  m := α;   n := β;                     if ((m > n)&&(x = 0))
  flag[0] := 0;                           m := m − n;
  fork(gcd, 0);                         else if ((m < n)&&(x = 1))
  flag[1] := 0;                           n := n − m;
  fork(gcd, 1);                       }
  while (!flag[0])                    flag[x] := 1;
    yield;                          }
  while (!flag[1])
    yield;
  post proc . . .
```

Figure 5.14: GCD: high-level program

5.4.3 Partial Correctness of A Lock-Free Program

Figure 5.15 and 5.16 give the CMAP implementation of the Euclidean algorithm to compute the greatest common divisor (GCD) of α and β, stored at locations m and n initially. This is a lock-free algorithm, *i.e.,* no synchronization is required to ensure the non-interference even if atomic operations are machine instructions. To show the lock-free property, we insert yield instructions at every program point of the chld thread (some yield instructions are omitted in the main thread for clarity).

In Figure 5.17 we show definitions of assertions used to specify the program. The following formulae show the (partial) specifications of the code heap, and the initial memory and instruction sequence.

$$
\begin{aligned}
&\psi \triangleq \{(\texttt{main},\ (\mathsf{True}, \mathbb{G}_0, \mathbb{A}_0, \mathbb{G}_0)), (\texttt{chld},\ (\mathtt{p}_2 \wedge \mathtt{p}_8, \mathbb{G}_g, \mathbb{A}_g, \mathbb{G}_g)), \dots\}\\
&\text{Initial } \mathbb{H} \triangleq \{\mathtt{m} \rightsquigarrow \_, \mathtt{n} \rightsquigarrow \_, \mathtt{flag} \rightsquigarrow \_, (\mathtt{flag}+1) \rightsquigarrow \_\}\\
&\text{Initial } \mathsf{pc} \triangleq \texttt{main}
\end{aligned}
$$

The partial correctness of the gcd algorithm is inferred by the fact that $\mathtt{p}_2$ is established at the entry point and $\mathbb{G}_g$ is satisfied.

```
main : $-\{(\mathsf{True}, \mathbb{G}_0, \mathbb{A}_0, \mathbb{G}_0)\}$
       j begn
begn : $-\{(\mathsf{True}, \mathbb{G}_0, \mathbb{A}_0, \mathbb{G}_0)\}$
       addiu $\$r_1, \$r_0, \alpha$
       sw $\$r_1, m(\$r_0)$
       yield
       addiu $\$r_1, \$r_0, \beta$
       sw $\$r_1, n(\$r_0)$
       yield
       addiu $\$r_1, \$r_0, \mathtt{flag}$
       sw $\$r_0, 0(\$r_1)$
       yield
       fork chld, $\$r_0$

next :  $-\{p_0 \wedge p_2, \hat{g}, \mathbb{A}_1, \mathbb{G}_1\}$
        yield
        sw $\$r_0, 1(\$r_1)$
        yield
        addiu $\$r_2, \$r_0, 1$
        fork chld, $\$r_2$
join1 : $-\{p_0 \wedge p_2 \wedge p_3, \hat{g}, \mathbb{A}_2, \mathbb{G}_2\}$
        yield
        lw $\$r_2, 0(\$r_1)$
        beq $\$r_0, \$r_2, \mathtt{join}_1$
join2 : $-\{p_0 \wedge p_2 \wedge p_4, \hat{g}, \mathbb{A}_2, \mathbb{G}_2\}$
        yield
        lw $\$r_2, 1(\$r_1)$
        beq $\$r_0, \$r_2, \mathtt{join}_2$
post :  $-\{(p_2 \wedge p_5, \hat{g}, \mathbb{A}_2, \mathbb{G}_2)\}$
        yield
        ...
```

Figure 5.15: GCD: the main thread

```
chld : $-\{(p_2 \wedge p_8, \mathbb{G}_g, \mathbb{A}_g, \mathbb{G}_g)\}$
       j gcd
gcd :  $-\{p_2 \wedge p_8\}$
       yield
       lw $\$r_1, m(\$r_0)$
       yield
       $-\{p_2 \wedge p_{10}\}$
       lw $\$r_2, n(\$r_0)$
       yield
       $-\{p_2 \wedge p_{11}\}$
       beq $\$r_1, \$r_2$, done
       yield
       subu $\$r_3, \$r_1, \$r_2$
       yield
       bgtz $\$r_3, \mathtt{calc}_1$
       yield
       j $\mathtt{calc}_2$
done : $-\{p_1 \wedge p_2 \wedge p_8\}$
       yield
       addiu $\$r_1, \$r_0, 1$
       yield
       sw $\$r_1, \mathtt{flag}(\$ra)$
       exit

calc1 : $-\{p_2 \wedge p_{12}\}$
        yield
        beq $\$r_0, \$ra, \mathtt{cld}_1$
        yield
        j loop
cld1 :  $-\{p_2 \wedge p_{13}\}$
        yield
        sw $\$r_3, m(\$r_0)$
        yield
        j loop
calc2 : $-\{p_2 \wedge p_{14}\}$
        yield
        addiu $\$r_3, \$r_0, 1$
        yield
        beq $\$r_3, \$ra, \mathtt{cld}_2$
        yield
        j loop
cld2 :  $-\{p_2 \wedge p_{15}\}$
        yield
        subu $\$r_3, \$r_2, \$r_1$
        yield
        sw $\$r_3, n(\$r_0)$
        yield
        j loop
```

Figure 5.16: GCD: the chld thread

$$
\begin{array}{lll}
\texttt{id}_1 & \triangleq & [\texttt{m}] = [\texttt{m}]' \qquad \texttt{id}_2 \triangleq [\texttt{n}] = [\texttt{n}]' \\
\texttt{id}_3 & \triangleq & ([\texttt{flag}{+}0] = [\texttt{flag}{+}0]') \wedge ([\texttt{flag}{+}1] = [\texttt{flag}{+}1]') \\
\\
\mathbb{G}_0 & \triangleq & \mathsf{True} \qquad \mathbb{A}_0 \triangleq \texttt{id}_1 \wedge \texttt{id}_2 \wedge \texttt{id}_3 \\
\\
\texttt{p} & \triangleq & \texttt{id}_2 \wedge ([\texttt{m}] > [\texttt{n}] \to (\texttt{GCD}([\texttt{m}],[\texttt{n}]) = \texttt{GCD}([\texttt{m}]',[\texttt{n}]'))) \wedge ([\texttt{m}] \le [\texttt{n}] \to \texttt{id}_1) \\
\texttt{p}' & \triangleq & \texttt{id}_1 \wedge ([\texttt{m}] < [\texttt{n}] \to (\texttt{GCD}([\texttt{m}],[\texttt{n}]) = \texttt{GCD}([\texttt{m}]',[\texttt{n}]'))) \wedge ([\texttt{m}] \ge [\texttt{n}] \to \texttt{id}_2) \\
\\
\mathbb{G}_g & \triangleq & ([\texttt{flag}{+}\$\mathsf{ra}]' = 0 \to ((\$\mathsf{ra} = 0 \wedge \texttt{p}) \vee (\$\mathsf{ra} = 1 \wedge \texttt{p}'))) \\
 & & \wedge([\texttt{flag}{+}\$\mathsf{ra}]' = 1 \to (\texttt{id}_1 \wedge \texttt{id}_2 \wedge [\texttt{m}]' = [\texttt{n}]')) \\
 & & \wedge([\texttt{flag}{+}\$\mathsf{ra}] = 1 \to \mathbb{H} = \mathbb{H}') \\
 & & \wedge([\texttt{flag}{+}(1-\$\mathsf{ra})] = [\texttt{flag}{+}(1-\$\mathsf{ra})]') \\
\mathbb{A}_g & \triangleq & ([\texttt{flag}{+}\$\mathsf{ra}]' = 0) \to (([\texttt{flag}{+}\$\mathsf{ra}] = [\texttt{flag}{+}\$\mathsf{ra}]') \\
 & & \qquad\qquad \wedge((\$\mathsf{ra} = 0 \wedge \texttt{p}') \vee (\$\mathsf{ra} = 1 \wedge \texttt{p}))) \\
\\
\mathbb{G}_1 & \triangleq & \mathbb{G}_0 \wedge ([\texttt{flag}{+}0]' = 0 \to (([\texttt{flag}{+}0] = [\texttt{flag}{+}0]') \wedge \texttt{p}')) \\
\mathbb{A}_1 & \triangleq & \mathbb{A}_0 \vee (([\texttt{flag}{+}1] = [\texttt{flag}{+}1]') \wedge ([\texttt{flag}{+}0]' = 0 \to \texttt{p}) \\
 & & \qquad \wedge([\texttt{flag}{+}0]' = 1 \to (\texttt{id}_1 \wedge \texttt{id}_2 \wedge [\texttt{m}]' = [\texttt{n}]'))) \\
 & & \qquad \wedge([\texttt{flag}{+}0] = 1 \to \mathbb{H} = \mathbb{H}')) \\
\\
\mathbb{G}_2 & \triangleq & \mathbb{G}_1 \wedge ([\texttt{flag}{+}1]' = 0 \to (([\texttt{flag}{+}1] = [\texttt{flag}{+}1]') \wedge \texttt{p})) \\
\mathbb{A}_2 & \triangleq & \mathbb{A}_1 \vee (([\texttt{flag}{+}0] = [\texttt{flag}{+}0]') \wedge ([\texttt{flag}{+}1]' = 0 \to \texttt{p}') \\
 & & \qquad \wedge([\texttt{flag}{+}1]' = 1 \to (\texttt{id}_1 \wedge \texttt{id}_2 \wedge [\texttt{m}]' = [\texttt{n}]'))) \\
 & & \qquad \wedge([\texttt{flag}{+}1] = 1 \to \mathbb{H} = \mathbb{H}')) \\
\\
\hat{\texttt{g}} & \triangleq & \mathbb{H} = \mathbb{H}' \qquad \texttt{p}_0 \triangleq \$\texttt{r}_1 = \texttt{flag} \qquad \texttt{p}_1 \triangleq [\texttt{m}] = [\texttt{n}] \\
\texttt{p}_2 & \triangleq & \texttt{GCD}([\texttt{m}],[\texttt{n}]) = \texttt{GCD}(\alpha,\beta) \qquad \texttt{p}_3 \triangleq [\texttt{flag}{+}0] = 0 \vee ([\texttt{flag}{+}0] = 1 \wedge \texttt{p}_1) \\
\texttt{p}_4 & \triangleq & ([\texttt{flag}{+}1] = 0 \vee [\texttt{flag}{+}1] = 1) \wedge [\texttt{flag}{+}0] = 1 \wedge \texttt{p}_1 \\
\texttt{p}_5 & \triangleq & ([\texttt{flag}{+}0] = 1) \wedge ([\texttt{flag}{+}1] = 1) \wedge \texttt{p}_1 \\
\\
\texttt{p}_6 & \triangleq & (\$\mathsf{ra} = 0) \wedge ([\texttt{flag}{+}0] = 0) \qquad \texttt{p}_7 \triangleq (\$\mathsf{ra} = 1) \wedge ([\texttt{flag}{+}1] = 0) \\
\texttt{p}_8 & \triangleq & \texttt{p}_6 \vee \texttt{p}_7 \\
\texttt{p}_{10} & \triangleq & (\texttt{p}_6 \wedge \$\texttt{r}_1 = [\texttt{m}]) \vee (\texttt{p}_7 \wedge (\$\texttt{r}_1 \le [\texttt{n}] \to \$\texttt{r}_1 = [\texttt{m}])) \\
\texttt{p}_{11} & \triangleq & (\texttt{p}_6 \wedge \$\texttt{r}_1 = [\texttt{m}] \wedge (\$\texttt{r}_1 \ge \$\texttt{r}_2 \to \$\texttt{r}_2 = [\texttt{n}])) \\
 & & \vee(\texttt{p}_7 \wedge \$\texttt{r}_2 = [\texttt{n}] \wedge (\$\texttt{r}_1 \le \$\texttt{r}_2 \to \$\texttt{r}_1 = [\texttt{m}])) \\
\\
\texttt{p}_{12} & \triangleq & (\$\texttt{r}_3 = \$\texttt{r}_1 - \$\texttt{r}_2) \wedge (\$\texttt{r}_3 > 0) \\
 & & \wedge(\$\mathsf{ra} = 0 \to ([\texttt{flag}{+}0] = 0) \wedge (\$\texttt{r}_1 = [\texttt{m}]) \wedge (\$\texttt{r}_2 = [\texttt{n}])) \\
\texttt{p}_{13} & \triangleq & \texttt{p}_6 \wedge \texttt{p}_{12} \\
\texttt{p}_{14} & \triangleq & (\$\texttt{r}_1 < \$\texttt{r}_2) \wedge (\$\mathsf{ra} = 1 \to ([\texttt{flag}{+}1] = 0) \wedge (\$\texttt{r}_1 = [\texttt{m}]) \wedge (\$\texttt{r}_2 = [\texttt{n}])) \\
\texttt{p}_{15} & \triangleq & \texttt{p}_7 \wedge \texttt{p}_{14}
\end{array}
$$

Figure 5.17: GCD: Assertion Definitions

Two child threads are created using the same static thread `chld`. They use thread arguments to distinguish their task. Correspondingly, the assumption $\mathbb{A}_g$ and $\mathbb{G}_g$ of the static thread `chld` also use the thread argument to distinguish the specification of different dynamic copies. The child thread does not change its assumption and guarantee throughout its lifetime. Therefore we omit $\mathbb{A}_g$ and $\mathbb{G}_g$ in the code heap specifications. Since we insert yield instructions at every program point, every local guarantee is simply $\mathbb{G}_g$, which is also omitted in the specifications.

At the data initiation stage, the main thread assumes no threads in the environment changes $[\mathtt{m}]$, $[\mathtt{n}]$ and the flags, and guarantees nothing, as shown in $\mathbb{A}_0$ and $\mathbb{G}_0$ (see Figure 5.17). After creating child threads, the main thread changes its assumption and guarantee to reflect the changing environment. The new assumption is just the disjunction of the previous assumption and the guarantee of the new thread (instantiated by the thread id and thread argument), similarly the new guarantee is the conjunction of the previous guarantee and the new thread's assumption.

This example also shows how thread join can be implemented in CMAP by synchronization. We use one flag for each thread to indicate if the thread is alive or not. The assumptions and guarantees of the main thread also take advantages of these flags so that it can weaken its guarantee and strengthen its assumption after the child threads die.

5.5 Discussions and Summary

CMAP is proposed to certify multi-threaded assembly code with unbounded dynamic thread creations. The work is related to two directions of research: concurrency verification and PCC. The rely-guarantee method [59, 90, 93, 1, 38, 102] is one of the best studied technique for compositional concurrent program verification. However, most of the work on it are based on high-level languages or calculi, and none of them support unbounded dynamic thread creation. On the other hand, many PCC frameworks [72, 69, 6, 46, 101, 27, 102] have been proposed for machine/assembly code verification, but most of them only support sequential code. The only intersection point of these two directions is the work on CCAP [102], which applies the R-G method at the assembly level to verify general concurrency properties, like mutual exclusion and deadlock-freedom. Unfortunately, CCAP does not support dynamic thread creation either.

Recently, O'Hearn and others [78, 14, 13] applied separation logic to reason about concurrent programs. Bornat et al. [13] showed how to verify the race-free property of a program solving the readers and writers problem, which is similar to the program presented in this paper. However, their work heavily depends on the higher-level language features, such as resources and conditional critical regions. As other traditional work on concurrency verification, they only support nested cobegin/coend structure. It is not clear how their technique can be applied to support assembly code verification with unbounded dynamic thread creation. The relationship between rely-guarantee reasoning and concurrent separation logic will be studied in Chapter 7.

A number of model checkers have been developed for concurrent software verification, but most of them only check programs with a fixed finite number of threads [53, 29, 48, 18, 41, 39]. The CIRC algorithm [49] supports unbounded number of threads, but it does not model the dynamic thread creation and the changing thread environment. Qadeer and Rehof [84] pointed out that verification of concurrent Boolean program with unbounded parallelism is decidable if the number of context switches is bounded, but they do not directly verify dynamic thread creation either. Given a context bound k, they reduce a dynamic concurrent program P to a program Q with k+1 threads and verify Q instead. 3VMC [100] supports both unbounded number of threads and dynamic thread creation, but it is not based on the rely-guarantee method and does not support compositional verification.

Many type systems are also proposed to reason about concurrent programs [36, 37, 40, 42]. Unlike CMAP, which uses high-order predicate logic to verify general program properties, they are designed to automatically reason about specific properties of programs, like races, deadlocks and atomicity. Also, they do not directly generate proofs about program properties. Instead, proofs are implicitly embedded in their soundness proofs.

CMAP extends previous work on R-G method and CCAP with dynamic thread creation. We unify the concepts of a thread's assumption/guarantee and its environment's guarantee/assumption, and allow a thread to change its assumption and guarantee to track the changing environment caused by dynamic thread creation. Code segments in CMAP can be specified and verified once and used in multiple threads with different assumptions and guarantees, therefore CMAP achieves better modularity than CCAP. Some practical issues, such as argument passing at thread creation, thread local data, and multiple invocation of one copy of thread code, are also discussed to support practical multithreaded programming. CMAP has been developed using the Coq proof assistant, along with a formal soundness proof and the verified example programs.

CMAP is designed as a general program logic for assembly level multi-threaded programs. Although directly specifying and proving CMAP programs may be daunting, it is simpler at the higher level, and it is possible to compile code and specifications at higher level down to CMAP code. Also, there are common concurrent idioms that would permit the assumption/guarantee to be generated automatically during compilation. For example, for critical sections protected by lock, the assumption is always like "nobody else will update the protected data if I am holding the lock..." (see Figure 5.13). Another scenario is illustrated by the example shown in Figure 5.3 and 5.9, where each thread is exclusively responsible for a single piece of global data. In these scenarios, the assumption is always like "nobody else will touch my share of data" and the guarantee is like "I will not update other threads' data". Automatically generating assumptions/guarantees and proofs for common idioms, and compiling higher level concurrent programs and specifications down to CMAP, will be our future work.

Chapter 6

Applications of OCAP

We have seen the embedding of TAL in OCAP in Chapter 3, and SCAP in OCAP in Chapter 4. In this chapter, we show OCAP's support of interoperability of different verification systems, which can be applied to link certified program modules at different abstraction levels. Applications presented here are derived from the joint work I have done with Zhaozhong Ni, Zhong Shao and Yu Guo [33].

6.1 Linking TAL with Certified Runtime

As discussed in Capter 3, TAL [69] treats memory allocation as an abstract primitive instruction that is not available in any real Instruction Set Architectures (ISAs). We cannot use TAL to certify implementations of memory allocation library because TAL enforces the invariant that types of values stored in data heap cannot change during program execution. Although the invariant of type-preserving memory update frees programmers from worrying about memory aliasing, it is too high-level to type-check the code implementing memory allocations, which does type-changing memory update. In order to solve this problem, we can augment TAL's type system with linearity and explicit control of memory aliasing [98], but this would make the type system more complicated and lose the abstraction in the original TAL. We take a different approach here: we still use

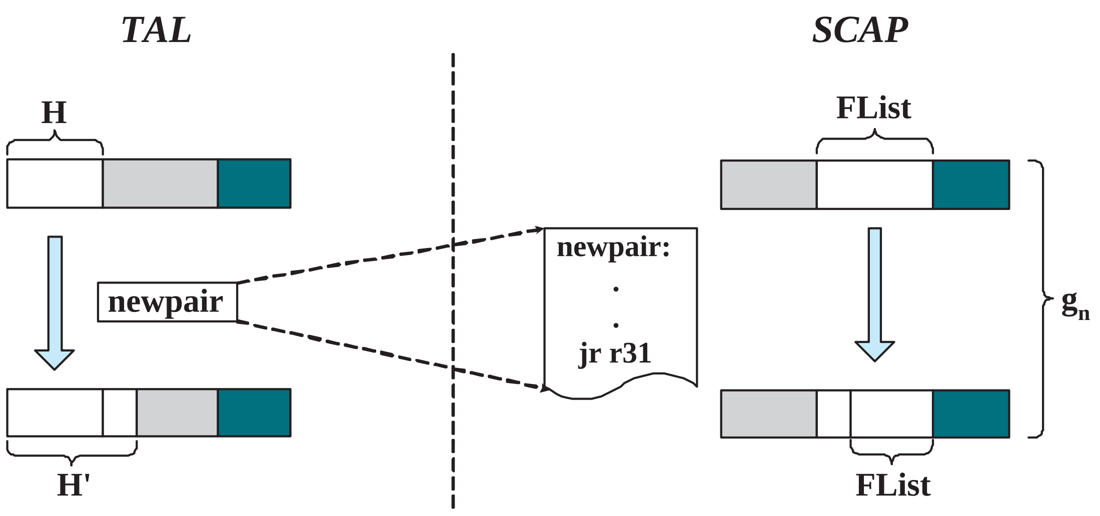


Figure 6.1: Interoperation with TAL and SCAP

TAL to certify code that does type-preserving memory update and enjoy the abstraction provided by TAL, but use SCAP to certify implementations of memory allocations. In TAL, we change the primitive instruction for memory allocation to a function call to the implementation, so we directly handle real machine code with no special instructions.

In this section, we use a simple function `newpair` to do memory allocation. The code for `newpair` is specified and certified in SCAP without knowing about the future interoperation with TAL. The client code is certified in TAL. There is also a TAL interface for `newpair` so that the call to `newpair` can be type-checked. Given our embedding of TAL and SCAP in OCAP and the corresponding interpretations, we can prove that the TAL interface for `newpair` is compatible with the SCAP interface. Linking of `newpair` with its client code is done in OCAP.

The tricky part is that TAL and SCAP have different views about machine states. As shown in Figure 6.1, TAL (the left side) only knows the heap reachable from the client code. It believes that `newpair` will magically generate a memory block of two-word size. The free list of memory blocks (FList) and other parts of the system resource is invisible to TAL code and types. SCAP (on the right side) only cares about operations over the free list. It does not know what the heap for TAL is. But when it returns, it has to ensure that the invariant in TAL is not violated. As we will show in this section, the way we use specification interpretations and our SCAP have nice support of memory polymorphism. They help us achieve similar effect of the frame rule in separation logic [80].

6.1.1 Certifying `newpair` in SCAP.

The `newpair` function takes no argument and returns a memory block of two-word size. The reference to the memory block is saved in the register $\mathtt{r}_{30}$ as the return value. The callee-save registers are $\mathtt{r}_1, \ldots, \mathtt{r}_9$. The following code schema shows the implementation $\mathbb{C}_{\text{SCAP}}$ of `newpair`, which largely follows the `malloc` function certified by Yu *et al.* [101]. Since this chapter is focused on interoperations between different systems instead of illustrating the usage of individual system, we omit the actual code and its specification details here, which would not affect the linking in OCAP.

```
newpair:
          ...
          jr  r31
```

Before we specify the `newpair` function in SCAP, we first define separation logic connectors in our meta-logic:

$$\begin{array}{lcl}
\mathtt{l} \mapsto i & \triangleq & \lambda\mathbb{S}.\ dom(\mathbb{S}.\mathbb{H}) = \{\mathtt{l}\} \ \wedge\ \mathbb{S}.\mathbb{H}(\mathtt{l}) = i \\
\mathtt{l} \mapsto \_ & \triangleq & \lambda\mathbb{S}.\ \exists n.\ (\mathtt{l} \mapsto n)\ \mathbb{S} \\
\mathsf{p}_1 * \mathsf{p}_2 & \triangleq & \lambda(\mathbb{H},\mathbb{R}).\ \exists\mathbb{H}',\mathbb{H}''.\ \mathbb{H} = \mathbb{H}' \uplus \mathbb{H}'' \ \wedge \\
 & & \qquad\qquad \mathsf{p}_1\ (\mathbb{H}',\mathbb{R}) \ \wedge\ \mathsf{p}_2\ (\mathbb{H}'',\mathbb{R}) \\
\left\{\begin{matrix}\mathsf{p}\\ \mathsf{q}\end{matrix}\right\} * \mathsf{ID} & \triangleq & \lambda(\mathbb{H}_1,\mathbb{R}_1),(\mathbb{H}_2,\mathbb{R}_2). \\
 & & \quad \forall\mathbb{H},\mathbb{H}_1'.\ \mathbb{H}_1 = \mathbb{H}_1' \uplus \mathbb{H} \ \wedge\ \mathsf{p}\ (\mathbb{H}_1',\mathbb{R}_1) \rightarrow \\
 & & \qquad \exists\mathbb{H}_2'.\ \mathbb{H}_2 = \mathbb{H}_2' \uplus \mathbb{H} \ \wedge\ \mathsf{q}\ (\mathbb{H}_2',\mathbb{R}_2)
\end{array}$$

Following Yu *et al.*'s specification for `malloc` [101], we use an assertions FList to specify a well-formed list of free memory blocks maintained by `newpair`. The SCAP code specification for `newpair` is $(\mathrm{p}_n, \mathrm{g}_n)$ where

$$\mathrm{p}_n \triangleq \mathsf{FList}$$

$$\mathrm{g}_n \triangleq (\forall \mathtt{r} \in \{\mathtt{r}_1, \ldots, \mathtt{r}_9, \mathtt{r}_{31}\}.\, \mathtt{r} = \mathtt{r}') \wedge \left\{\begin{array}{l} \mathsf{FList} \\ \mathsf{FList} \,*\, (\mathtt{r}_{30}{}' \mapsto \_) \,*\, (\mathtt{r}_{30}{}'{+}1 \mapsto \_) \end{array}\right\} * \mathsf{ID}\,.$$

Recall that g in SCAP specifies the guarantee of functions. We use `r` to represent the value of `r` in the first state (the current state), while the primed value $\mathtt{r}'$ means the value of `r` in the second state (the return state). Here the precondition p_n requires there is a well-formed list of free memory blocks in the heap. The guarantee g_n says the function will reinstate the value of callee-save registers and the return address before it returns. Also, as shown in Figure 6.1, the original FList is split into a smaller FList and a memory block of two-word size. It is also important to see that g_n requires that the rest of the memory is unchanged. Therefore, execution of `newpair` would not break invariants established in TAL over the part of the memory accessible from client code.

The specification for the code heap $\mathbb{C}_{\mathrm{SCAP}}$ containing the `newpair` function is as follows:

$$\psi_s \;\triangleq\; \{(\mathtt{newpair},\ (\mathrm{p}_n, \mathrm{g}_n)),\ \ldots\}\,.$$

After `newpair` is specified in SCAP, we would have the SCAP derivation for the following SCAP judgment:

$$\psi_s \vdash \{(\mathrm{p}_n, \mathrm{g}_n)\}\, \mathtt{newpair} :\ \mathbb{I}_{\mathtt{newpair}} \tag{6.1}$$

where $\mathbb{I}_{\mathtt{newpair}} = \mathbb{C}_{\mathrm{SCAP}}[\mathtt{newpair}]$.

6.1.2 Certifying the caller in TAL.

The following code schema ($\mathbb{C}_{\text{TAL}}$) shows part of the code for the caller `getm`. Code following the jal instruction is labeled by `cont`, which is passed to `newpair` as the return address. When we reach the label `cont`, we know the register $\mathtt{r}_{30}$ points to a pair of fresh memory cells.

```
getm:
      jal   newpair
cont: ...                ; r30 points to a pair
```

We use the following TAL code heap specification to type check the above code $\mathbb{C}_{\text{TAL}}$. In addition to specifications for `getm` and `cont`, `newpair` is also specified here, so that the function call to it can be type checked in TAL. In ψ_t, each code label is associated with one type, so we present it in the form of a partial mapping (which is a special form of binary relations).

$$\begin{aligned}\psi_t \triangleq \{\, \mathtt{newpair} &\leadsto [\alpha_1,\ldots,\alpha_9].\,\{\mathtt{r}_1\leadsto\alpha_1,\ldots,\mathtt{r}_9\leadsto\alpha_9,\\ &\qquad \mathtt{r}_{31}\leadsto\forall[].\,\{\mathtt{r}_1\leadsto\alpha_1,\ldots,\mathtt{r}_9\leadsto\alpha_9,\mathtt{r}_{30}\leadsto\langle\tau^0,\tau'^0\rangle\}\},\\ \mathtt{getm} &\leadsto [\ldots].\,\{\mathtt{r}_1\leadsto\tau_1,\ldots,\mathtt{r}_9\leadsto\tau_9,\ldots\},\\ \mathtt{cont} &\leadsto [\ldots].\,\{\mathtt{r}_1\leadsto\tau_1,\ldots,\mathtt{r}_9\leadsto\tau_9,\mathtt{r}_{30}\leadsto\langle\tau^0,\tau'^0\rangle\}.\end{aligned}$$

Recall that TAL types are defined in Figure 3.5. TAL uses register file types as preconditions for instruction sequences. From TAL's point of view (see Fig. 6.1), `newpair` takes no argument and returns a reference in $\mathtt{r}_{30}$ pointing to two fresh memory cells with types τ and τ'. The tag 0 means they are uninitialized memory cells (see Section 3.5). Also values of callee-save registers have to be preserved, which is enforced by the polymorphic type.

After $\mathbb{C}_{\text{TAL}}$ is certified in TAL, we get derivations for the following TAL judgments.

$$\psi_t \vdash \{\psi_t(\mathtt{getm})\}\,\mathtt{getm} : \mathbb{I}_{\mathtt{getm}} \tag{6.2}$$

$$\psi_t \vdash \{\psi_t(\mathtt{cont})\}\,\mathtt{cont} : \mathbb{I}_{\mathtt{cont}} \tag{6.3}$$

where $\mathbb{I}_{\mathtt{getm}} = \mathbb{C}_{\text{TAL}}[\mathtt{getm}]$ and $\mathbb{I}_{\mathtt{cont}} = \mathbb{C}_{\text{TAL}}[\mathtt{cont}]$.

6.1.3 Linking the caller and callee

So far, we have specified and certified the caller and callee independently in TAL and SCAP. When one of them is specified, we need no knowledge about the system in which the other is specified. Our next step is to link the caller and the callee in OCAP, given our previous embedding of TAL and SCAP in OCAP. We suppose the language ID for TAL and SCAP are ρ and ρ', respectively.

Recall that the interpretation for TAL's register file types is $[\![\_]\!]^{(\rho,r)}_{\mathcal{L}_{\mathrm{TAL}}}$, as defined by formula 3.13, where r is an open parameter specifying the invariant of data heap used by TAL's runtime systems. Here we use FList to instantiate r. Then we define $\mathcal{D}_{\mathrm{TAL}}$ as:

$$\mathcal{D}_{\mathrm{TAL}} \triangleq \{\rho \rightsquigarrow \langle \mathcal{L}_{\mathrm{TAL}},\ [\![\_]\!]^{(\rho,\mathsf{FList})}_{\mathcal{L}_{\mathrm{TAL}}} \rangle\}.$$

The interpretation for SCAP specifications is $[\![\_]\!]^{(\rho',\mathcal{D})}_{\mathcal{L}_{\mathrm{SCAP}}}$ (see Section 4.6 for the definition), where $\mathcal{D}$ is an open parameter describing verification systems in which the client code is certified. So we use $\mathcal{D}_{\mathrm{TAL}}$ to instantiate $\mathcal{D}$ in this case. Then the language dictionary $\mathcal{D}_{\mathrm{FULL}}$ for both languages is:

$$\mathcal{D}_{\mathrm{FULL}} \triangleq \mathcal{D}_{\mathrm{TAL}} \cup \{\rho' \rightsquigarrow \langle \mathcal{L}_{\mathrm{SCAP}},\ [\![\_]\!]^{(\rho',\mathcal{D}_{\mathrm{TAL}})}_{\mathcal{L}_{\mathrm{SCAP}}} \rangle\}\,.$$

Merging the code of the caller and the callee, we get:

$$\mathbb{C}_{\mathrm{FULL}} \triangleq \{\mathtt{getm} \rightsquigarrow \mathbb{I}_{\mathtt{getm}}, \mathtt{cont} \rightsquigarrow \mathbb{I}_{\mathtt{cont}}, \mathtt{newpair} \rightsquigarrow \mathbb{I}_{\mathtt{np}},\ \ldots\}\,.$$

Then we lift TAL and SCAP specifications to get the OCAP specification Ψ_{FULL} for $\mathbb{C}_{\mathrm{FULL}}$ (recall that $(\mathtt{newpair},\ (\mathtt{p}_n, \mathtt{g}_n)) \in \psi_s$):

$$\begin{aligned}\Psi_{\mathrm{FULL}} \triangleq \{&(\ \mathtt{getm}\ ,\ \langle \rho\ ,\ \mathcal{L}_{\mathrm{TAL}}\ ,\ \psi_t(\mathtt{getm})\ \rangle\ ),\\ &(\ \mathtt{cont}\ ,\ \langle \rho\ ,\ \mathcal{L}_{\mathrm{TAL}}\ ,\ \psi_t(\mathtt{cont})\ \rangle\ ),\\ &(\ \mathtt{newpair},\ \langle \rho',\ \mathcal{L}_{\mathrm{SCAP}},\ (\mathtt{p}_n, \mathtt{g}_n)\ \rangle\ ),\\ &(\ \mathtt{newpair},\ \langle \rho\ ,\ \mathcal{L}_{\mathrm{TAL}}\ ,\ \psi_t(\mathtt{newpair})\ \rangle\ ),\ \ldots\ \}\,.\end{aligned}$$

To certify $\mathbb{C}_{\text{FULL}}$, we need to construct the proof for:

$$\mathcal{D}_{\text{FULL}}; \Psi_{\text{FULL}} \vdash \mathbb{C}_{\text{FULL}} : \Psi_{\text{FULL}} \tag{6.4}$$

By applying the OCAP CDHP rule, we need derivations for the well-formedness of each instruction sequence in OCAP. By the soundness of the embedding of TAL (Theorem 3.13) and the embedding of SCAP (Theorem 4.1), we can convert derivations (6.2), (6.3) and (6.1) to corresponding OCAP derivations for free. The only tricky part is to show the `newpair` code is well-formed with respect to the TAL specification, *i.e.*,

$$\begin{gathered}\mathcal{D}_{\text{FULL}} \vdash \{\langle \mathtt{a} \rangle_{\Psi_{\text{FULL}}}\}\, \mathtt{newpair} : \ \mathbb{I}_{\mathtt{newpair}} \\ \text{where } \mathtt{a} = [\![\, \langle \rho, \mathcal{L}_{\text{TAL}}, \psi_t(\mathtt{newpair}) \,\rangle \,]\!]_{\mathcal{D}_{\text{FULL}}} .\end{gathered} \tag{6.5}$$

To prove (6.5), we prove the following theorem, which says the TAL specification for `newpair` is compatible with the SCAP one under their interpretations. Then we apply the OCAP WEAKEN rule and get (6.5).

Theorem 6.1

Let $\mathtt{a} = [\![\, \langle \rho, \mathcal{L}_{\text{TAL}}, \psi_t(\mathtt{newpair}) \,\rangle \,]\!]_{\mathcal{D}_{\text{FULL}}}$, we have

$$\mathtt{a} \Rightarrow [\![\, \langle \rho', \mathcal{L}_{\text{SCAP}}, (\psi_s(\mathtt{newpair})) \rangle \,]\!]_{\mathcal{D}_{\text{FULL}}} .$$

To prove Theorem 6.1, we need to prove the following lemmas:

Lemma 6.2 (Heap Extension in TAL)

If $\psi \vdash \mathbb{H} : \Phi$, $\psi; \Phi \vdash \mathbb{R} : \Gamma$, $\mathtt{l} \notin \mathit{dom}(\Phi)$ and $\mathtt{l} \notin \psi$, then

(1) $\psi \vdash \mathbb{H}\{\mathtt{l} \leadsto \mathtt{w}\} : \Phi\{\mathtt{l} \leadsto \tau^0\}$;

(2) $\psi; \Phi\{\mathtt{l} \leadsto \tau^0\} \vdash \mathbb{R} : \Gamma$.

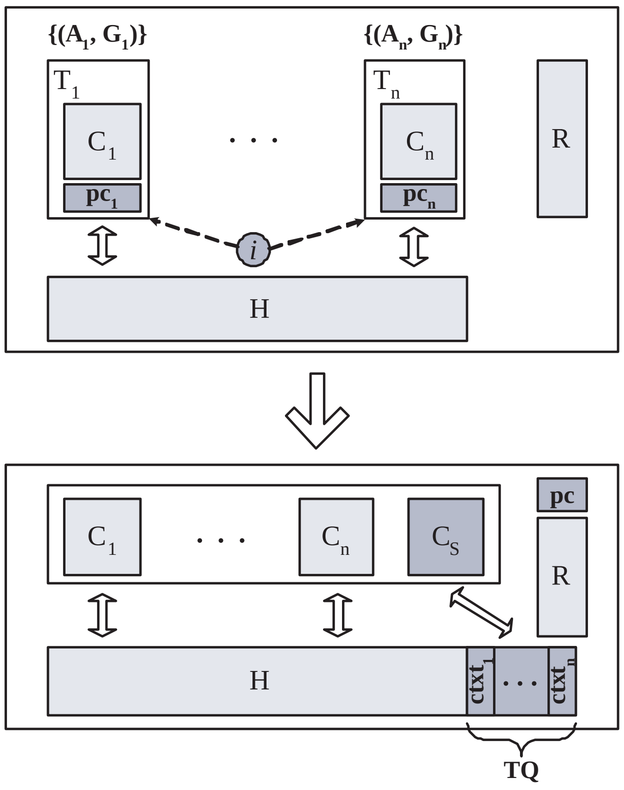


Figure 6.2: Concurrent Code at Different Abstraction Levels

Lemma 6.3 (Register File Update)

If $\psi; \Phi \vdash \mathbb{R} : \Gamma$ and $\psi; \Phi \vdash \mathtt{w} : \tau$, then $\psi; \Phi \vdash \mathbb{R}\{\mathtt{r} \rightsquigarrow \mathtt{w}\} : \Gamma\{\mathtt{r} \rightsquigarrow \tau\}$.

Proof of Lemmas 6.2 and 6.3 are very similar to their counterparts in the original TAL [70]. The whole linking process has been formalized [31] in Coq.

6.2 Certified Threads and Scheduler

As an important application of OCAP, we show how to construct a fully certified package for concurrent code *without* putting the thread scheduler code into the trusted computing base, yet still support thread-modular verification.

6.2.1 The Problem

Almost all work on concurrency verification assumes built-in language constructs for concurrency, including the work for low-level code, such as CCAP by Yu and Shao [102] and and CMAP shown in Chapter 5.

The top part of Figure 6.2 shows a (fairly low-level) abstract machine with built-in support of threads. Each thread has its own program counter (pc). The index i points to the current running thread. This index and the pc of the corresponding thread decide the next instruction to be executed by the machine. The machine provides a primitive yield instruction. Executing yield will change the index i in a nondeterministic way, therefore the control is transferred to another thread. All threads share the data heap $\mathbb{H}$ and the register file $\mathbb{R}$. The machine model is very similar to the one used in our CMAP, except that threads in this model also share the register file, *i.e.,* we do not save the register file as part of the execution context during context switching. It can be viewed as a simplified model of our CMAP machine.

As introduced in Chapter 5, the classic rely-guarantee method [59] allows concurrent code in such a machine to be certified in a thread modular way. The method assigns specifications $\mathbb{A}$ and $\mathbb{G}$ to each thread. $\mathbb{A}$ and $\mathbb{G}$ are predicates over a pair of states. They are used to specify state transitions. The guarantee $\mathbb{G}$ specifies state transitions made by the specified thread between two yield points. The assumption $\mathbb{A}$ specifies the expected state transition made by other threads while the specified thread is waiting for the processor. If all threads satisfy their specifications, the following non-interference property ensures proper collaboration between threads:

$$\mathtt{NI}[(\mathbb{A}_1, \mathbb{G}_1), \ldots, (\mathbb{A}_n, \mathbb{G}_n)] \triangleq \mathbb{G}_i \Rightarrow \mathbb{A}_j \qquad \forall i \neq j\,.$$

Note the definition of NI here is simpler than formula 5.1 in Chapter 5 because we do not need to handle thread-private register file here.

As explained in CMAP, to certify concurrent code, we prove that each thread fulfills its guarantee as long as its assumption is satisfied. When we certify one thread, we do not need knowledge about other threads. Therefore we do not have to worry about the exponential state space.

However, this beautiful abstraction also relies on the built-in thread abstraction. In a single processor machine such as our TM (see Figure 2.1), there is no built-in abstractions for threads. As shown in the bottom part of Figure 6.2, we have multiple execution contexts saved in heap as the thread queue. Code $\mathbb{C}_i$ *calls* the thread scheduler (implemented by $\mathbb{C}_S$), which switches the current context (pc) with one in the thread queue and *jumps* to the pc saved in the selected context. All we have at this level is sequential code.

It is difficult to use the rely-guarantee method to certify the whole system consisting of $\mathbb{C}_i$ and $\mathbb{C}_S$. We cannot treat $\mathbb{C}_S$ as a special thread because the context-switching behavior cannot be specified unless first-class code pointers is supported. It is already very challenging to support first-class code pointers in a sequential setting [75], and we do not know any existing work supporting first-class code pointers in a rely-guarantee based framework for concurrent code. On the other hand, certifying all the code, including $\mathbb{C}_i$, as sequential code loses the abstraction of threads and the nice thread modularity provided by the rely-guarantee method.

In this section, we use a variation of CMAP to certify user thread code $\mathbb{C}_i$. The variation, called CMAP^{-}, can be viewed as a simplified CMAP logic without considering thread creation, termination and thread-local register files. Each $\mathbb{C}_i$ is specified and certified following the rely-guarantee method, as if it works at the high-level machine shown in Figure 6.2, although it actually runs at the low-level machine and thread yielding is achieved by calling the function $\mathbb{C}_s$ that implements the scheduler instead of executing an abstract "yield" instruction. The scheduler code $\mathbb{C}_S$ is certified as sequential code in SCAP. From SCAP point of view, the context switching is no more special than memory load and store, as we will show below. Then the certified code can be linked in OCAP. This two-level approach also allows us to avoid treating code pointers stored in each thread context as first-class code pointers.

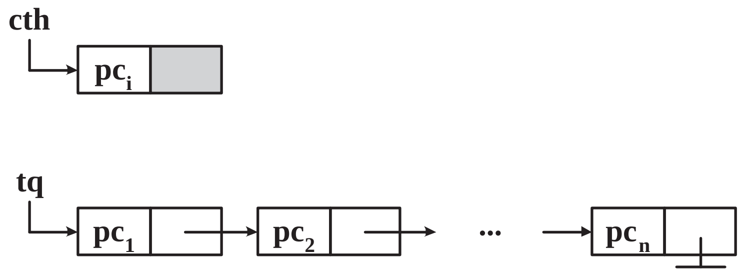


Figure 6.3: Current thread and the thread queue

6.2.2 Certifying The Scheduler Code in SCAP

User threads yield by calling the scheduler with the return address saved in register $\mathtt{r}_{31}$. The scheduler will save $\mathtt{r}_{31}$ in the current context, put the context in the thread queue, pick another execution context, restore $\mathtt{r}_{31}$, and finally return by jumping to $\mathtt{r}_{31}$. Then the control is transferred to the selected thread.

We have made several simplifications in the above procedure: we do not save the register file in the thread context because it is shared by threads. Also threads do not have stacks. Data structures for the scheduler is thus very simple, as shown in Figure 6.3. Each thread context only contains the saved pc. The global constant tq points to the queue containing execution contexts of ready threads. The queue is organized as a linked list. We use $\mathsf{TQ}(\mathtt{tq}, Q)$ to specify a well-formed linked list pointed by tq containing Q. Q is an abstract (nonempty) list of code labels $[\mathtt{pc}_1, \ldots, \mathtt{pc}_n]$. The global constant cth points to a node detached from the linked list. It contains the context of the current running thread and a dummy pointer pointing to the tail of the list.

The scheduler is then given the following specification $(\mathtt{p}_s, \mathtt{g}_s)$, where $|Q|$ represents

$$\begin{array}{rccl}
(\mathit{StPred}) & \mathtt{p}, \mathtt{q} & \in & \mathit{State} \to \mathsf{Prop} \\
(\mathit{Assumption}) & \mathbb{A} & \in & \mathit{State} \to \mathit{State} \to \mathsf{Prop} \\
(\mathit{Guarant.}) & \hat{\mathtt{g}}, \mathbb{G} & \in & \mathit{State} \to \mathit{State} \to \mathsf{Prop} \\
(\mathit{CdSpec}) & \theta & ::= & (\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}, \mathbb{G}) \\
(\mathit{CHSpec}) & \psi & ::= & \{(\mathtt{f}_0,\, \theta_0), \ldots, (\mathtt{f}_n,\, \theta_n)\}
\end{array}$$

Figure 6.4: Specification Constructs for CMAP⁻

the set of elements in the list Q.

$$\mathtt{p}_s \triangleq \exists Q.\ (\mathtt{cth} \mapsto \_) * (\mathtt{cth}{+}1 \mapsto \_) * \mathsf{TQ}(\mathtt{tq}, Q) * \mathsf{True}$$

$$\begin{array}{rl}
\mathtt{g}_s \triangleq & (\forall \mathtt{r} \in \mathtt{r}_0, \ldots \mathtt{r}_{30}.\mathtt{r} = \mathtt{r}') \wedge \\
& \forall Q.\exists \mathsf{pc}_x \in |Q| \cup \{\mathtt{r}_{31}\}.\exists Q'.(|Q'| = |Q| \cup \{\mathtt{r}_{31}\} \setminus \{\mathsf{pc}_x\}) \wedge \\
& \quad \mathtt{r}_{31}{}' = \mathsf{pc}_x \wedge \left\{ \begin{array}{l} (\mathtt{cth} \mapsto \_) \quad\ \ * (\mathtt{cth}{+}1 \mapsto \_) * \mathsf{TQ}(\mathtt{tq}, Q) \\ (\mathtt{cth} \mapsto \mathsf{pc}_x) * (\mathtt{cth}{+}1 \mapsto \_) * \mathsf{TQ}(\mathtt{tq}, Q') \end{array} \right\} * \mathsf{ID}
\end{array}$$

The precondition $\mathtt{p}_s$ simply requires that there is a node and a well-formed ready queue in the heap. The guarantee $\mathtt{g}_s$ requires that, at the return point of the scheduler, the register file (except $\mathtt{r}_{31}$) be restored; a label pc_x be picked from Q (or it can still be old $\mathtt{r}_{31}$) and be saved in $\mathtt{r}_{31}$; the thread queue be well-formed; and the rest part of data heap not be changed. Note $\mathtt{g}_s$ leaves the scheduling strategy unspecified. As we can see, this SCAP specification for the scheduler code $\mathbb{C}_S$ does not need any knowledge about the CMAP⁻ logic.

6.2.3 CMAP⁻ for User Thread Code

The code specifications in CMAP⁻ is also a quadruple $(\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}, \mathbb{G})$, as shown in Figure 6.4. $\mathbb{A}$, $\mathbb{G}$ and $\hat{\mathtt{g}}$ have similar meanings as their counterparts in CMAP, except that they are now predicates over a pair of program states in our low level machine shown in Chapter 2. As usual, $\mathtt{p}$ is a predicate over the current state.

We use $\mathcal{L}_{\mathrm{CMAP}^-}$ to represent the type of θ (in CiC). The following lift function converts

$\boxed{\psi \vdash \mathbb{C} : \psi'}$ **(*Well-formed code heap*)**

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

$\boxed{\psi \vdash \{(\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}, \mathbb{G})\}\, \mathtt{f} : \mathbb{I}}$ **(*Well-formed instruction sequence*)**

$$\frac{\begin{array}{l}\psi \vdash \{(\mathtt{p}', \hat{\mathtt{g}}', \mathbb{A}, \mathbb{G})\}\, \mathtt{f}{+}1 : \mathbb{I} \qquad \iota \in \{\mathsf{addu}, \mathsf{addiu}, \mathsf{lw}, \mathsf{subu}, \mathsf{sw}\} \\ \mathtt{p} \Rightarrow \mathtt{p}' \circ \mathsf{NextS}_{(\mathtt{f},\iota)} \quad \forall \mathbb{S}, \mathbb{S}'.\ \mathtt{p}\ \mathbb{S} \rightarrow \hat{\mathtt{g}}'\ (\mathsf{NextS}_{(\mathtt{f},\iota)}(\mathbb{S}))\ \mathbb{S}' \rightarrow \hat{\mathtt{g}}\ \mathbb{S}\ \mathbb{S}'\end{array}}{\psi \vdash \{(\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}, \mathbb{G})\}\, \mathtt{f} : \iota;\ \mathbb{I}} \ (\textsc{seq})$$

$$\frac{\begin{array}{l}\forall \mathbb{S}.\ \mathtt{p}\ \mathbb{S} \rightarrow \hat{\mathtt{g}}\ \mathbb{S}\ (\mathbb{S}.\mathbb{H}, \mathbb{S}.\mathbb{R}\{\mathtt{r}_{31} \rightsquigarrow \mathtt{f}{+}1\}) \\ \forall \mathbb{S}, \mathbb{S}'.\mathtt{p}\ \mathbb{S} \rightarrow \mathbb{A}\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathtt{p}\ \mathbb{S}' \quad (\mathtt{f}{+}1,\ (\mathtt{p}, \mathbb{G}, \mathbb{A}, \mathbb{G})) \in \psi\end{array}}{\psi \vdash \{(\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}, \mathbb{G})\}\, \mathtt{f} : \mathsf{jal}\ \mathtt{yield};\ \mathbb{I}} \ (\textsc{yield})$$

Figure 6.5: Selected CMAP⁻ Rules

ψ for CMAP⁻ to the OCAP code heap specification Ψ.

$$\llcorner \psi \lrcorner_\rho \triangleq \{(\mathtt{f}, \langle \rho, \mathcal{L}_{\text{CMAP}^-}, (\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}, \mathbb{G})\rangle) \mid (\mathtt{f},\ (\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}, \mathbb{G})) \in \psi\}$$

In Figure 6.5 we show some selected CMAP⁻ rules, which are very similar to CMAP rules. As we explained before, since this variation works in the low-level machine that does not have an abstract yield, it yields the control by calling the runtime library `yield` (*i.e.*, jal `yield`). The YIELD rule is revised accordingly. To certify the user thread code $\mathbb{C}_i$, we use CMAP⁻ rules and construct the derivation $\psi \vdash \mathbb{C}_i : \psi'$.

Program invariants and the interpretation. During program execution, we want the following invariants to hold at each state with specification $(\mathtt{p}, \hat{\mathtt{g}}, \mathbb{A}, \mathbb{G})$:

- $\mathtt{p}$ holds on the state visible to the user thread;
- there are well-formed thread queue Q and other runtime data structures as specified in Section 6.2.2;
- each pc_i in Q is a code pointer with specification $(\mathtt{p}_i, \mathbb{G}_i, \mathbb{A}_i, \mathbb{G}_i)$;

- assumptions and guarantees of threads (including the executing one) are compatible, *i.e.,* $\mathtt{NI}[\ldots,(\mathbb{A}_i,\mathbb{G}_i),\ldots,(\mathbb{A},\mathbb{G})]$;

- if $\mathtt{p}_i$ holds at a state $\mathbb{S}$, any state transition satisfies the assumption $\mathbb{A}_i$ does not break $\mathtt{p}_i$, *i.e.,* $\forall \mathbb{S},\mathbb{S}'.\ \mathtt{p}_i\ \mathbb{S} \rightarrow \mathbb{A}_i\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathtt{p}_i\ \mathbb{S}'$;

- when we reach a state that $\hat{\mathtt{g}}$ is satisfied (*i.e.,* the current thread can yield), it is safe for all threads in Q to take over the control, *i.e.,* $(\hat{\mathtt{g}}\ \mathbb{S}) \Rightarrow \mathtt{p}_i$ for all i, where $\mathbb{S}$ is the current program state.

These invariants are very similar to the invariants formalized by the PROG rule in CMAP. Here, the following interpretation for CMAP⁻ specification $(\mathtt{p},\hat{\mathtt{g}},\mathbb{A},\mathbb{G})$ formalizes these invariants.

$$
\begin{aligned}
&[\![\, (\mathtt{p},\hat{\mathtt{g}},\mathbb{A},\mathbb{G})\,]\!]^{\rho}_{\mathcal{L}_{\text{CMAP}^-}} \triangleq \lambda\Psi,\mathbb{S}.\\
&\qquad \exists \mathbb{H}_1,\mathbb{H}_2,Q.\ \mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{S}.\mathbb{H} \wedge \mathtt{p}\ (\mathbb{H}_1,\mathbb{S}.\mathbb{R}) \wedge\\
&\qquad\quad (\mathtt{cth} \mapsto \_ * \mathtt{cth}{+}1 \mapsto \_ * \mathsf{TQ}(\mathtt{tq},Q))\ (\mathbb{H}_2,\mathbb{S}.\mathbb{R}) \wedge\\
&\qquad\quad \mathsf{WFTQ}(Q,\hat{\mathtt{g}}\ (\mathbb{H}_1,\mathbb{S}.\mathbb{R}),\mathbb{A},\mathbb{G},\Psi)
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathsf{WFTQ}([\mathsf{pc}_1 \ldots \mathsf{pc}_n],\mathtt{q},\mathbb{A},\mathbb{G},\Psi) \triangleq\\
&\quad \forall i.\ \exists \mathtt{p}_i,\mathbb{A}_i,\mathbb{G}_i.\ (\mathsf{pc}_i,\langle \rho,\mathcal{L}_{\text{CMAP}^-},(\mathtt{p}_i,\mathbb{G}_i,\mathbb{A}_i,\mathbb{G}_i)\rangle) \in \Psi\\
&\qquad \wedge \mathtt{NI}[\ldots,(\mathbb{A}_i,\mathbb{G}_i),\ldots,(\mathbb{A},\mathbb{G})]\\
&\qquad \wedge (\forall \mathbb{S},\mathbb{S}'.\mathtt{p}_i\ \mathbb{S} \rightarrow \mathbb{A}_i\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathtt{p}_i\ \mathbb{S}') \wedge (\mathtt{q} \Rightarrow \mathtt{p}_i)
\end{aligned}
$$

6.2.4 Linking the scheduler with threads.

To link the certified scheduler with user thread code, we assign language IDs ρ and ρ' to SCAP and CMAP⁻ respectively. The following dictionary $\mathcal{D}_c$ contains the interpretation for CMAP⁻.

$$\mathcal{D}_c \triangleq \{\rho' \leadsto \langle \mathcal{L}_{\text{CMAP}^-}, [\![\, \_\,]\!]^{\rho'}_{\mathcal{L}_{\text{CMAP}^-}} \rangle\}\,.$$

Using $\mathcal{D}_c$ to instantiate the open parameter, SCAP interpretation is now $[\![\, \_\,]\!]^{(\rho,\mathcal{D}_c)}_{\mathcal{L}_{\text{SCAP}}}$ (see Chapter 4 for the definition). Since the scheduler has been certified, applying Theorem 4.1

will automatically convert the SCAP proof into OCAP proof.

However, CMAP⁻ derivations does not immediately give us a complete OCAP derivation. Readers may have notice that, in the YIELD rule, we jump to the `yield` without checking the specification of `yield`. It is not surprising that we cannot prove the YIELD rule as an OCAP lemma derivable from the OCAP JAL rule. Fortunately, the following theorem helps us construct sound OCAP proof from CMAP⁻ derivations after we know the specification of `yield` at the time of linkage. Proof of the theorem is similar to the soundness proof for CMAP and has been fully formalized in Coq.

Theorem 6.4 (CMAP⁻ Soundness)
Let $\mathcal{D} = \mathcal{D}_c \cup \{\rho \rightsquigarrow \langle \mathcal{L}_{\text{SCAP}},\ [\![\_]\!]^{(\rho,\mathcal{D}_c)}_{\mathcal{L}_{\text{SCAP}}}\ \rangle\}$ and $\Psi_s = \{(\texttt{yield},\ \langle \rho, \mathcal{L}_{\text{SCAP}}, (\texttt{p}_s, \texttt{g}_s)\ \rangle\ )\}$.

1. If we have $\psi \vdash \{(\texttt{p}, \hat{\texttt{g}}, \mathbb{A}, \mathbb{G})\}\,\texttt{f} :\ \mathbb{I}$, then $\mathcal{D} \vdash \{\langle \texttt{a} \rangle_{\Psi}\}\,\texttt{f} :\ \mathbb{I}$, where $\Psi = \llcorner \psi \lrcorner_{\rho'} \cup \Psi_s$ and $\texttt{a} = [\![\,(\texttt{p}, \hat{\texttt{g}}, \mathbb{A}, \mathbb{G})\,]\!]^{\rho'}_{\mathcal{L}_{\text{CMAP}^-}}$.

2. If we have $\psi \vdash \mathbb{C} : \psi'$ in CMAP⁻, then $\mathcal{D}; \Psi \vdash \mathbb{C} : \llcorner \psi' \lrcorner_{\rho'}$, where $\Psi = \llcorner \psi \lrcorner_{\rho'} \cup \Psi_s$.

6.3 Summary

In this chapter, we show two applications of the OCAP framework. The first one links TAL code with a simple memory allocation library. In original TAL, the memory allocation library is treated as a primitive in an abstract machine. The type system in TAL is too high-level to certify the implementation of the library itself. The TAL shown here is adapted to the low-level machine which contains no abstract primitives. The primitive instruction is replaced with a function call to the run-time library, which can be certified in SCAP. Then the well-typed TAL code is linked with certified SCAP code in our OCAP framework to get full certified software package.

In the second application, we link non-preemptive concurrent code with run-time library for yield. We can certify user threads using a high-level logic for concurrency verification, although there is no built-in abstraction of threads in our machine. The yield operation in the user thread code is implemented by calling the runtime function. We certify the implementation of the `yield` library in SCAP and link it with certified user threads. Before this work, we have never seen a fully certified concurrent program without trusting the implementation of thread libraries. Recently Ni *et al.* [77] certified a non-preemptive thread library using an extension of XCAP [75]. They treat code pointers saved in the threads' execution contexts as first class code pointers. As a result, specifications of thread control blocks have to use recursive types. It is also unclear how to link the thread library certified in this approach with user thread code.

Another interesting application of the OCAP framework is done by Lin *et al.* [64]. They used SCAP to certify a conservative garbage collector for TAL and linked it with TAL code in the OCAP framework. Guo *et al.* [43] also embedded CMAP into the OCAP framework and showed how to link multi-threaded user code certified in CMAP with certified thread libraries in the OCAP framework.

Chapter 7

R-G Reasoning and Concurrent Separation Logic

As we have discussed before, the rely-guarantee (R-G) method [59] (also known as assume-guarantee (A-G) method together with the assume-commit method by Misra and Chandy [67]) provides nice support of thread modularity in the sense that each thread is verified with regard to its own specifications, and without looking into code of other threads. Invariants of state transitions are specified using assumptions and guarantees. Each thread ensures that its atomic transitions satisfy its guarantee to the environment (*i.e.,* the collection of all other threads) as long as its assumption is satisfied by the environment. Non-interference is guaranteed as long as threads have compatible specifications, *i.e.,* the guarantee of each thread satisfies the assumptions of all other threads. As shown in Chapters 5 and 6, the R-G method is very general and does not require language constructs for synchronizations, such as locks or conditional critical regions (CCRs) [51]. However, we have to ensure each atomic transition made by a thread satisfies the thread's guarantee. In a preemptive setting, we have to check this for each atomic instruction, which complicates the verification procedure. Also, assumptions and guarantees are usually complicated and hard to define, because they specify global invariants for all shared resources during the program execution.

Another approach to modular verification of shared-state concurrent programs is concurrent separation logic (CSL) by Brookes and O'Hearn [14, 78, 16, 79]. It applies the local-reasoning idea from separation logic [57, 87] to verify shared-state concurrent programs with memory pointers. Separation logic assertions are used to capture ownerships of resources. Separating conjunction enforces the partition of resources. Verification of sequential threads in CSL is no different from verification of sequential programs. Memory modularity is supported by using separating conjunction and frame rules. However, following Owicki and Gries [81], CSL works only for *well-synchronized programs* in the sense that transfer of resource ownerships can only occur at entry and exit points of critical regions. It is unclear how to apply CSL to support general concurrent programs with ad-hoc synchronizations.

In this chapter we study the relationship between CSL and R-G reasoning. We propose the Separated Rely-Guarantee Logic (SAGL), which extends R-G reasoning with the local-reasoning idea in separation logic. Instead of treating all resources as shared, SAGL partitions resources into shared and private. Like in CSL, each thread has full access to its private resources, which are invisible to its environments. Shared resources can be accessed in two ways in SAGL: they can be accessed directly, or be converted into private first and then accessed. Conversions between shared and private can occur at any program point, instead of being coupled with critical regions. Both direct accesses and conversions are governed by guarantees, so that non-interference is ensured following R-G reasoning. Private resources are not specified in assumptions and guarantees, therefore specifications in SAGL are simpler and more modular than R-G reasoning.

Then we will show that CSL can be viewed as a specialization of SAGL with the invariant that *shared resources are well-formed outside of critical regions*. The specialization is pinned down by formalizing the CSL invariant as a specific assumption and guarantee in SAGL. Our formulation can also be viewed as a novel approach to proving the soundness of CSL. Different from Brookes' proof based on an action-trace semantics [14, 16], we prove that CSL inference rules are lemmas in SAGL with the specific assumption and guarantee. The soundness of SAGL is then proved following the syntactic approach to type soundness [99]. The proofs are formalized in the Coq proof assistant [24].

Here we study the relationship between R-G reasoning and CSL based on an extension of the MIPS-style low-level machine with built-in lock/unlock and memory allocation/free primitives, so that it can be readily applied to build certified low-level software packages. Another advantage is that the simplicity of the machine language and its semantics makes the formulation much simpler. For instance, we do not use variables, instead we only use register files and memory. Therefore we do not need to formalize the complicated syntactic constraints enforced in CSL over shared variables. Unlike the machine used for CMAP, the abstract machine works in a preemptive setting: execution of a thread can be preempted by others at any program point. We use this preemptive thread model so that the variation of CSL presented here is consistent with the origial work by Brookes and O'Hearn. The material in this chapter is derived from research I have done jointly with Rodrigo Ferreira and Zhong Shao [32].

7.1 The Abstract Machine

Figure 7.1 defines the model of an abstract machine and the syntax of the assembly language. The whole program state $\mathbb{P}$ contains a shared memory $\mathbb{H}$, a lock mapping $\mathbb{L}$ which maps a lock to the id of its owner thread, and n threads $[\mathbb{T}_1, \ldots, \mathbb{T}_n]$. The memory is modeled as a finite partial mapping from memory locations `l` (natural numbers) to word values (natural numbers). Each thread $\mathbb{T}_i$ contains its own code heap $\mathbb{C}$, register file $\mathbb{R}$, the instruction sequence $\mathbb{I}$ that is currently being executed, and its thread id i. Here we allow each thread to have its own register file, which is consistent with most implementation of thread libraries where the register file is saved in the execution context when a thread is preempted.

The code heap $\mathbb{C}$ maps code labels to instruction sequences, which is a list of assembly instructions ending with a jump instruction. The set of instructions we present here are

$$
\begin{array}{rlll}
(\textit{Program}) & \mathbb{P} & ::= & (\mathbb{H}, [\mathbb{T}_1, \ldots, \mathbb{T}_n], \mathbb{L}) \\
(\textit{Thread}) & \mathbb{T}_i & ::= & (\mathbb{C}, \mathbb{R}, \mathbb{I}, i) \\
(\textit{CodeHeap}) & \mathbb{C} & \in & \textit{Labels} \rightharpoonup \textit{InstrSeq} \\
(\textit{Memory}) & \mathbb{H} & \in & \textit{Labels} \rightharpoonup \textit{Word} \\
(\textit{RegFile}) & \mathbb{R} & \in & \textit{Register} \rightarrow \textit{Word} \\
(\textit{LockMap}) & \mathbb{L} & ::= & \textit{Locks} \rightharpoonup \{1, \ldots, n\} \\
(\textit{Register}) & \mathtt{r} & ::= & \mathtt{r}_0 \mid \ldots \mid \mathtt{r}_{31} \\
(\textit{Labels}) & \mathtt{f}, \mathtt{l} & ::= & n \ \ (\textit{nat nums}) \\
(\textit{Locks}) & l & ::= & n \ \ (\textit{nat nums}) \\
(\textit{Word}) & \mathtt{w} & ::= & i \ \ (\textit{integers}) \\
(\textit{InstrSeq}) & \mathbb{I} & ::= & \mathsf{j}\ \mathtt{f} \mid \mathsf{jr}\ \mathtt{r}_s \mid \iota; \mathbb{I} \\
(\textit{Instr}) & \iota & ::= & \mathsf{addu}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \mid \mathsf{addiu}\ \mathtt{r}_d, \mathtt{r}_s, i \mid \mathsf{alloc}\ \mathtt{r}_d, \mathtt{r}_s \mid \mathsf{beq}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f} \mid \mathsf{bgtz}\ \mathtt{r}_s, \mathtt{f} \\
 & & \mid & \mathsf{free}\ \mathtt{r}_s \mid \mathsf{lock}\ l \mid \mathsf{lw}\ \mathtt{r}_t, i(\mathtt{r}_s) \mid \mathsf{subu}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \mid \mathsf{sw}\ \mathtt{r}_t, i(\mathtt{r}_s) \mid \mathsf{unlock}\ l
\end{array}
$$

Figure 7.1: The Abstract Machine

the commonly used subsets in RISC machines. We also use lock/unlock primitives to do synchronization, and use alloc/free to do dynamic memory allocation and free.

The step relation ($\longmapsto$) of program states ($\mathbb{P}$) is defined in Figure 7.2. We use the auxiliary relation $(\mathbb{H}, \mathbb{T}, \mathbb{L}) \overset{\mathrm{t}}{\longmapsto} (\mathbb{H}', \mathbb{T}', \mathbb{L}')$ to define the effects of the execution of the thread $\mathbb{T}$. Here we follow the preemptive thread model where execution of threads can be preempted at any program point, but execution of individual instructions is *atomic*. Operational semantics of most instructions are standard. Note that we do not support reentrant-locks. If the lock l has been acquired, execution of the "lock l" instruction will be blocked even if the lock is owned by the current thread. The relation NextS_ι defines the effects of the sequential instruction ι over memory and register files. Its definition is very similar to the NextS$relationdefinedin$Figure 2.3, except that we have two new instructions here, *i.e.*, alloc and free.

The abstract machine is similar to the one for CMAP, but it supports preemptive threads and there is no yield instruction. Also, it does not have fork and exit, which are orthogonal to our focus on the relationship between R-G reasoning and CSL. The machine has lock and unlock instructions, which are used to simulate CCRs at higher level, as

$$(\mathbb{H}, [\mathbb{T}_1, \ldots, \mathbb{T}_n], \mathbb{L}) \longmapsto (\mathbb{H}', [\mathbb{T}_1, \ldots, \mathbb{T}_{k-1}, \mathbb{T}'_k, \mathbb{T}_{k+1}, \ldots, \mathbb{T}_n], \mathbb{L}')$$
$$\text{if } (\mathbb{H}, \mathbb{T}_k, \mathbb{L}) \overset{\mathrm{t}}{\longmapsto} (\mathbb{H}', \mathbb{T}'_k, \mathbb{L}') \text{ for any } k;$$
$$\text{where}$$

$(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}, k), \mathbb{L}) \overset{\mathrm{t}}{\longmapsto} (\mathbb{H}', \mathbb{T}', \mathbb{L}')$		
if $\mathbb{I} =$	then $(\mathbb{H}', \mathbb{T}', \mathbb{L}') =$	
j $\mathtt{f}$	$(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L})$	where $\mathbb{I}' = \mathbb{C}(\mathtt{f})$
jr $\mathtt{r}_s$	$(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L})$	where $\mathbb{I}' = \mathbb{C}(\mathbb{R}(\mathtt{r}_s))$

bgtz $\mathtt{r}_s, \mathtt{f}; \mathbb{I}'$	$(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L})$
	 $(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}'', k), \mathbb{L})$
		if $\mathbb{R}(\mathtt{r}_s) \leq 0$
		 if $\mathbb{R}(\mathtt{r}_s) > 0$ and $\mathbb{I}'' = \mathbb{C}(\mathtt{f})$

beq $\mathtt{r}_s, \mathtt{r}_t, \mathtt{f}; \mathbb{I}'$	$(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L})$
	 $(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}'', k), \mathbb{L})$
		if $\mathbb{R}(\mathtt{r}_s) \neq \mathbb{R}(\mathtt{r}_t)$
		 if $\mathbb{R}(\mathtt{r}_s) = \mathbb{R}(\mathtt{r}_t)$ and $\mathbb{I}'' = \mathbb{C}(\mathtt{f})$

lock $l; \mathbb{I}'$	$(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L}\{l \rightsquigarrow k\})$
	 $(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}, k), \mathbb{L})$
		if $l \notin dom(\mathbb{L})$
		 if $l \in dom(\mathbb{L})$

unlock $l; \mathbb{I}'$	$(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L} \setminus \{\{l\}\})$	if $\mathbb{L}(l) = k$
$\iota; \mathbb{I}'$ for other ι	$(\mathbb{H}', (\mathbb{C}, \mathbb{R}', \mathbb{I}', k), \mathbb{L})$	where $(\mathbb{H}', \mathbb{R}') = \mathsf{NextS}_\iota(\mathbb{H}, \mathbb{R})$

and

if $\iota =$	then $\mathsf{NextS}_\iota(\mathbb{H}, \mathbb{R}) =$	
addu $\mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) + \mathbb{R}(\mathtt{r}_t)\})$	
addiu $\mathtt{r}_d, \mathtt{r}_s, i$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) + i\})$	
lw $\mathtt{r}_t, i(\mathtt{r}_s)$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_t \rightsquigarrow \mathbb{H}(\mathbb{R}(\mathtt{r}_s) + i)\})$	when $\mathbb{R}(\mathtt{r}_s) + i \in dom(\mathbb{H})$
subu $\mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) - \mathbb{R}(\mathtt{r}_t)\})$	
sw $\mathtt{r}_t, i(\mathtt{r}_s)$	$(\mathbb{H}\{\mathbb{R}(\mathtt{r}_s) + i \rightsquigarrow \mathbb{R}(\mathtt{r}_t)\}, \mathbb{R})$	when $\mathbb{R}(\mathtt{r}_s) + i \in dom(\mathbb{H})$
alloc $\mathtt{r}_d, \mathtt{r}_s$	$(\mathbb{H}\{\mathtt{l}, \ldots, \mathtt{l} + \mathbb{R}(\mathtt{r}_s) - 1 \rightsquigarrow \_\}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathtt{l}\})$	where $\mathtt{l}, \ldots, \mathtt{l} + \mathbb{R}(\mathtt{r}_s) - 1 \notin dom(\mathbb{H})$
free $\mathtt{r}_s$	$(\mathbb{H} \setminus \{\mathbb{R}(\mathtt{r}_s)\}, \mathbb{R})$	when $\mathbb{R}(\mathtt{r}_s) \in dom(\mathbb{H})$

Figure 7.2: Operational Semantics of the Machine

used in the original work on CSL [78, 14].

Note the way we distinguish "blocking" states from "stuck" states caused by unsafe operations, *e.g.,* freeing dangling pointers. If an unsafe operation is made, there is no resulting state satisfying the step relation ($\overset{\mathrm{t}}{\longmapsto}$) for the current thread. If a thread tries to acquire a lock which has been taken, it stutters: the resulting state will be the same as the current one (therefore the lock instruction will be executed again).

$$
\begin{array}{rlll}
(\textit{XState}) & \mathbb{X} & ::= & (\mathbb{H}, (\mathbb{R}, i), \mathbb{L}) \\
(\textit{ProgSpec}) & \Phi & ::= & ([\Psi_1, \ldots, \Psi_n], [(\mathbb{A}_1, \mathbb{G}_1), \ldots, (\mathbb{A}_n, \mathbb{G}_n)]) \\
(\textit{CHSpec}) & \Psi & ::= & \{\mathtt{f} \rightsquigarrow \mathtt{a}\}^* \\
(\textit{Assert}) & \mathtt{a} & \in & \textit{XState} \to \mathsf{Prop} \\
(\textit{Assume}) & \mathbb{A} & \in & \textit{XState} \to \textit{XState} \to \mathsf{Prop} \\
(\textit{Guarantee}) & \mathbb{G} & \in & \textit{XState} \to \textit{XState} \to \mathsf{Prop}
\end{array}
$$

Figure 7.3: Specification Constructs for AGL

7.2 AGL: an R-G Based Program Logic

In this section we present an R-G based program logic (AGL) for our assembly language. AGL is a variation of the CMAP⁻ logic shown in Chapter 6 or the CCAP logic by Yu and Shao [102] which applies the R-G method for assembly code verification, but it works for the preemptive thread model instead of the non-preemptive model.

Figure 7.3 shows the specification constructs for AGL. For each thread in the program, its specification contains three parts: the specification Ψ for the code heap, the assumption $\mathbb{A}$ and the guarantee $\mathbb{G}$. The specification Φ of the whole program just groups specifications for each thread.

Assumptions and guarantees are meta-logic predicates over a pair of extended thread states $\mathbb{X}$, which contains the shared memory $\mathbb{H}$, the thread's register file $\mathbb{R}$ and id k, and the global lock mapping $\mathbb{L}$. As in CMAP, the assumption $\mathbb{A}$ for a thread specifies the expected invariant of state transitions made by the environment. The arguments it takes are extended thread states before and after a transition, respectively. The guarantee $\mathbb{G}$ of a thread specifies the invariant of state transitions made by the thread itself.

The code heap specification Ψ assigns a precondition $\mathtt{a}$ to each instruction sequence in the code heap $\mathbb{C}$. The assertion $\mathtt{a}$ is a meta-logic predicate over the extended thread state $\mathbb{X}$. It ensures the safe execution of the corresponding instruction sequence. As before, we do not assign postconditions to instruction sequences. Since each instruction sequence ends with a jump instruction, we use the assertion at the target address as the postcondition.

Note we do not need a local guarantee $\hat{\mathtt{g}}$ here as we did for CMAP, because we are

$\boxed{\Phi, [\mathtt{a}_1, \ldots, \mathtt{a}_n] \vdash \mathbb{P}}$ ***(Well-formed program)***

$$\frac{\begin{array}{c}\Phi = ([\Psi_1, \ldots, \Psi_n], [(\mathbb{A}_1, \mathbb{G}_1), \ldots, (\mathbb{A}_n, \mathbb{G}_n)]) \\ \mathtt{NI}([(\mathbb{A}_1, \mathbb{G}_1), \ldots, (\mathbb{A}_n, \mathbb{G}_n)]) \qquad \Psi_k, \mathbb{A}_k, \mathbb{G}_k \vdash \{\mathtt{a}_k\}\, (\mathbb{H}, \mathbb{T}_k, \mathbb{L}) \text{ for all } k\end{array}}{\Phi, [\mathtt{a}_1, \ldots, \mathtt{a}_n] \vdash (\mathbb{H}, [\mathbb{T}_1, \ldots, \mathbb{T}_n], \mathbb{L})} \text{ (PROG)}$$

$\boxed{\Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathtt{a}\}\, (\mathbb{H}, \mathbb{T}, \mathbb{L})}$ ***(Well-formed thread)***

$$\frac{\mathtt{a}\, (\mathbb{H}, (\mathbb{R}, k), \mathbb{L}) \qquad \Psi, \mathbb{A}, \mathbb{G} \vdash \mathbb{C} : \Psi \qquad \Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathtt{a}\}\, \mathbb{I}}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathtt{a}\}\, (\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}, k), \mathbb{L})} \text{ (THRD)}$$

$\boxed{\Psi, \mathbb{A}, \mathbb{G} \vdash \mathbb{C} : \Psi'}$ ***(Well-formed code heap)***

$$\frac{\forall \mathtt{f} \in dom(\Psi') : \qquad \Psi, \mathbb{A}, \mathbb{G} \vdash \{\Psi'(\mathtt{f})\}\, \mathbb{C}(\mathtt{f})}{\Psi, \mathbb{A}, \mathbb{G} \vdash \mathbb{C} : \Psi'} \text{ (CDHP)}$$

$\boxed{\Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathtt{a}\}\, \mathbb{I}}$ ***(Well-formed instr. sequences)***

$$\frac{\Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathtt{a}\}\, \iota\, \{\mathtt{a}'\} \qquad \Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathtt{a}'\}\, \mathbb{I} \qquad (\mathtt{a} \circ \mathbb{A}) \Rightarrow \mathtt{a}}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathtt{a}\}\, \iota; \mathbb{I}} \text{ (SEQ)}$$

$$\frac{\forall \mathbb{X} @ (\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ \mathtt{a}\, \mathbb{X} \rightarrow \Psi(\mathbb{R}(\mathtt{r}_s))\, \mathbb{X} \qquad (\mathtt{a} \circ \mathbb{A}) \Rightarrow \mathtt{a}}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathtt{a}\}\, \mathsf{jr}\ \mathtt{r}_s} \text{ (JR)}$$

$$\frac{\forall \mathbb{X}.\ \mathtt{a}\, \mathbb{X} \rightarrow \Psi(\mathtt{f})\, \mathbb{X} \qquad (\mathtt{a} \circ \mathbb{A}) \Rightarrow \mathtt{a}}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathtt{a}\}\, \mathsf{j}\ \mathtt{f}} \text{ (J)}$$

Figure 7.4: AGL Inference Rules

using a preemptive thread model, and each atomic transition is made by only one machine instruction instead of a sequence of instructions between two yield instructions. So the state transition made by each instruction needs to satisfy the global guarantee $\mathbb{G}$. Also, $\mathbb{A}$ and $\mathbb{G}$ are not changing during threads' execution because we are not dealing with dynamic thread creation.

Inference rules. Inference rules of AGL are presented in Figs. 7.4 and 7.5. The PROG rule defines the well-formedness of the program $\mathbb{P}$ with respect to the program specification Φ and the set of preconditions ($[\mathtt{a}_1, \ldots, \mathtt{a}_n]$) for the instruction sequences that are currently executed by all the threads. Checking the well-formedness of $\mathbb{P}$ involves two

$\boxed{\Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathsf{a}\}\, \iota\, \{\mathsf{a}'\}}$ ***(Well-formed instructions)***

$$\frac{\begin{array}{l}\forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ \mathsf{a}\ \mathbb{X} \wedge l \notin dom(\mathbb{L}) \rightarrow \mathsf{a}'\ \mathbb{X}' \wedge \mathbb{G}\ \mathbb{X}\ \mathbb{X}' \\ \qquad\qquad\qquad \text{where } \mathbb{X}' = (\mathbb{H}, (\mathbb{R}, k), \mathbb{L}\{l \rightsquigarrow k\}).\end{array}}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathsf{a}\}\, \mathsf{lock}\ l\, \{\mathsf{a}'\}}\ (\text{LOCK})$$

$$\frac{\begin{array}{l}\forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ \mathsf{a}\ \mathbb{X} \rightarrow \mathbb{L}(l) = k \wedge \mathsf{a}'\ \mathbb{X}' \wedge \mathbb{G}\ \mathbb{X}\ \mathbb{X}' \\ \qquad\qquad\qquad \text{where } \mathbb{X}' = (\mathbb{H}, (\mathbb{R}, k), \mathbb{L} \setminus \{l\}).\end{array}}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathsf{a}\}\, \mathsf{unlock}\ l\, \{\mathsf{a}'\}}\ (\text{UNLOCK})$$

$$\frac{\begin{array}{l}\forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\forall \mathtt{l}.\ \mathsf{a}\ \mathbb{X} \wedge (\{\mathtt{l}, \ldots, \mathtt{l}+\mathbb{R}(\mathtt{r}_s)-1\} \cap dom(\mathbb{H}) = \emptyset) \rightarrow \\ \qquad\qquad\qquad\qquad \mathbb{R}(\mathtt{r}_s) > 0 \wedge \mathsf{a}'\ \mathbb{X}' \wedge \mathbb{G}\ \mathbb{X}\ \mathbb{X}' \\ \qquad \text{where } \mathbb{X}' = (\mathbb{H}\{\mathtt{l}, \ldots, \mathtt{l}+\mathbb{R}(\mathtt{r}_s)-1 \rightsquigarrow \_\}, (\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathtt{l}\}, k), \mathbb{L})\end{array}}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathsf{a}\}\, \mathsf{alloc}\ \mathtt{r}_d, \mathtt{r}_s\, \{\mathsf{a}'\}}\ (\text{ALLOC})$$

$$\frac{\begin{array}{l}\forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ \mathsf{a}\ \mathbb{X} \rightarrow \mathbb{R}(\mathtt{r}_s) \in dom(\mathbb{H}) \wedge \mathsf{a}'\ \mathbb{X}' \wedge \mathbb{G}\ \mathbb{X}\ \mathbb{X}' \\ \qquad\qquad\qquad \text{where } \mathbb{X}' = (\mathbb{H} \setminus \{\mathbb{R}(\mathtt{r}_s)\}, (\mathbb{R}, k), \mathbb{L})\end{array}}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathsf{a}\}\, \mathsf{free}\ \mathtt{r}_s\, \{\mathsf{a}'\}}\ (\text{FREE})$$

$$\frac{\begin{array}{l}\forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ \mathsf{a}\ \mathbb{X} \rightarrow (\mathbb{R}(\mathtt{r}_s)+i) \in dom(\mathbb{H}) \wedge \mathsf{a}'\ \mathbb{X}' \wedge \mathbb{G}\ \mathbb{X}\ \mathbb{X}' \\ \qquad\qquad\qquad \text{where } \mathbb{X}' = (\mathbb{H}\{\mathbb{R}(\mathtt{r}_s)+i \rightsquigarrow \mathbb{R}(\mathtt{r}_t)\}, (\mathbb{R}, k), \mathbb{L})\end{array}}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathsf{a}\}\, \mathsf{sw}\ \mathtt{r}_t, i(\mathtt{r}_s)\, \{\mathsf{a}'\}}\ (\text{SW})$$

$$\frac{\begin{array}{l}\forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ \mathsf{a}\ \mathbb{X} \rightarrow (\mathbb{R}(\mathtt{r}_s)+i) \in dom(\mathbb{H}) \wedge \mathsf{a}'\ \mathbb{X}' \\ \qquad\qquad\qquad \text{where } \mathbb{X}' = (\mathbb{H}, (\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{H}(\mathbb{R}(\mathtt{r}_s)+i)\}, k), \mathbb{L})\end{array}}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathsf{a}\}\, \mathsf{lw}\ \mathtt{r}_d, i(\mathtt{r}_s)\, \{\mathsf{a}'\}}\ (\text{LW})$$

$$\frac{\forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ \mathsf{a}\ \mathbb{X} \rightarrow ((\mathbb{R}(\mathtt{r}_s) \leq 0 \rightarrow \mathsf{a}'\ \mathbb{X}) \wedge (\mathbb{R}(\mathtt{r}_s) > 0 \rightarrow \Psi(\mathtt{f})\ \mathbb{X}))}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathsf{a}\}\, \mathsf{bgtz}\ \mathtt{r}_s, \mathtt{f}\, \{\mathsf{a}'\}}\ (\text{BGTZ})$$

$$\frac{\forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ \mathsf{a}\ \mathbb{X} \rightarrow ((\mathbb{R}(\mathtt{r}_s) \neq \mathbb{R}(\mathtt{r}_t) \rightarrow \mathsf{a}'\ \mathbb{X}) \wedge (\mathbb{R}(\mathtt{r}_s) = \mathbb{R}(\mathtt{r}_t) \rightarrow \Psi(\mathtt{f})\ \mathbb{X}))}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathsf{a}\}\, \mathsf{beq}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f}\, \{\mathsf{a}'\}}\ (\text{BEQ})$$

$$\frac{\begin{array}{l}\forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ \mathsf{a}\ \mathbb{X} \rightarrow \mathsf{a}'\ \mathbb{X}' \qquad\qquad \iota \in \{\mathsf{addu}, \mathsf{addiu}, \mathsf{subu}\} \\ \qquad\qquad \text{where } \mathbb{X}' = (\mathbb{H}', (\mathbb{R}', k), \mathbb{L}),\ \text{and } (\mathbb{H}', \mathbb{R}') = \mathsf{NextS}_\iota(\mathbb{H}, \mathbb{R})\end{array}}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{\mathsf{a}\}\, \iota\, \{\mathsf{a}'\}}\ (\text{SIMPLE})$$

Figure 7.5: AGL Inference Rules (cont'd)

steps. First we check the compatibility of assumptions and guarantees for all the threads. The predicate NI is redefined here for the new settings:

$$\begin{aligned}\mathtt{NI}([(\mathbb{A}_1,\mathbb{G}_1),\ldots,(\mathbb{A}_n,\mathbb{G}_n)]) \;&\triangleq \forall i,j,\mathbb{H},\mathbb{H}',\mathbb{R}_i,\mathbb{R}_i',\mathbb{R}_j,\mathbb{L},\mathbb{L}'.\\ &i\neq j \rightarrow \mathbb{G}_i\;(\mathbb{H},(\mathbb{R}_i,i),\mathbb{L})\;(\mathbb{H}',(\mathbb{R}_i',i),\mathbb{L}') \rightarrow \mathbb{A}_j\;(\mathbb{H},(\mathbb{R}_j,j),\mathbb{L})\;(\mathbb{H}',(\mathbb{R}_j,j),\mathbb{L}')\,,\end{aligned} \tag{7.1}$$

which simply says that the guarantee of each thread should satisfy assumptions of all other threads. Then we apply the THRD rule to check that implementation of each thread actually satisfies the specification. Each thread $\mathbb{T}_i$ is verified separately, therefore thread modularity is supported.

In the THRD rule, we require that the precondition $\mathtt{a}$ be satisfied by the current extended thread state $(\mathbb{H},(\mathbb{R},k),\mathbb{L})$; that the thread code heap satisfy its specification Ψ, $\mathbb{A}$ and $\mathbb{G}$; and that it be safe to execute the current instruction sequence $\mathbb{I}$ under the precondition $\mathtt{a}$ and the thread specification.

The CDHP rule checks the well-formedness of thread code heaps. It requires that each instruction sequence specified in Ψ' be well-formed with respect to the imported interfaces specified in Ψ, the assumption $\mathbb{A}$ and the guarantee $\mathbb{G}$.

The SEQ rule and the JR rule ensure that it is safe to execute the instruction sequence if the precondition is satisfied. If the instruction sequence starts with a normal sequential instruction ι, we need to come up with an assertion $\mathtt{a}'$ which serves both as the postcondition of ι and as the precondition of the remaining instruction sequence. Also we need to ensure that, if the current thread is preempted at a state satisfying $\mathtt{a}$, $\mathtt{a}$ must be preserved by any state transitions (by other threads) satisfying the assumption $\mathbb{A}$. This is enforced by $(\mathtt{a}\circ\mathbb{A})\Rightarrow\mathtt{a}$:

$$(\mathtt{a}\circ\mathbb{A})\Rightarrow\mathtt{a} \;\triangleq\; \forall\mathbb{X},\mathbb{X}'.\,\mathtt{a}\;\mathbb{X}\wedge\mathbb{A}\;\mathbb{X}\;\mathbb{X}'\rightarrow\mathtt{a}\;\mathbb{X}'\,.$$

If we reach the last jump instruction of the instruction sequence, the JR rule requires that the assertion assigned to the target address in Ψ be satisfied after the jump. It also requires that $\mathtt{a}$ be preserved by state transitions satisfying $\mathbb{A}$. Here we use the syntactic sugar $\forall X@(x_1, \ldots, x_n).\ P(X, x_1, \ldots, x_n)$ to mean that, for all tuple X containing elements $x_1, \ldots, x_n$, the predicate P holds. It is formally defined as:

$$\forall X, x_1, \ldots, x_n.(X = (x_1, \ldots, x_n)) \rightarrow P(X, x_1, \ldots, x_n)\,.$$

The notation $\lambda X@(x_1, \ldots, x_n).\ f(X, x_1, \ldots, x_n)$ that we use later is defined similarly. The rule for direct jumps ($\mathtt{j\ f}$) is similar to the JR rule and requires no further explanation.

Instruction rules require that the precondition ensure the safe execution of the instruction; and that the resulting state satisfy the postcondition. Also, if shared states ($\mathbb{H}$ and $\mathbb{L}$) are updated by the instruction, we need to ensure that the update satisfies the guarantee $\mathbb{G}$. For the lock instruction, if the control falls through, we know that the lock is not held by any thread. This extra knowledge can be used together with the precondition $\mathtt{a}$ to show the postcondition is satisfied by the resulting state. The rest of instruction rules are straightforward and will not be explained here.

Soundness. The soundness of AGL shows that the PROG rule enforces the non-interference and the partial correctness of programs with respect to the specifications. It is proved following the syntactic approach [99] to proving soundness of type systems. We first prove the following progress and preservation lemma.

Lemma 7.1 (AGL-Progress) For any program $\mathbb{P} = (\mathbb{H}, [\mathbb{T}_1, \ldots, \mathbb{T}_n], \mathbb{L})$, if $\Phi, [\mathtt{a}_1, \ldots, \mathtt{a}_n] \vdash \mathbb{P}$, then, for any thread $\mathbb{T}_k$, there exist $\mathbb{H}'$, $\mathbb{T}'_k$ and $\mathbb{L}'$ such that $(\mathbb{H}, \mathbb{T}_k, \mathbb{L}) \overset{\mathtt{t}}{\longmapsto} (\mathbb{H}', \mathbb{T}'_k, \mathbb{L}')$.

Lemma 7.2 (AGL-Preservation) If $\Phi, [\mathtt{a}_1, \ldots, \mathtt{a}_n] \vdash \mathbb{P}$ and $(\mathbb{P} \longmapsto \mathbb{P}')$, then there exist $\mathtt{a}'_1, \ldots, \mathtt{a}'_n$ such that $\Phi, [\mathtt{a}'_1, \ldots, \mathtt{a}'_n] \vdash \mathbb{P}'$.

The soundness theorem follows the progress and preservation lemmas.

Theorem 7.3 (AGL-Soundness) For any program $\mathbb{P}$ with specification $\Phi = [\Psi_1, \ldots, \Psi_n], [(\mathbb{A}_1, \mathbb{G}_1), \ldots, (\mathbb{A}_n, \mathbb{G}_n)]$, if $\Phi, [\mathtt{a}_1, \ldots, \mathtt{a}_n] \vdash \mathbb{P}$, then,

- for any m, there exists a $\mathbb{P}'$ such that $(\mathbb{P} \longmapsto^m \mathbb{P}')$;
- for any m and $\mathbb{P}' = (\mathbb{H}', [\mathbb{T}'_1, \ldots, \mathbb{T}'_n], \mathbb{L}')$, if $(\mathbb{P} \longmapsto^m \mathbb{P}')$, then,
  - $\Phi, [\mathtt{a}'_1, \ldots, \mathtt{a}'_n] \vdash \mathbb{P}'$ for some $\mathtt{a}'_1, \ldots, \mathtt{a}'_n$;
  - for any k, there exist $\mathbb{H}''$, $\mathbb{T}''_k$ and $\mathbb{L}''$ such that $(\mathbb{H}', \mathbb{T}'_k, \mathbb{L}') \overset{\mathrm{t}}{\longmapsto} (\mathbb{H}'', \mathbb{T}''_k, \mathbb{L}'')$;
  - for any k, if $\mathbb{T}'_k = (\mathbb{C}_k, \mathbb{R}'_k, \mathtt{j}\ \mathtt{f}, k)$, then $\Psi_k(\mathtt{f})\ (\mathbb{H}', (\mathbb{R}'_k, k), \mathbb{L}')$ holds;
  - for any k, if $\mathbb{T}'_k = (\mathbb{C}_k, \mathbb{R}'_k, \mathtt{jr}\ \mathtt{r}_s, k)$, then $\Psi_k(\mathbb{R}'_k(\mathtt{r}_s))\ (\mathbb{H}', (\mathbb{R}'_k, k), \mathbb{L}')$ holds;
  - for any k, if $\mathbb{T}'_k = (\mathbb{C}_k, \mathbb{R}'_k, \mathtt{beq}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f}; \mathbb{I}, k)$ and $\mathbb{R}'_k(\mathtt{r}_s) = \mathbb{R}'_k(\mathtt{r}_t)$,
    then $\Psi_k(\mathtt{f})\ (\mathbb{H}', (\mathbb{R}'_k, k), \mathbb{L}')$ holds;
  - for any k, if $\mathbb{T}'_k = (\mathbb{C}_k, \mathbb{R}'_k, \mathtt{bgtz}\ \mathtt{r}_s, \mathtt{f}; \mathbb{I}, k)$ and $\mathbb{R}'_k(\mathtt{r}_s) > 0$,
    then $\Psi_k(\mathtt{f})\ (\mathbb{H}', (\mathbb{R}'_k, k), \mathbb{L}')$ holds.

Note that our AGL does not guarantee deadlock-freedom, which can be easily achieved by enforcing a partial order of lock acquiring in the LOCK rule. We do not show the proof details here, which are very similar to the proof of SAGL's soundness shown in the next section.

7.3 SAGL: Separated Rely-Guarantee Logic

AGL is a general program logic supporting thread modular verification of concurrent code. However, because it treats all memory as shared resources, it does not have good memory modularity, and assumptions and guarantees are hard to define and use. For instance, suppose we partition the memory into n blocks and each block is assigned to one thread. Each thread simply works on its own part of memory and never accesses other parts. This scenario is not unusual in parallel programs and non-interference is trivially satisfied. However, to certify the code in AGL, the assumption for each thread has to be like "my private part of memory is never updated by others", and the guarantee is like "I will not touch other threads' private memory". During program verification, we

$$\begin{array}{rrcl}(\mathit{CHSpec}) & \Psi & ::= & \{\mathtt{f} \leadsto (\mathtt{a}, \nu)\}^* \\ (\mathit{Assert}) & \mathtt{a}, \nu & \in & \mathit{XState} \rightarrow \mathsf{Prop}\end{array}$$

Figure 7.6: Extension of AGL Specification Constructs in SAGL

have to prove for each individual instruction that the guarantee is not broken, even if it is trivially true. To make things worse, if each thread dynamically allocates memory and uses the allocated memory as private resources, as shown in the example in Section 7.5, the domain of private memory becomes dynamic, which makes it very hard to define the assumption and guarantee.

In this section, we extend AGL with explicit partition of private resources and shared resources. The extended logic, which we call Separated Rely-Guarantee Logic (SAGL), has much better support of memory modularity than AGL without sacrificing any expressiveness. Borrowing the local-reasoning idea in separation logic, private resources of one thread are not visible to other threads, therefore will not be touched by others. Assumptions and guarantees in SAGL only specifies shared resources, therefore in the scenario above the definition of SAGL assumption and guarantee becomes completely trivial since there is no shared resources. The dynamic domain of private memory caused by dynamic memory allocation/free is no longer a challenge to define assumptions and guarantees because private memory does not have to be specified.

Figure 7.6 shows the extensions of AGL specifications for SAGL. In the specification Ψ of each thread code heap, the precondition assigned to each code label now becomes a pair of assertions $(\mathtt{a}, \nu)$. The assertion $\mathtt{a}$ plays the same role as in AGL. It specifies the shared resources (all memory are treated as shared in AGL). The assertion ν specifies the private resources of the thread. Other threads' private resources are not specified.

Inference rules. The inference rules of SAGL are shown in Figures 7.7 and 7.8. They look very similar to AGL rules. In the PROG rule, as in AGL, we check the compatibility of assumptions and guarantees, and check the well-formedness of each thread. However,

$\boxed{\Phi, [(\mathtt{a}_1,\nu_1),\ldots,(\mathtt{a}_n,\nu_n)] \vdash \mathbb{P}}$ ***(Well-formed program)***

$$\frac{\begin{array}{c}\Phi = ([\Psi_1,\ldots,\Psi_n],[(\mathbb{A}_1,\mathbb{G}_1),\ldots,(\mathbb{A}_n,\mathbb{G}_n)]) \quad \mathtt{NI}([(\mathbb{A}_1,\mathbb{G}_1),\ldots,(\mathbb{A}_n,\mathbb{G}_n)]) \\ \mathbb{H}_s \uplus \mathbb{H}_1 \uplus \cdots \uplus \mathbb{H}_n = \mathbb{H} \quad \Psi_k,\mathbb{A}_k,\mathbb{G}_k \vdash \{(\mathtt{a}_k,\nu_k)\}\,(\mathbb{H}_s,\mathbb{H}_k,\mathbb{T}_k,\mathbb{L}) \text{ for all } k\end{array}}{\Phi, [(\mathtt{a}_1,\nu_1),\ldots,(\mathtt{a}_n,\nu_n)] \vdash (\mathbb{H},[\mathbb{T}_1,\ldots,\mathbb{T}_n],\mathbb{L})}\ (\text{PROG})$$

$\boxed{\Psi,\mathbb{A},\mathbb{G} \vdash \{(\mathsf{a},\nu)\}\,(\mathbb{H}_s,\mathbb{H}_v,\mathbb{T},\mathbb{L})}$ ***(Well-formed thread)***

$$\frac{\mathtt{a}\ (\mathbb{H}_s,(\mathbb{R},k),\mathbb{L}) \quad \nu\ (\mathbb{H}_v,(\mathbb{R},k),\mathbb{L}|_k) \quad \Psi,\mathbb{A},\mathbb{G} \vdash \mathbb{C}:\Psi \quad \Psi,\mathbb{A},\mathbb{G} \vdash \{(\mathsf{a},\nu)\}\,\mathbb{I}}{\Psi,\mathbb{A},\mathbb{G} \vdash \{(\mathsf{a},\nu)\}\,(\mathbb{H}_s,\mathbb{H}_v,(\mathbb{C},\mathbb{R},\mathbb{I},k),\mathbb{L})}\ (\text{THRD})$$

$\boxed{\Psi,\mathbb{A},\mathbb{G} \vdash \mathbb{C}:\Psi'}$ ***(Well-formed code heap)***

$$\frac{\forall \mathtt{f} \in dom(\Psi') : \quad \Psi,\mathbb{A},\mathbb{G} \vdash \{\Psi'(\mathtt{f})\}\,\mathbb{C}(\mathtt{f})}{\Psi,\mathbb{A},\mathbb{G} \vdash \mathbb{C}:\Psi'}\ (\text{CDHP})$$

$\boxed{\Psi,\mathbb{A},\mathbb{G} \vdash \{(\mathsf{a},\nu)\}\,\mathbb{I}}$ ***(Well-formed instr. sequences)***

$$\frac{\Psi,\mathbb{A},\mathbb{G} \vdash \{(\mathsf{a},\nu)\}\,\iota\,\{(\mathsf{a}',\nu')\} \quad \Psi,\mathbb{A},\mathbb{G} \vdash \{(\mathsf{a}',\nu')\}\,\mathbb{I} \quad (\mathtt{a}\circ\mathbb{A}) \Rightarrow \mathtt{a} \quad \mathsf{Precise}(\mathtt{a})}{\Psi,\mathbb{A},\mathbb{G} \vdash \{(\mathsf{a},\nu)\}\,\iota;\mathbb{I}}\ (\text{SEQ})$$

$$\frac{\begin{array}{r}\mathsf{Precise}(\mathtt{a}) \quad (\mathtt{a}\circ\mathbb{A}) \Rightarrow \mathtt{a} \quad \forall \mathbb{X}@(\mathbb{H},(\mathbb{R},k),\mathbb{L}).\ (\mathtt{a} * \nu)\ \mathbb{X} \rightarrow (\mathsf{a}' * \nu')\ \mathbb{X} \wedge (\lfloor\mathbb{G}\rfloor_{(\mathsf{a},\mathsf{a}')}\ \mathbb{X}\ \mathbb{X}) \\ \text{where } (\mathsf{a}',\nu') = \Psi(\mathbb{R}(\mathtt{r}_s))\end{array}}{\Psi,\mathbb{A},\mathbb{G} \vdash \{(\mathsf{a},\nu)\}\,\mathsf{jr}\ \mathtt{r}_s}\ (\text{JR})$$

$$\frac{\begin{array}{r}\mathsf{Precise}(\mathtt{a}) \quad (\mathtt{a}\circ\mathbb{A}) \Rightarrow \mathtt{a} \quad \forall \mathbb{X}.\ (\mathtt{a} * \nu)\ \mathbb{X} \rightarrow (\mathsf{a}' * \nu')\ \mathbb{X} \wedge (\lfloor\mathbb{G}\rfloor_{(\mathsf{a},\mathsf{a}')}\ \mathbb{X}\ \mathbb{X}) \\ \text{where } (\mathsf{a}',\nu') = \Psi(\mathtt{f})\end{array}}{\Psi,\mathbb{A},\mathbb{G} \vdash \{(\mathsf{a},\nu)\}\,\mathsf{j}\ \mathtt{f}}\ (\text{J})$$

Figure 7.7: SAGL Inference Rules

$\boxed{\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathsf{a}, \nu)\}\, \iota\, \{(\mathsf{a}', \nu')\}}$ **(*Well-formed instructions*)**

$$\frac{\begin{array}{r}\forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ (\mathsf{a} * \nu)\ \mathbb{X} \wedge l \notin \mathit{dom}(\mathbb{L}) \rightarrow (\mathsf{a}' * \nu')\ \mathbb{X}' \wedge (\lfloor \mathbb{G} \rfloor_{(\mathsf{a},\mathsf{a}')}\ \mathbb{X}\ \mathbb{X}') \\ \text{where } \mathbb{X}' = (\mathbb{H}, (\mathbb{R}, k), \mathbb{L}\{l \rightsquigarrow k\})\end{array}}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathsf{a}, \nu)\}\ \mathsf{lock}\ l\ \{(\mathsf{a}', \nu')\}} \ (\textsc{lock})$$

$$\frac{\begin{array}{r}\forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ (\mathsf{a} * \nu)\ \mathbb{X} \rightarrow \mathbb{L}(l) = k \wedge (\mathsf{a}' * \nu')\ \mathbb{X}' \wedge (\lfloor \mathbb{G} \rfloor_{(\mathsf{a},\mathsf{a}')}\ \mathbb{X}\ \mathbb{X}') \\ \text{where } \mathbb{X}' = (\mathbb{H}, (\mathbb{R}, k), \mathbb{L} \setminus \{l\})\end{array}}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathsf{a}, \nu)\}\ \mathsf{unlock}\ l\ \{(\mathsf{a}', \nu')\}} \ (\textsc{unlock})$$

$$\frac{\begin{array}{r}\forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\forall \mathtt{l}.\ (\mathsf{a} * \nu)\ \mathbb{X} \wedge \{\mathtt{l}, \ldots, \mathtt{l}{+}\mathbb{R}(\mathtt{r}_s){-}1\} \notin \mathit{dom}(\mathbb{H}) \rightarrow \\ \mathbb{R}(\mathtt{r}_s) > 0 \wedge (\mathsf{a}' * \nu')\ \mathbb{X}' \wedge (\lfloor \mathbb{G} \rfloor_{(\nu,\nu')})\ \mathbb{X}\ \mathbb{X}' \\ \text{where } \mathbb{X}' = (\mathbb{H}\{\mathtt{l}, \ldots, \mathtt{l}{+}\mathbb{R}(\mathtt{r}_s){-}1 \rightsquigarrow \_\}, (\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathtt{l}\}, k), \mathbb{L})\end{array}}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathsf{a}, \nu)\}\ \mathsf{alloc}\ \mathtt{r}_d, \mathtt{r}_s\ \{(\mathsf{a}', \nu')\}} \ (\textsc{alloc})$$

$$\frac{\begin{array}{l}\iota \in \{\mathsf{addu}, \mathsf{addiu}, \mathsf{subu}, \mathsf{lw}, \mathsf{sw}, \mathsf{free}\} \\ \forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ (\mathsf{a} * \nu)\ \mathbb{X} \\ \qquad \rightarrow \exists (\mathbb{H}', \mathbb{R}') = \mathsf{NextS}_\iota(\mathbb{H}, \mathbb{R}).\ (\mathsf{a}' * \nu')\ \mathbb{X}' \wedge (\lfloor \mathbb{G} \rfloor_{(\nu,\nu')})\ \mathbb{X}\ \mathbb{X}' \\ \qquad\qquad \text{where } \mathbb{X}' = (\mathbb{H}', (\mathbb{R}', k), \mathbb{L})\end{array}}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathsf{a}, \nu)\}\ \iota\ \{(\mathsf{a}', \nu')\}} \ (\textsc{simple})$$

$$\frac{\begin{array}{r}\forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ (\mathsf{a} * \nu)\ \mathbb{X} \rightarrow ((\mathbb{R}(\mathtt{r}_s) \leq 0 \rightarrow (\mathsf{a}' * \nu')\ \mathbb{X} \wedge (\lfloor \mathbb{G} \rfloor_{(\nu,\nu')})\ \mathbb{X}\ \mathbb{X}) \\ \wedge (\mathbb{R}(\mathtt{r}_s) > 0 \rightarrow (\mathsf{a}'' * \nu'')\ \mathbb{X} \wedge (\lfloor \mathbb{G} \rfloor_{(\nu,\nu'')})\ \mathbb{X}\ \mathbb{X})) \\ \text{where } (\mathsf{a}'', \nu'') = \Psi(\mathtt{f})\end{array}}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathsf{a}, \nu)\}\ \mathsf{bgtz}\ \mathtt{r}_s, \mathtt{f}\ \{(\mathsf{a}', \nu')\}} \ (\textsc{bgtz})$$

$$\frac{\begin{array}{r}\forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ (\mathsf{a} * \nu)\ \mathbb{X} \rightarrow ((\mathbb{R}(\mathtt{r}_s) \neq \mathbb{R}(\mathtt{r}_t) \rightarrow (\mathsf{a}' * \nu')\ \mathbb{X} \wedge (\lfloor \mathbb{G} \rfloor_{(\nu,\nu')})\ \mathbb{X}\ \mathbb{X}) \\ \wedge (\mathbb{R}(\mathtt{r}_s) = \mathbb{R}(\mathtt{r}_t) \rightarrow (\mathsf{a}'' * \nu'')\ \mathbb{X} \wedge (\lfloor \mathbb{G} \rfloor_{(\nu,\nu'')})\ \mathbb{X}\ \mathbb{X})) \\ \text{where } (\mathsf{a}'', \nu'') = \Psi(\mathtt{f})\end{array}}{\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathsf{a}, \nu)\}\ \mathsf{beq}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f}\ \{(\mathsf{a}', \nu')\}} \ (\textsc{beq})$$

Figure 7.8: SAGL Inference Rules (Cont'd)

here we require that there be a partition of memory into $n+1$ parts: one part $\mathbb{H}_s$ is shared and other parts $\mathbb{H}_1, \ldots, \mathbb{H}_n$ are privately owned by the threads $\mathbb{T}_1, \ldots, \mathbb{T}_n$, respectively. When we check the well-formedness of thread $\mathbb{T}_k$, the memory in the extended thread state is not the global memory. It just contains $\mathbb{H}_s$ and $\mathbb{H}_k$.

The THRD rule in SAGL is similar to the one in AGL, except that the memory visible by each thread is separated into two parts: the shared $\mathbb{H}_s$ and the private $\mathbb{H}_v$. We require that assertions a and ν hold over $\mathbb{H}_s$ and $\mathbb{H}_v$ respectively. Since ν only specifies the private resource, we use the "filter" operator $\mathbb{L}|_k$ to prevent ν from having access to the ownership information of locks not owned by the current thread:

$$(\mathbb{L}|_k)(l) \triangleq \begin{cases} k & \mathbb{L}(l) = k \\ \textit{undefined} & \text{otherwise} \end{cases} \tag{7.2}$$

i.e., $\mathbb{L}|_k$ is a subset of $\mathbb{L}$ which maps locks to k.

In the SEQ rule, we use (a, ν) as the precondition. However, to ensure that the precondition is preserved by state transitions satisfying $\mathbb{A}$, we only check a (*i.e.*, we check $(\mathsf{a} \circ \mathbb{A}) \Rightarrow \mathsf{a}$) because $\mathbb{A}$ only specifies shared resources. We know that the private resources will not be touched by the environment. We require a to be precise to enforce the unique boundary between shared and private resources. Following the definition in CSL [78], an assertion a is precise if and only if for any memory $\mathbb{H}$, there is at most one subset $\mathbb{H}'$ that satisfies a, *i.e.*,

$$\begin{aligned} \mathsf{Precise}(\mathsf{a}) \triangleq\ & \forall \mathbb{H}, \mathbb{R}, k, \mathbb{L}, \mathbb{H}_1, \mathbb{H}_2.\ (\mathbb{H}_1 \subseteq \mathbb{H}) \wedge (\mathbb{H}_2 \subseteq \mathbb{H}) \wedge \\ & \qquad \mathsf{a}\ (\mathbb{H}_1, (\mathbb{R}, k), \mathbb{L}) \wedge \mathsf{a}\ (\mathbb{H}_2, (\mathbb{R}, k), \mathbb{L}) \rightarrow \mathbb{H}_1 = \mathbb{H}_2\,. \end{aligned} \tag{7.3}$$

The JR rule requires a be precise and it be preserved by state transitions satisfying the assumption. Also, the specification assigned to the target address needs to be satisfied by the resulting state of the jump, and the identity state transition made by the jump satisfies the guarantee $\mathbb{G}$. We use the separating conjunction of the shared and private predicates as the pre- and post-condition. We define $\mathsf{a} * \nu$ as:

$$\begin{aligned} \mathsf{a} * \nu \;\triangleq\; & \lambda(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}). \\ & \quad \exists \mathbb{H}_1, \mathbb{H}_2.(\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{H}) \land \mathsf{a}\ (\mathbb{H}_1, (\mathbb{R}, k), \mathbb{L}) \land \nu\ (\mathbb{H}_2, (\mathbb{R}, k), \mathbb{L}|_k)\,. \end{aligned} \tag{7.4}$$

Again, the use of $\mathbb{L}|_k$ prevents ν from having access to the ownership information of locks not owned by the current thread. We use $f_1 \uplus f_2$ to represent the union of finite partial mappings with disjoint domains.

To ensure $\mathbb{G}$ is satisfied over shared resources, we lift $\mathbb{G}$ to $\lfloor \mathbb{G} \rfloor_{(\mathsf{a},\mathsf{a}')}$:

$$\begin{aligned} \lfloor \mathbb{G} \rfloor_{(\mathsf{a},\mathsf{a}')} \;\triangleq\; & \lambda \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}), \mathbb{X}'@(\mathbb{H}', (\mathbb{R}', k'), \mathbb{L}'). \\ & \quad \exists \mathbb{H}_1, \mathbb{H}_2, \mathbb{H}'_1, \mathbb{H}'_2.\ (\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{H}) \land (\mathbb{H}'_1 \uplus \mathbb{H}'_2 = \mathbb{H}') \\ & \qquad \land \mathsf{a}\ (\mathbb{H}_1, (\mathbb{R}, k), \mathbb{L}) \land \mathsf{a}'\ (\mathbb{H}'_1, (\mathbb{R}', k'), \mathbb{L}') \\ & \qquad \land \mathbb{G}\ (\mathbb{H}_1, (\mathbb{R}, k), \mathbb{L})\ (\mathbb{H}'_1, (\mathbb{R}', k'), \mathbb{L}')\,, \end{aligned} \tag{7.5}$$

Here we use precise predicates a and a' to enforce the unique boundary between shared and private resources.

As expected, the SAGL rule for each individual instruction is almost the same as its counterpart in AGL, except that we always use the separating conjunction of predicates for shared and private resources. Each instruction rule requires that memory in states before and after the transition can be partitioned to private and shared; private parts satisfy private predicates and shared parts satisfy shared predicates and $\mathbb{G}$.

It is important that we always combine shared predicates with private predicates instead of checking separately the relationship between a and a' and between ν and ν'. This gives us the ability to support *dynamic redistribution* of private and shared memory. Instead of enforcing static partition, we allow that part of private memory becomes shared under certain conditions and vice versa. As we will show in the next section, this ability makes our SAGL very expressive and is the enabling feature that makes the embedding of CSL into SAGL possible. This also explains why we require $\lfloor \mathbb{G} \rfloor_{(\mathtt{a},\mathtt{a}')}$ holds over state transitions made by every instruction: even though the instruction does not change $\mathbb{H}$ and $\mathbb{L}$, we may change the partition between private and shared memory at the end of this instruction, which would affect other threads and thus this change must satisfy $\lfloor \mathbb{G} \rfloor_{(\mathtt{a},\mathtt{a}')}$.

AGL can be viewed as a specialized version of SAGL where all the ν's are set to emp (emp is an assertion which can only be satisfied by memory with empty domain).

Soundness. Soundness of SAGL is proved following the syntactic approach [99] to proving soundness of type systems.

Lemma 7.4 If $\Psi, \mathbb{A}, \mathbb{G} \vdash \mathbb{C} : \Psi$, then $dom(\Psi) \subseteq dom(\mathbb{C})$.

Lemma 7.5 (Thread-Progress) If $\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathtt{a}, \nu)\}\ (\mathbb{H}_s, \mathbb{H}_v, \mathbb{T}_k, \mathbb{L})$, $dom(\mathbb{H}_s) \cap dom(\mathbb{H}_v) = \emptyset$, and $\mathbb{T}_k = (\mathbb{C}, \mathbb{R}, \mathbb{I}, k)$, then there exist $\mathbb{H}'$, $\mathbb{R}'$, $\mathbb{I}'$ and $\mathbb{L}'$ such that $(\mathbb{H}_s \cup \mathbb{H}_v, \mathbb{T}_k, \mathbb{L}) \overset{\mathsf{t}}{\longmapsto} (\mathbb{H}', \mathbb{T}'_k, \mathbb{L}')$, where $\mathbb{T}'_k = (\mathbb{C}, \mathbb{R}', \mathbb{I}', k)$.

Proof sketch: From $\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathtt{a}, \nu)\}\ (\mathbb{H}_s, \mathbb{H}_v, \mathbb{T}_k, \mathbb{L})$ and $dom(\mathbb{H}_s) \cap dom(\mathbb{H}_v) = \emptyset$, we know (1) $\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathtt{a}, \nu)\}\, \mathbb{I}$; (2) $\Psi, \mathbb{A}, \mathbb{G} \vdash \mathbb{C} : \Psi$; and (3) $(\mathtt{a} * \nu)\ (\mathbb{H}_s \cup \mathbb{H}_v, (\mathbb{R}, k), \mathbb{L})$. If $\mathbb{I} = \mathsf{j}\ \mathtt{f}$ ($\mathbb{I} = \mathsf{jr}\ \mathtt{r}_s$), by (1) and the J (JR) rule, we know $\mathtt{f} \in dom(\Psi)$ ($\mathbb{R}(\mathtt{r}_s) \in dom(\Psi)$). By Lemma 7.4, we know the target address is a valid code label in $\mathbb{C}$, therefore it is safe to execute the jump instruction. If $\mathbb{I} = \iota; \mathbb{I}'$, we know by (1) that there are $\mathtt{a}'$ and ν' such that $\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathtt{a}, \nu)\}\, \iota\, \{(\mathtt{a}', \nu')\}$. By inspection of instruction rules, we know it is always safe to execute the instruction ι as long as the state satisfy $\mathtt{a} * \nu$. Since we have (3), the thread can progress by executing ι. $\square$

Lemma 7.6 (Thread-Progress-Monotone) If $(\mathbb{H}, \mathbb{T}_k, \mathbb{L}) \overset{\mathsf{t}}{\longmapsto} (\mathbb{H}', \mathbb{T}'_k, \mathbb{L}')$, where $\mathbb{T}_k = (\mathbb{C}, \mathbb{R}, \mathbb{I}, k)$ and $\mathbb{T}'_k = (\mathbb{C}, \mathbb{R}', \mathbb{I}', k)$, then, for any $\mathbb{H}_0$ such that $dom(\mathbb{H}) \cap dom(\mathbb{H})_0 = \emptyset$, there exists $\mathbb{H}''$ and $\mathbb{R}''$ such that $(\mathbb{H} \cup \mathbb{H}_0, \mathbb{T}_k, \mathbb{L}) \overset{\mathsf{t}}{\longmapsto} (\mathbb{H}'', \mathbb{T}''_k, \mathbb{L}')$, where $\mathbb{T}''_k = (\mathbb{C}, \mathbb{R}'', \mathbb{I}', k)$.

Proof sketch: The proof trivially follows the definition of the operational semantics. Note that we model memory $\mathbb{H}$ as a *finite* partial mapping, so the monotonicity also holds for alloc. $\square$

Lemma 7.7 (SAGL-Progress) For any program $\mathbb{P} = (\mathbb{H}, [\mathbb{T}_1, \ldots, \mathbb{T}_n], \mathbb{L})$, if $\Phi, [(\mathsf{a}_1, \nu_1), \ldots, (\mathsf{a}_n, \nu_n)] \vdash \mathbb{P}$, then, for any thread $\mathbb{T}_k$, there exist $\mathbb{H}'$, $\mathbb{T}'_k$ and $\mathbb{L}'$ such that $(\mathbb{H}, \mathbb{T}_k, \mathbb{L}) \overset{\mathrm{t}}{\longmapsto} (\mathbb{H}', \mathbb{T}'_k, \mathbb{L}')$.

Proof sketch: By inversion of the PROG rule, Lemma 7.4 and Lemma 7.6. $\square$

Lemma 7.8 If $\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathsf{a}, \nu)\}\ (\mathbb{H}_s, \mathbb{H}_v, \mathbb{T}_k, \mathbb{L})$, then $(\mathsf{a} \circ \mathbb{A}) \Rightarrow \mathsf{a}$ and $\mathsf{Precise}(\mathsf{a})$.

Lemma 7.9 (Thread-Preservation) If $\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathsf{a}, \nu)\}\ (\mathbb{H}_s, \mathbb{H}_v, \mathbb{T}_k, \mathbb{L})$, $dom(\mathbb{H}_s) \cap dom(\mathbb{H}_v) = \emptyset$, and $(\mathbb{H}_s \cup \mathbb{H}_v, \mathbb{T}_k, \mathbb{L}) \overset{\mathrm{t}}{\longmapsto} (\mathbb{H}', \mathbb{T}'_k, \mathbb{L}')$, where $\mathbb{T}_k = (\mathbb{C}, \mathbb{R}, \mathbb{I}, k)$ and $\mathbb{T}'_k = (\mathbb{C}, \mathbb{R}', \mathbb{I}', k)$, then there exist a', ν', $\mathbb{H}'_s$ and $\mathbb{H}'_v$ such that $\mathbb{H}'_s \uplus \mathbb{H}'_v = \mathbb{H}'$, $\mathbb{G}\ (\mathbb{H}_s, (\mathbb{R}, k), \mathbb{L})\ (\mathbb{H}'_s, (\mathbb{R}', k), \mathbb{L}')$, and $\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathsf{a}', \nu')\}\ (\mathbb{H}'_s, \mathbb{H}'_v, \mathbb{T}'_k, \mathbb{L}')$.

Proof sketch: By $\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathsf{a}, \nu)\}\ (\mathbb{H}_s, \mathbb{H}_v, \mathbb{T}_k, \mathbb{L})$ and inversion of the THRD rule, we know $\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathsf{a}, \nu)\}\ \mathbb{I}$. Then we prove the lemma by inspection of the structure of $\mathbb{I}$ and the inversion of the corresponding instruction sequence rules and instruction rules. $\square$

Lemma 7.10 (Thread-Inv) If $\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathsf{a}, \nu)\}\ (\mathbb{H}_s, \mathbb{H}_v, \mathbb{T}_k, \mathbb{L})$, $dom(\mathbb{H}_s) \cap dom(\mathbb{H}_v) = \emptyset$, and $(\mathbb{H}_s \cup \mathbb{H}_v, \mathbb{T}_k, \mathbb{L}) \overset{\mathrm{t}}{\longmapsto} (\mathbb{H}', \mathbb{T}'_k, \mathbb{L}')$, where $\mathbb{T}_k = (\mathbb{C}, \mathbb{R}, \mathbb{I}, k)$ and $\mathbb{T}'_k = (\mathbb{C}, \mathbb{R}', \mathbb{I}', k)$, then

- if $\mathbb{I} = \mathsf{j}\ \mathtt{f}$, then $(\mathsf{a}' * \nu')\ (\mathbb{H}', (\mathbb{R}', k), \mathbb{L}')$ holds, where $(\mathsf{a}', \nu') = \Psi(\mathtt{f})$;
- if $\mathbb{I} = \mathsf{jr}\ \mathtt{r}_s$, then $(\mathsf{a}' * \nu')\ (\mathbb{H}', (\mathbb{R}', k), \mathbb{L}')$ holds, where $(\mathsf{a}', \nu') = \Psi(\mathbb{R}(\mathtt{r}_s))$;
- if $\mathbb{I} = \mathsf{beq}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f}; \mathbb{I}'$ and $\mathbb{R}(\mathtt{r}_s) = \mathbb{R}(\mathtt{r}_t)$, then $(\mathsf{a}' * \nu')\ (\mathbb{H}', (\mathbb{R}', k), \mathbb{L}')$ holds, where $(\mathsf{a}', \nu') = \Psi(\mathtt{f})$;
- if $\mathbb{I} = \mathsf{bgt}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f}; \mathbb{I}'$ and $\mathbb{R}(\mathtt{r}_s) > \mathbb{R}(\mathtt{r}_t)$, then $(\mathsf{a}' * \nu')\ (\mathbb{H}', (\mathbb{R}', k), \mathbb{L}')$ holds, where $(\mathsf{a}', \nu') = \Psi(\mathtt{f})$;

Proof sketch: By $\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathtt{a}, \nu)\}\, (\mathbb{H}_s, \mathbb{H}_v, \mathbb{T}_k, \mathbb{L})$ and inversion of the THRD rule, we know $\Psi, \mathbb{A}, \mathbb{G} \vdash \{(\mathtt{a}, \nu)\}\, \mathbb{I}$. The lemma trivially follows inversion of the J rule, the JR rule, the BEQ rule and the BGT rule. (The J rule is similar to the JR rule. The BEQ rule and the BGT rule are similar to their counterparts in AGL. These rules are not shown in Figure 7.7.) $\Box$

Lemma 7.11 (Thread-Frame) If $(\mathbb{H}, \mathbb{T}_k, \mathbb{L}) \xmapsto{\mathrm{t}} (\mathbb{H}', \mathbb{T}'_k, \mathbb{L}')$, $\mathbb{H} = \mathbb{H}_1 \uplus \mathbb{H}_2$, and there exists $\mathbb{H}''_1$ and $\mathbb{T}''_k$ such that $(\mathbb{H}_1, \mathbb{T}_k, \mathbb{L}) \xmapsto{\mathrm{t}} (\mathbb{H}''_1, \mathbb{T}''_k, \mathbb{L}')$, then there exists $\mathbb{H}'_1$ such that $\mathbb{H}' = \mathbb{H}'_1 \uplus \mathbb{H}_2$, and $(\mathbb{H}_1, \mathbb{T}_k, \mathbb{L}) \xmapsto{\mathrm{t}} (\mathbb{H}'_1, \mathbb{T}'_k, \mathbb{L}')$.

Proof sketch: By inspection of the operational semantics of instructions. $\Box$

Lemma 7.12 (SAGL-Preservation) If $\Phi, [(\mathtt{a}_1, \nu_1) \ldots, (\mathtt{a}_n, \nu_n)] \vdash \mathbb{P}$ and $(\mathbb{P} \longmapsto \mathbb{P}')$, where $\Phi = ([\Psi_1, \ldots, \Psi_n], [(\mathbb{A}_1, \mathbb{G}_1), \ldots, (\mathbb{A}_n, \mathbb{G}_n)])$, then there exist $\mathtt{a}'_1$, ν'_1, ..., $\mathtt{a}'_n$, ν'_n such that $\Phi, [(\mathtt{a}'_1, \nu'_1) \ldots, (\mathtt{a}'_n, \nu'_n)] \vdash \mathbb{P}'$.

Proof sketch: If the thread k executes its instruction in the step $\mathbb{P} \longmapsto \mathbb{P}'$, we know:

- the private memory of other threads will not be touched, following Lemma 7.11; and
- other threads' assertions $\mathtt{a}_i$ for shared resources are preserved, following Lemma 7.8, Lemma 7.9 and the non-interference of specifications, *i.e.*, $\mathtt{NI}([(\mathbb{A}_1, \mathbb{G}_1), \ldots, (\mathbb{A}_n, \mathbb{G}_n)])$.

Then, by the PROG rule, it is easy to prove $\Phi, [(\mathtt{a}'_1, \nu'_1) \ldots, (\mathtt{a}'_n, \nu'_n)] \vdash \mathbb{P}'$ for some $\mathtt{a}'_1$, ν'_1, ..., $\mathtt{a}'_n$, ν'_n. $\Box$

Finally, the soundness of SAGL is formulated in Theorem 7.13. In addition to the safety of well-formed programs, it also characterizes partial correctness: assertions assigned to labels in Ψ will hold whenever the labels are reached.

Theorem 7.13 (SAGL-Soundness) For any program $\mathbb{P}$ with specification $\Phi = ([\Psi_1, \ldots, \Psi_n], [(\mathbb{A}_1, \mathbb{G}_1), \ldots, (\mathbb{A}_n, \mathbb{G}_n)])$, if $\Phi, [(\mathtt{a}_1, \nu_1) \ldots, (\mathtt{a}_n, \nu_n)] \vdash \mathbb{P}$, then,

- for any natural number m, there exists $\mathbb{P}'$ such that $(\mathbb{P} \longmapsto^m \mathbb{P}')$;

$$\begin{array}{rrcl}
(\mathit{ProgSpec}) & \phi & ::= & ([\psi_1,\ldots,\psi_n],\Gamma)\\
(\mathit{CHSpec}) & \psi & ::= & \{\mathtt{f} \leadsto \nu\}^*\\
(\mathit{ResourceINV}) & \Gamma & \in & \mathit{Locks} \rightharpoonup \mathit{MemPred}\\
(\mathit{MemPred}) & \mathtt{m} & \in & \mathit{Memory} \to \mathsf{Prop}
\end{array}$$

Figure 7.9: Specification Constructs for CSL

- for any m and $\mathbb{P}' = (\mathbb{H}', [\mathbb{T}'_1,\ldots,\mathbb{T}'_n], \mathbb{L}')$, if $(\mathbb{P} \longmapsto^m \mathbb{P}')$, then,
  - $\Phi, [(\mathtt{a}'_1,\nu'_1),\ldots,(\mathtt{a}'_n,\nu'_n)] \vdash \mathbb{P}'$ for some $\mathtt{a}'_1,\ldots,\mathtt{a}'_n$ and $\nu'_1,\ldots,\nu'_n$;
  - for any k, there exist $\mathbb{H}''$, $\mathbb{T}''_k$ and $\mathbb{L}''$ such that $(\mathbb{H}', \mathbb{T}'_k, \mathbb{L}') \overset{\mathtt{t}}{\longmapsto} (\mathbb{H}'', \mathbb{T}''_k, \mathbb{L}'')$;
  - for any k, if $\mathbb{T}'_k = (\mathbb{C}_k, \mathbb{R}'_k, \mathsf{j}\ \mathtt{f}, k)$, then $(\mathtt{a}''_k * \nu''_k * \mathsf{TRUE})\ (\mathbb{H}', (\mathbb{R}'_k, k), \mathbb{L}')$ holds, where $(\mathtt{a}''_k, \nu''_k) = \Psi_k(\mathtt{f})$;
  - for any k, if $\mathbb{T}'_k = (\mathbb{C}_k, \mathbb{R}'_k, \mathsf{jr}\ \mathtt{r}_s, k)$, then $(\mathtt{a}''_k * \nu''_k * \mathsf{TRUE})\ (\mathbb{H}', (\mathbb{R}'_k, k), \mathbb{L}')$ holds, where $(\mathtt{a}''_k, \nu''_k) = \Psi_k(\mathbb{R}'_k(\mathtt{r}_s))$;
  - for any k, if $\mathbb{T}'_k = (\mathbb{C}_k, \mathbb{R}'_k, \mathsf{beq}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f}; \mathbb{I}, k)$ and $\mathbb{R}'_k(\mathtt{r}_s) = \mathbb{R}'_k(\mathtt{r}_t)$, then $(\mathtt{a}''_k * \nu''_k * \mathsf{TRUE})\ (\mathbb{H}', (\mathbb{R}'_k, k), \mathbb{L}')$ holds, where $(\mathtt{a}''_k, \nu''_k) = \Psi_k(\mathtt{f})$;
  - for any k, if $\mathbb{T}'_k = (\mathbb{C}_k, \mathbb{R}'_k, \mathsf{bgtz}\ \mathtt{r}_s, \mathtt{f}; \mathbb{I}, k)$ and $\mathbb{R}'_k(\mathtt{r}_s) > 0$, then $(\mathtt{a}''_k * \nu''_k * \mathsf{TRUE})\ (\mathbb{H}', (\mathbb{R}'_k, k), \mathbb{L}')$ holds, where $(\mathtt{a}''_k, \nu''_k) = \Psi_k(\mathtt{f})$;

Proof sketch: By Lemma 7.7, 7.12 and 7.10. □

7.4 Concurrent Separation Logic (CSL)

Both AGL and SAGL treat lock/unlock primitives as normal instructions. They do not require that shared memory be protected by locks. This shows the generality of the R-G method, which makes no assumption about language constructs for synchronizations. Any ad-hoc synchronizations can be verified using the R-G method.

If we focus on a special class of programs following Hoare [51] where accesses of shared resources are protected by critical regions (implemented by locks in our language), we can further simplify our SAGL logic and derive a variation of CSL (CSL adapted to our assembly language).

7.4.1 CSL Specifications and Rules

In CSL, shared memory is partitioned and each part is protected by a unique lock. For each part of the partition, an invariant is assigned to specify its well-formedness. A thread cannot access shared memory unless it has acquired the corresponding lock. After the lock is acquired, the thread takes advantage of mutual-exclusion provided by locks and treats the part of memory as private. When the thread releases the lock, it must ensure that the part of memory is well-formed with regard to the corresponding invariant. In this way the following global invariant is enforced:

Shared resources are well-formed outside critical regions.

Figure 7.9 shows the specification constructs for CSL. The program specification ϕ contains a collection of code heap specifications for each thread and the specification Γ for lock-protected memory. Code heap specification ψ maps a code label to an assertion ν as the precondition of the corresponding instruction sequence. Here ν plays similar role of the private predicate in SAGL. Since each thread privately owns the lock protected memory if it owns the lock, all memory accessible by a thread is viewed as private memory. Therefore we do not need an assertion $\mathtt{a}$ to specify the shared memory as we did in SAGL. This also explains why we do not need assumptions and guarantees in CSL. The specification Γ of lock-protected memory maps a lock to an invariant $\mathtt{m}$, which specifies the corresponding part of memory. The invariant $\mathtt{m}$ is simply a predicate over memory because the register file is private to each thread.

Inference rules. The inference rules for CSL are presented in Figure 7.11. The PROG rule requires that there be a partition of the global memory into $n+1$ parts. Each $\mathbb{H}_k$ is privately owned by thread $\mathbb{T}_k$. The well-formedness of $\mathbb{T}_k$ is checked by applying the THRD rule. $\mathbb{H}_s$

$$\mathtt{m} * \mathtt{m}' \triangleq \lambda\mathbb{H}.\exists\mathbb{H}_1, \mathbb{H}_2.\ (\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{H}) \wedge \mathtt{m}\ \mathbb{H}_1 \wedge \mathtt{m}'\ \mathbb{H}_2$$
$$\nu * \mathtt{m} \triangleq \lambda\mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\exists\mathbb{H}_1, \mathbb{H}_2.\ (\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{H}) \wedge \nu\ (\mathbb{H}_1, (\mathbb{R}, k), \mathbb{L}) \wedge \mathtt{m}\ \mathbb{H}_2$$
$$\forall_* x \in S.\ P(x) \triangleq \begin{cases} \mathsf{emp} & \text{if } S = \emptyset \\ P(x_i) * \forall_* x \in S'.\ P(x) & \text{if } S = S' \uplus \{x_i\} \end{cases}$$

$$\mathsf{Precise}(\mathtt{m}) \triangleq \forall\mathbb{H}, \mathbb{H}_1, \mathbb{H}_2.\ \mathbb{H}_1 \subseteq \mathbb{H} \wedge \mathbb{H}_2 \subseteq \mathbb{H} \wedge \mathtt{m}\ \mathbb{H}_1 \wedge \mathtt{m}\ \mathbb{H}_2 \rightarrow \mathbb{H}_1 = \mathbb{H}_2$$
$$\mathsf{Precise}(\Gamma) \triangleq \forall l \in dom(\Gamma).\ \mathsf{Precise}(\Gamma(l))$$
$$\nu \Rightarrow \nu' \triangleq \forall\mathbb{X}.\ \nu\ \mathbb{X} \rightarrow \nu'\ \mathbb{X}$$
$$\mathsf{acq}\ l\ \nu \triangleq \lambda(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ \nu\ (\mathbb{H}, (\mathbb{R}, k), \mathbb{L}\{l \rightsquigarrow k\})$$
$$\mathsf{rel}\ l\ \nu \triangleq \lambda(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ \mathbb{L}(l) = k \wedge \nu\ (\mathbb{H}, (\mathbb{R}, k), \mathbb{L} \setminus \{l\})$$
$$\nu \circ \mathsf{NextS}_\iota \triangleq \lambda(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ \exists(\mathbb{H}', \mathbb{R}') = \mathsf{NextS}_\iota(\mathbb{H}, \mathbb{R}).\ \nu\ (\mathbb{H}', (\mathbb{R}', k), \mathbb{L})$$

Figure 7.10: Definitions of Notations in CSL

$\boxed{\phi, [\nu_1, \ldots, \nu_n] \vdash \mathbb{P}}$ ***(Well-formed program)***

$$\frac{\begin{array}{cc} \phi = ([\psi_1, \ldots, \psi_n], \Gamma) & \mathbb{H}_s \uplus \mathbb{H}_1 \uplus \cdots \uplus \mathbb{H}_n = \mathbb{H} \\ \mathsf{a}_\Gamma\ (\mathbb{H}_s, \_, \mathbb{L}) \quad \mathsf{Precise}(\Gamma) & \psi_k, \Gamma \vdash \{\nu_k\}\ (\mathbb{H}_k, \mathbb{T}_k, \mathbb{L}) \ \text{for all } k \end{array}}{\phi, [\nu_1, \ldots, \nu_n] \vdash (\mathbb{H}, [\mathbb{T}_1, \ldots, \mathbb{T}_n], \mathbb{L})} \ (\textsc{prog})$$

$\boxed{\psi, \Gamma \vdash \{\nu\}\ (\mathbb{H}, \mathbb{T}, \mathbb{L})}$ ***(Well-formed thread)***

$$\frac{\nu\ (\mathbb{H}, (\mathbb{R}, k), \mathbb{L}|_k) \quad \psi, \Gamma \vdash \mathbb{C} : \psi \quad \psi, \Gamma \vdash \{\nu\}\ \mathbb{I}}{\psi, \Gamma \vdash \{\nu\}\ (\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}, k), \mathbb{L})} \ (\textsc{thrd})$$

$\boxed{\psi, \Gamma \vdash \mathbb{C} : \psi'}$ ***(Well-formed code heap)***

$$\frac{\forall \mathtt{f} \in dom(\psi') : \quad \psi, \Gamma \vdash \{\psi'(\mathtt{f})\}\ \mathbb{C}(\mathtt{f})}{\psi, \Gamma \vdash \mathbb{C} : \psi'} \ (\textsc{cdhp})$$

$\boxed{\psi, \Gamma \vdash \{\nu\}\ \mathbb{I}}$ ***(Well-formed instr. sequences)***

$$\frac{\psi, \Gamma \vdash \{\nu\}\ \iota\ \{\nu'\} \quad \psi, \Gamma \vdash \{\nu'\}\ \mathbb{I}}{\psi, \Gamma \vdash \{\nu\}\ \iota; \mathbb{I}} \ (\textsc{seq})$$

$$\frac{\forall\mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ \nu\ \mathbb{X} \rightarrow \psi(\mathbb{R}(\mathtt{r}_s))\ \mathbb{X}}{\psi, \Gamma \vdash \{\nu\}\ \mathsf{jr}\ \mathtt{r}_s} \ (\textsc{jr})$$

$$\frac{\forall\mathbb{X}.\ \nu\ \mathbb{X} \rightarrow \psi(\mathtt{f})\ \mathbb{X}}{\psi, \Gamma \vdash \{\nu\}\ \mathsf{j}\ \mathtt{f}} \ (\textsc{j})$$

Figure 7.11: CSL Inference Rules

$\boxed{\psi, \Gamma \vdash \{\nu\}\, \iota\, \{\nu'\}}$ **(*Well-formed instructions*)**

$$\frac{\nu * \mathtt{m} \Rightarrow \mathsf{acq}\ l\ \nu'}{\psi, \Gamma\{l \rightsquigarrow \mathtt{m}\} \vdash \{\nu\}\, \mathsf{lock}\ l\, \{\nu'\}}\ (\textsc{lock}) \qquad \frac{\nu \Rightarrow (\mathsf{rel}\ l\ \nu') * \mathtt{m}}{\psi, \Gamma\{l \rightsquigarrow \mathtt{m}\} \vdash \{\nu\}\, \mathsf{unlock}\ l\, \{\nu'\}}\ (\textsc{unlock})$$

$$\frac{\begin{array}{l}\forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\forall \mathtt{l}.\\ \quad \nu\ \mathbb{X} \wedge (\{\mathtt{l}, \ldots, \mathtt{l}{+}\mathbb{R}(\mathtt{r}_s){-}1\} \not\in dom(\mathbb{H})) \to \mathbb{R}(\mathtt{r}_s) > 0 \wedge \nu'\ \mathbb{X}'\\ \qquad\qquad \text{where } \mathbb{X}' = (\mathbb{H}\{\mathtt{l} \rightsquigarrow \_, \ldots, \mathtt{l}{+}\mathbb{R}(\mathtt{r}_s){-}1 \rightsquigarrow \_\}, (\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathtt{l}\}, k), \mathbb{L})\end{array}}{\psi, \Gamma \vdash \{\nu\}\, \mathsf{alloc}\ \mathtt{r}_d, \mathtt{r}_s\, \{\nu'\}}\ (\textsc{alloc})$$

$$\frac{\nu \Rightarrow \nu' \circ \mathsf{NextS}_\iota \qquad \iota \in \{\mathsf{addu}, \mathsf{addiu}, \mathsf{subu}, \mathsf{lw}, \mathsf{sw}, \mathsf{free}\}}{\psi, \Gamma \vdash \{\nu\}\, \iota\, \{\nu'\}}\ (\textsc{simple})$$

$$\frac{\forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ \nu\ \mathbb{X} \to ((\mathbb{R}(\mathtt{r}_s) \le 0 \to \nu'\ \mathbb{X}) \wedge (\mathbb{R}(\mathtt{r}_s) > 0 \to \psi(\mathtt{f})\ \mathbb{X}))}{\psi, \Gamma \vdash \{\nu\}\, \mathsf{bgtz}\ \mathtt{r}_s, \mathtt{f}\, \{\nu'\}}\ (\textsc{bgtz})$$

$$\frac{\forall \mathbb{X}@(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ \nu\ \mathbb{X} \to ((\mathbb{R}(\mathtt{r}_s) \neq \mathbb{R}(\mathtt{r}_t) \to \nu'\ \mathbb{X}) \wedge (\mathbb{R}(\mathtt{r}_s) = \mathbb{R}(\mathtt{r}_t) \to \psi(\mathtt{f})\ \mathbb{X}))}{\psi, \Gamma \vdash \{\nu\}\, \mathsf{beq}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f}\, \{\nu'\}}\ (\textsc{beq})$$

Figure 7.12: CSL Inference Rules (Cont'd)

is the part of memory protected by free locks (locks not owned by any threads). It must satisfy the invariants specified in Γ. Here a_Γ is the separating conjunction of invariants assigned to free locks in Γ, which is defined as:

$$\mathsf{a}_\Gamma \triangleq \lambda(\mathbb{H}, (\mathbb{R}, k), \mathbb{L}).\ (\forall_* l \in (dom(\Gamma) - dom(\mathbb{L})).\ \Gamma(l))\ \mathbb{H}\,, \tag{7.6}$$

that is, *shared resources are well-formed outside of critical regions*. Here $\forall_*$ is an indexed, finitely iterated separating conjunction, which is formalized in Figure 7.10. Separating conjunctions with memory predicates ($\nu * \mathtt{m}$ and $\mathtt{m} * \mathtt{m}'$) are also defined in Figure 7.10. As in O'Hearn's original work on CSL [78], we also require invariants specified in Γ to be precise (*i.e.,* $\mathsf{Precise}(\Gamma)$, as defined in Figure 7.10), therefore we know a_Γ is also precise.

The THRD rule checks the well-formedness of threads. It requires that the current extended thread state satisfies the precondition ν. Since ν only cares about the resource privately owned by the thread, it takes $\mathbb{L}|_k$ instead of complete $\mathbb{L}$ as argument. Recall that

$$\frac{\psi, \Gamma \vdash \{\nu\}\, \mathbb{I}}{\psi * \mathtt{m}, \Gamma \uplus \Gamma' \vdash \{\nu * \mathtt{m}\}\, \mathbb{I}} \;(\textsc{frame-s}) \qquad \frac{\psi, \Gamma \vdash \{\nu\}\, \iota\, \{\nu'\}}{\psi * \mathtt{m}, \Gamma \uplus \Gamma' \vdash \{\nu * \mathtt{m}\}\, \iota\, \{\nu' * \mathtt{m}\}} \;(\textsc{frame-i})$$

$$\text{where } \psi * \mathtt{m} \triangleq \lambda \mathtt{f}.\, \psi(\mathtt{f}) * \mathtt{m} \quad \text{if } \mathtt{f} \in dom(\psi)$$

$$\frac{\psi, \Gamma \vdash \{\nu\}\, \mathbb{I} \qquad \psi, \Gamma \vdash \{\nu'\}\, \mathbb{I}}{\psi, \Gamma \vdash \{\nu \wedge \nu'\}\, \mathbb{I}} \;(\textsc{conj-s}) \qquad \frac{\psi, \Gamma \vdash \{\nu_1\}\, \iota\, \{\nu_1'\} \qquad \psi, \Gamma \vdash \{\nu_2\}\, \iota\, \{\nu_2'\}}{\psi, \Gamma \vdash \{\nu_1 \wedge \nu_2\}\, \iota\, \{\nu_1' \wedge \nu_2'\}} \;(\textsc{conj-i})$$

Figure 7.13: Admissible Rules for CSL

$\mathbb{L}|_k$ is defined in (7.2) in Section 7.3 to represent the subset of $\mathbb{L}$ which maps locks to k. The CDHP rule and rules for instruction sequences are similar to their counterparts in AGL and SAGL and require no more explanation.

In the LOCK rule, we use "acq l ν'" to represent the weakest precondition of ν'; and "$\nu \Rightarrow \nu'$" for logical implication lifted for state predicates. They are formalized in Figure 7.10. If the lock l instruction successfully acquires the lock l, we know by our global invariant that the part of memory protected by l satisfies the invariant $\Gamma(l)$ (*i.e.*, $\mathtt{m}$), because l is a free lock before lock l is executed. Therefore, we can carry the knowledge $\mathtt{m}$ in the postcondition ν'. Also, carrying $\mathtt{m}$ in ν' allows subsequent instructions to access that part of memory, since separation logic predicates capture ownerships of memory.

In the UNLOCK rule, "rel l ν'" is the weakest precondition for ν' (see Fig. 7.10). At the time the lock l is released, the memory protected by l must be well formed with respect to $\mathtt{m} = \Gamma(l)$. The separating conjunction here ensures that ν' does not specify this part of memory. Therefore the following instructions cannot use the part of memory unless the lock is acquired again. Rules for other instructions are straightforward and are not shown here.

Figure 7.13 shows admissible rules for CSL, including the frame rules and conjunction rules, which can be proved as lemmas in our meta-logic based on the rules shown in Figure 7.11. The frame rules (the FRAME-S rule and the FRAME-I rule) are very similar to the hypothetical frame rules presented in [80]. Interested readers can refer to previous papers on separation logic for their meanings.

7.4.2 Interpretation of CSL in SAGL

We prove the soundness of CSL by giving it an interpretation in SAGL, and proving CSL rules as derivable lemmas. This interpretation also formalizes the specialization made for CSL to achieve the simplicity.

From SAGL's point of view, each thread has two parts of memory: the private and the shared. In CSL, the private memory of a thread includes the memory protected by locks held by the thread and the memory that will never be shared. The shared memory are the parts protected by free locks. Therefore, we can use the following interpretation to translate a CSL specification to a SAGL specification:

$$[\![\, \nu \,]\!]_\Gamma \triangleq (\mathtt{a}_\Gamma, \nu) \tag{7.7}$$

$$[\![\, \psi \,]\!]_\Gamma \triangleq \lambda \mathtt{f}.[\![\, \psi(\mathtt{f}) \,]\!]_\Gamma \;\text{ if } \mathtt{f} \in dom(\psi)\,, \tag{7.8}$$

where $\mathtt{a}_\Gamma$ formalizes the CSL invariant and is defined by (7.6). We just reuse CSL specification ν as the specification of private memory, and use the separating conjunction $\mathtt{a}_\Gamma$ of invariants assigned to free locks as the specification for shared memory.

Since the assumption and guarantee in SAGL only specifies shared memory, we can define $\mathbb{A}_\Gamma$ and $\mathbb{G}_\Gamma$ for CSL threads:

$$\mathbb{A}_\Gamma \triangleq \lambda \mathbb{X}@(\mathbb{H},(\mathbb{R},k),\mathbb{L}), \mathbb{X}'@(\mathbb{H}',(\mathbb{R}',k'),\mathbb{L}').\mathbb{R}=\mathbb{R}' \wedge k=k' \wedge (\mathtt{a}_\Gamma\; \mathbb{X} \rightarrow \mathtt{a}_\Gamma\; \mathbb{X}') \tag{7.9}$$

$$\mathbb{G}_\Gamma \triangleq \lambda \mathbb{X}@(\mathbb{H},(\mathbb{R},k),\mathbb{L}), \mathbb{X}'@(\mathbb{H}',(\mathbb{R}',k'),\mathbb{L}').\; k=k' \wedge \mathtt{a}_\Gamma\; \mathbb{X} \wedge \mathtt{a}_\Gamma\; \mathbb{X}' \tag{7.10}$$

which enforces the invariant $\mathtt{a}_\Gamma$ of shared memory.

With above interpretations, we can prove the following soundness theorem.

Theorem 7.14 (CSL-Soundness) 1. If $\psi, \Gamma \vdash \{\nu\}\, \iota\, \{\nu'\}$ in CSL, then $[\![\, \psi \,]\!]_\Gamma, \mathbb{A}_\Gamma, \mathbb{G}_\Gamma \vdash \{[\![\, \nu \,]\!]_\Gamma\}\, \iota\, \{[\![\, \nu' \,]\!]_\Gamma\}$ in SAGL;

2. If $\psi, \Gamma \vdash \{\nu\}\, \mathbb{I}$ in CSL and $\mathsf{Precise}(\Gamma)$, then $[\![\, \psi \,]\!]_\Gamma, \mathbb{A}_\Gamma, \mathbb{G}_\Gamma \vdash \{[\![\, \nu \,]\!]_\Gamma\}\, \mathbb{I}$ in SAGL;

```
(1)     start:  -{(emp, emp)}
(2)             addiu r1, r0, 1                 ;; local int x, y;
(3)             alloc r2, r1                    ;; x := alloc(1);
                -{(emp, $r_2 \mapsto \_$)}
(4)             addiu r1, r0, $m$
(5)             sw    r1, 0(r2)                 ;; [x] := $m$;
                -{(emp, $(r_2 \mapsto m) \wedge r_1 = m$)}
(6)             lw    r3, 0(r2)                 ;; y := [x];
                -{(emp, $(r_2 \mapsto m) \wedge r_1 = m \wedge r_3 = m$)}
(7)             beq   r1, r3, safe              ;; while(y == $m$){}
(8)     unsafe: -{(emp, False)}
(9)             free  r0                        ;; free(0);  (* unsafe! *)
(10)    safe:   -{(emp, $r_2 \mapsto \_$)}
(11)            j     safe
```

Figure 7.14: Example 1: Memory Allocation

3. If $\psi, \Gamma \vdash \mathbb{C} : \psi'$ in CSL and $\mathsf{Precise}(\Gamma)$, then $[\![\,\psi\,]\!]_\Gamma, \mathbb{A}_\Gamma, \mathbb{G}_\Gamma \vdash \mathbb{C} : [\![\,\psi'\,]\!]_\Gamma$ in SAGL;

4. If $\psi, \Gamma \vdash \{\nu\}\ (\mathbb{H}_k, \mathbb{T}_k, \mathbb{L})$ in CSL, $\mathsf{Precise}(\Gamma)$, and $\mathsf{a}_\Gamma\ (\mathbb{H}_s, \_, \mathbb{L})$, then $[\![\,\psi\,]\!]_\Gamma, \mathbb{A}_\Gamma, \mathbb{G}_\Gamma \vdash \{[\![\,\nu\,]\!]_\Gamma\}\ (\mathbb{H}_s, \mathbb{H}_k, \mathbb{T}_k, \mathbb{L})$ in SAGL;

5. If $([\psi_1, \ldots, \psi_n], \Gamma), [\nu_1, \ldots, \nu_n] \vdash \mathbb{P}$ in CSL, then $\Phi, [[\![\,\nu_1\,]\!]_\Gamma, \ldots, [\![\,\nu_n\,]\!]_\Gamma] \vdash \mathbb{P}$ in SAGL, where $\Phi = ([\,[\![\,\psi_1\,]\!]_\Gamma, \ldots, [\![\,\psi_n\,]\!]_\Gamma], [(\mathbb{A}_\Gamma, \mathbb{G}_\Gamma), \ldots, (\mathbb{A}_\Gamma, \mathbb{G}_\Gamma)])$.

7.5 SAGL Examples

We use two complementary examples to demonstrate how SAGL combines merits of AGL and CSL. Figure 7.14 shows a simple program, which allocates a fresh memory cell and then writes into and reads from it. Following the MIPS convention, we assume the register $\mathtt{r}_0$ always contains 0. The corresponding high-level pseudo code is given as comments (followed by ";;"). It is obvious that two threads executing the same code (but may use different m) will never interfere with each other, therefore the test in line (7) is always TRUE and the program never reaches the unsafe branch.

It is trivial to certify the code in CSL since there is no memory-sharing at all. However, due to the nondeterministic operation of the alloc instruction, it is challenging to certify the code in AGL because the specification of $\mathbb{A}$ and $\mathbb{G}$ requires global knowledge of memory.

$$(\mathtt{m} \mapsto \alpha) * (\mathtt{n} \mapsto \beta)$$

```
local int x, y;                               local int x, y;
while(true){                                  while(true){
   x := [m];                                     x := [n];
   y := [n];                        ||           y := [m];
   if(x > y) {[m] := x-y;}                       if(x > y) {[n] := x-y;}
   if(x == y) { break;}                          if(x == y) { break;}
}                                             }
```

$$(\mathtt{m} \mapsto gcd(\alpha,\beta)) * (\mathtt{n} \mapsto gcd(\alpha,\beta))$$

Figure 7.15: Example 2: Parallel GCD

We certify the code in SAGL. Assertions are shown as annotations enclosed in "-{}". Recall that in SAGL the first assertion in the pair specifies shared resources and the second one specifies private resources. We treat all the resources as private, therefore the shared predicate is simply emp. The corresponding $\mathbb{A}$ and $\mathbb{G}$ are trivial. The whole verification is as simple as in CSL.

The second example is the GCD example shown in Chapter 5, which is a parallel implementation of the Euclidean algorithm to compute the greatest common divisor (GCD) of α and β, stored at locations m and n initially. Here we port it to the abstract machine with a preemptive thread model. The high-level pseudo code is shown in Figure 7.15. Memory cells at m and n are shared, but locks are not used for synchronization. Here we only show the code of the two collaborating threads and ignore thread creation, join and termination. To certify the code in CSL, we have to rewrite it by wrapping each memory-access command using "lock" and "unlock" commands and by introducing auxiliary variables. This time we use the "AGL part" of SAGL to certify the code. Figure 7.16 shows the assembly code of the first thread, with specifications as annotations. Private predicates are simply emp. The assumption and guarantee are defined below, where we use primed values (*e.g.*,

```
loop: -{($\exists x, y.\ (\mathtt{m} \mapsto x) * (\mathtt{n} \mapsto y) \wedge gcd(x,y) = gcd(\alpha,\beta)$, emp)}
      lw   r1, m(r0)                            ;; r1 <- [m]

      -{($\exists x, y.\ (\mathtt{m} \mapsto x) * (\mathtt{n} \mapsto y) \wedge gcd(x,y) = gcd(\alpha,\beta) \wedge \mathtt{r}_1 = x$, emp)}
      lw   r2, n(r0)                            ;; r2 <- [n]

      -{($\exists x, y.\ (\mathtt{m} \mapsto x) * (\mathtt{n} \mapsto y) \wedge gcd(x,y) = gcd(\alpha,\beta) \wedge \mathtt{r}_1 = x$
                              $\wedge \mathtt{r}_2 \geq y \wedge (x \geq y \rightarrow \mathtt{r}_2 = y)$, emp)}
      beq  r1, r2, done                         ;; if (r1 == r2) goto done
      subu r3, r1, r2                           ;; r3 = r1 - r2
      bgtz r1, calc                             ;; if (r1 > r2)  goto calc
      j    loop                                 ;; goto loop

calc: -{($\exists x, y.\ (\mathtt{m} \mapsto x) * (\mathtt{n} \mapsto y) \wedge gcd(x,y) = gcd(\alpha,\beta) \wedge (\mathtt{r}_3 = x - y) \wedge x > y$, emp)}
      sw   r3, m(r0)                            ;; [m] <- r3
      j    loop                                 ;; goto loop

done: -{($\exists x.\ (\mathtt{m} \mapsto x) * (\mathtt{n} \mapsto x) \wedge x = gcd(\alpha,\beta)$, emp)}
      j    done
```

Figure 7.16: Parallel GCD–Assembly Code for The First Thread

$[\mathtt{m}]'$ and $[\mathtt{n}]'$) to represent memory values in the resulting state of each action.

$$\mathbb{A}_1 \triangleq ([\mathtt{m}] = [\mathtt{m}]') \wedge ([\mathtt{n}] \geq [\mathtt{n}]') \wedge ([\mathtt{m}] \geq [\mathtt{n}] \rightarrow [\mathtt{n}] = [\mathtt{n}]') \wedge (gcd([\mathtt{m}],[\mathtt{n}]) = gcd([\mathtt{m}]',[\mathtt{n}]'))$$
$$\mathbb{G}_1 \triangleq ([\mathtt{n}] = [\mathtt{n}]') \wedge ([\mathtt{m}] \geq [\mathtt{m}]') \wedge ([\mathtt{n}] \geq [\mathtt{m}] \rightarrow [\mathtt{m}] = [\mathtt{m}]') \wedge (gcd([\mathtt{m}],[\mathtt{n}]) = gcd([\mathtt{m}]',[\mathtt{n}]'))$$

The example shown in Figure 7.17 is adapted from O'Hearn [79]. P-V primitives are firstly implemented using locks, and then they are used for synchronization. This example illustrates the support of redistribution of shared and private memory in SAGL.

7.6 Discussions and Summary

O'Hearn [78] and Brookes [14] proposed CSL for a high-level parallel language following Hoare [51]. Synchronization in the language is achieved by the conditional critical region (CCR) in the form of "with r when b do c". Semantics of CCRs is as follows: the statement c can be executed only if the resource r has not been acquired by others and the Boolean expression b is true; otherwise the thread will be blocked. We adapt CSL to an assembly

$$
\begin{array}{lcl}
\mathsf{I}(\mathtt{s}, x) & \triangleq & (\mathtt{s} \mapsto x) * ((x = 0 \wedge \mathsf{emp}) \vee (x = 1 \wedge 10 \mapsto)) \\
\mathsf{I}(\mathtt{s}) & \triangleq & \exists x. \mathsf{I}((\mathtt{s}, x)) \\
\Gamma & \triangleq & \{l_1 \leadsto \mathsf{I}(\mathtt{free}), l_2 \leadsto \mathsf{I}(\mathtt{busy})\}
\end{array}
$$

```
P_free: -{(a_Γ, emp)}                            ;;  while(true){
        lock   l_1                               ;;    lock $l_1$
        -{(a_Γ, $\mathsf{I}$(free))}
        lw     r1, free(r0)                      ;;
        -{(a_Γ, $\mathsf{I}$(free, r_1))}
        bgtz   r1, dec_P                         ;;    if([free]>0)  break;
        -{(a_Γ, $r_1 = 0 \wedge \mathsf{I}$(free, r_1))}
        unlock l_1                               ;;    unlock $l_1$
        -{(a_Γ, emp)}
        j      P_free                            ;;  }

dec_P:  -{(a_Γ, $r_1 = 1 \wedge \mathsf{I}$(free, r_1))}
        addiu  r2, r0, 1
        subu   r1, r1, r2                        ;;  [free] <- [free]-1
        -{(a_Γ, $r_1 = 0 \wedge \mathsf{I}$(free, 1))}
        st     r1, free(r0)
        -{(a_Γ, $(10 \mapsto \_) * \mathsf{I}$(free, 0))}
        -{(a_Γ, $(10 \mapsto \_) * \mathsf{I}$(free))}
        unlock l_1                               ;;   unlock $l_1$
        -{(a_Γ, $10 \mapsto \_$)}
        j      body

body:   -{(a_Γ, $10 \mapsto \_$)}
        addiu  r2, r0, $m$
        sw     r2, 10(r0)                        ;;  [10] <- $m$
        j      V_busy

V_busy: -{(a_Γ, $10 \mapsto \_$)}
        lock   l_2                               ;; lock $l_2$
        -{(a_Γ, $(10 \mapsto \_) * \mathsf{I}$(busy))}
        -{(a_Γ, $(10 \mapsto \_) * \mathsf{I}$(busy, 0))}
        lw     r1, busy(r0)
        addiu  r1, r0, 1                         ;;  [busy] <- [busy]+1
        sw     r1, busy(r0)
        -{(a_Γ, $\mathsf{I}$(busy, 1))}
        unlock l_2                               ;; unlock $l_2$
        -{(a_Γ, emp)}
        j      done

done:   -{(a_Γ, emp)}
        j      done
```

Figure 7.17: SAGL Example: Synchronizations Based on P-V Operations

language. The CCR can be implemented using our lock/unlock primitives. Each lock in our language corresponds to a resource name at the high-level. Atomic instructions in our assembly language are very similar to actions in Brookes Semantics [14], where semantic functions are defined for statements and expressions. These semantic functions can be viewed as a translation from the high-level language to a low-level language similar to ours. Recently, Reynolds [88] and Brookes [15] have studied grainless semantics for concurrency. Brookes also gives a grainless semantics to CSL [15].

The PROG rule of our CSL corresponds to O'Hearn's parallel composition rule [78]. The number of threads in our machine is fixed, therefore the nested parallel composition statement supported by Brookes [14] is not supported in our language. We studied verification of assembly code with dynamic thread creation in an earlier paper [34].

CSL is still evolving. Bornat *et al.* [13] proposed a refinement of CSL with fine-grained resource accounting. Parkinson *et al.* [82] applied CSL to verify a non-blocking implementation of stacks. As in the original CSL, these works also assume language constructs for synchronizations. We suspect that there exist reductions from these variations to SAGL-like logics. We leave this as our future work.

Concurrently with our work on SAGL, Vafeiadis and Parkinson [95] proposed another approach to combining rely/guarantee and separation logic, which we refer to here as RGSep. Both RGSep and SAGL partition memory into shared and private parts. However, shared memory cannot be accessed directly in RGSep. It has to be converted into private first to be accessed. Conversions can only occur at boundaries of critical regions, which is a built-in language construct required by RGSep to achieve atomicity. RGSep, in principle, does not assume smallest granularity of transitions. In SAGL, shared memory can be accessed directly, or be converted into private first and then accessed. Conversions can be made dynamically at any program point, instead of being coupled with critical regions. However, like A-G reasoning, SAGL assumes smallest granularity. We suspect that RGSep can be compiled into a specialized version of SAGL, following the way we translate CSL. On the other hand, if our instructions are wrapped using critical regions, SAGL might be derived from RGSep too.

We also use SAGL as the basis to formalize the relationship between CSL and R-G reasoning. We encode the CSL invariant as an assumption and guarantee in SAGL, and prove that CSL rules are derivable from corresponding SAGL rules with the specific assumption and guarantee.

Chapter 8

Conclusions and Future Work

System software consists of program modules that use many language features and span different abstraction levels. It is extremely difficult to design a verification system (*e.g.*, a type system or a program logic) to certify all the modules, just like we never use a single programming language to implement the whole system. We propose a new methodology in this thesis to solve this problem: we use the most appropriate verification system to certify individual program modules, and use a foundational open framework to support interoperability of different verification systems, so that certified modules can be linked in this common framework.

In this thesis we presented the OCAP framework, which is designed to satisfy the requirements over an open framework: extensibility, modularity and expressiveness. OCAP is not designed with a priori knowledge of foreign verification systems. It uses an extensible and heterogeneous program specification. Taking advantage of Coq's support of dependent types, specifications in foreign systems for modules can be easily incorporated as part of OCAP specifications. The heterogeneous program specification also allows OCAP to specify embedded code pointers, which enables OCAP's support for modularity. When one side of system code or client code is specified and certified in a foreign verification system, no knowledge about the other side is required. Modules certified in one verification system can be adapted in OCAP to interoperate with other modules in a different system without redoing the proof. The assertions used in OCAP inference rules are expressive enough to specify invariants enforced in most type systems and program logics, such as memory safety, well-formedness of stacks, non-interference between concurrent threads, *etc.*. The soundness of OCAP ensures that these invariants are maintained when foreign systems are embedded in the framework.

We also developed program logics to certify low-level code with different features. The SCAP logic can be applied to certify sequential assembly code with stack-based control abstractions. Instead of treating return pointers as first-class code pointers (which require "impredicative types" [68, 75]), SCAP specifies the invariant at each program point using a pair of a precondition and a guarantee (which states the obligation that the current function must fulfill before it can return or throw an exception). These guarantees, when chained together, are used to specify the well-formedness of a logical control stack. A function can also cut to any return pointer on the stack if it can establish the well-formedness of its control stack. We have shown that SCAP can support most stack-based control abstractions, including normal function call/return, tail call, exception handling based on stack cutting and stack unwinding [85], weak continuations in C-- [85], `setjmp/longjmp` in C [61], and multi-return function call [89].

The CMAP logic extends rely-guarantee method for shared-state concurrency verification [59] with unbounded dynamic thread creations. It unifies the concepts of a thread's assumption/guarantee and its environment's guarantee/assumption, and allows a thread to change its assumption/guarantee to track the composition of its dynamically changing environment.

We showed that SCAP and a variation of CMAP (CMAP$^-$) can be embedded into the OCAP framework. Recent work by Guo *et al.* [43] has shown that CMAP itself can be embedded into OCAP too. Applications of OCAP to support interoperation of verification systems are also interesting. In the first application, we showed how to link client code in TAL with a simple certified memory management library. TAL only supports weak-memory updates and the free memory is invisible to TAL code. The memory management library is specified in SCAP, which supports reasoning about operations over free memory and still ensures that the invariants of TAL code is maintained. In the second application, we showed how to construct foundational certified packages for concurrent code *without* trusting the scheduler. The user thread code is certified using the rely-guarantee method [59]; the thread scheduler is certified as sequential code in SCAP. They are linked in OCAP to construct FPCC packages. Another application of OCAP, shown by Lin *et al.* [64], links TAL code with a conservative garbage collector certified in SCAP.

This thesis also studied the relationship between rely-guarantee reasoning and concurrent separation logic, two widely used methodologies for modular verification of shared-state concurrent programs. The comparison is based on an abstract machine with preemptive threads. We also proposed the SAGL logic, which combines the merits of both sides. SAGL is as general as rely-guarantee reasoning, but has better support of modularity because of its application of the local-reasoning idea from separation logic. We gave a formal embedding of a variant of CSL to SAGL by defining the program invariant enforced in CSL as special assumptions and guarantees in SCAL. This shows that SAGL is more general than CSL. It also serves as a new way to prove the soundness of CSL.

Discussions and future work. The design of OCAP follows the invariant-based proof methodology, which is applied in most verification systems for safety properties. Given a verification system designed following the same methodology, it can be embedded into OCAP by defining an interpretation that encodes the program invariant enforced in the system. The linking of modules certified in different systems actually tests whether invariants enforced in these systems match at the point the interaction occurs. This methodology is very general for certifying safety properties of system software, but it is not clear how to extend it to support other properties, such as termination, liveness, fairness and resource usages. These properties are especially interesting for embedded and real-time systems. We will extend the OCAP framework and design program logics to support them in the future.

OCAP gives a general framework and formal disciplines for interaction of different verification systems, but it does not automatically solve the interaction problem for all systems. Interaction between different systems is always a challenging problem. As we showed in Chapter 6, in each interoperation scenario, we need to carefully formulate the program invariants and make sure they actually match at the linking point. This needs to be done for each specific applications and is non-trivial. We will try to find more applications to thoroughly test the applicability of OCAP.

The machine used by the OCAP framework is a sequential machine without support of interrupts and interactions with external devices. If we want to support the preemptive thread model on a single processor machine, we need to add the support of hardware interrupts and extend the OCAP framework to support interrupts too. Then we can might be able to embed SAGL and CSL shown in Chapter 7 into OCAP. Again, this will be part of the future work.

Another interesting future work is to push the framework and program logics to higher level. Currently the framework and all the program logics presented in this paper work at the assembly level, therefore we do not need to trust compilers. However, the specification at the assembly level is very complex because of the low-level details, such as calling conventions. To solve this problem, we can design program logics for higher level languages, such as C. Some low-level details can be abstracted away in specifications for higher-level languages. Then the interpretation that maps high-level specifications to low-level specifications needs to formalize a compilation procedure for the high level language. The soundness of the interpretation will be similar to the main theorem of type-preserving compilations [69].

Bibliography

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. on Programming Languages and Systems*, 17(3):507–535, 1995.

[2] 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.

[3] A. J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[4] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing process isolation. In *MSPC'06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 1–10, New York, NY, USA, Oct. 2006. ACM.

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

[6] A. W. Appel. Foundational proof-carrying code. In *LICS'01*, pages 247–258. IEEE Comp. Soc., June 2001.

[7] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. 27th ACM Symp. on Principles of Prog. Lang.*, pages 243–253. ACM Press, 2000.

[8] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23(5):657–683, 2001.

[9] A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *Proc. POPL'07*, pages 109–122, New York, NY, USA, 2007. ACM Press.

[10] 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.

[11] N. Benton. A typed, compositional logic for a stack-based abstract machine. In *Proc. Third Asian Symp. on Prog. Lang. and Sys. (APLAS'05), LNCS 3780*. Springer-Verlag, November 2005.

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

[13] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Proc. 32nd ACM Symp. on Principles of Prog. Lang.*, pages 259–270, 2005.

[14] S. Brookes. A semantics for concurrent separation logic. In *Proc. 15th International Conference on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 16–34, 2004.

[15] S. Brookes. A grainless semantics for parallel programs with shared mutable data. In *Proc. MFPS XXI*, volume 155 of *Electr. Notes Theor. Comput. Sci.*, pages 277–307, 2006.

[16] S. Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007. Journal version of [14].

[17] H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, pages 66–77, New York, NY, USA, June 2007. ACM Press.

[18] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE'03: International Conference on Software Engineering*, pages 385–395, 2003.

[19] B.-Y. Chang, A. Chlipala, G. Necula, and R. Schneck. The open verifier framework for foundational verifiers. In *TLDI'05*, pages 1–12, Jan. 2005.

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

[21] 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.

[22] 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.

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

[24] Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.0, Oct. 2005.

[25] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

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

[27] K. Crary. Toward a foundational typed assembly language. In *Proc. 30th ACM Symp. on Principles of Prog. Lang.*, pages 198–212, 2003.

[28] K. Crary and S. Sarkar. Foundational certified code in a metalogical framework. In *CADE'03*, volume 2741 of *LNCS*, pages 106–120. Springer, 2003.

[29] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In *Proc. 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 261–276, London, UK, 1999. Springer-Verlag.

[30] 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.

[31] X. Feng. An open framework for certified system software: Companion coq code. http://flint.cs.yale.edu/publications/feng-thesis, 2007.

[32] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science (LNCS)*, pages 173–188. Springer, 2007.

[33] X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework for foundational proof-carrying code. In *Proc. 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*, pages 67–78, New York, NY, USA, January 2007. ACM Press.

[34] X. Feng and Z. Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 254–267, New York, NY, USA, September 2005. ACM Press.

[35] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, pages 401–414, New York, NY, USA, June 2006. ACM Press.

[36] C. Flanagan and M. Abadi. Object types against races. In *CONCUR'99 – Concurrency Theory, volume 1664 of LNCS*, pages 288–303. Springer-Verlag, 1999.

[37] C. Flanagan and M. Abadi. Types for safe locking. In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 91–108. Springer-Verlag, 1999.

[38] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *Proc. 2002 European Symposium on Programming*, pages 262–277. Springer-Verlag, 2002.

[39] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Proc. 10th SPIN workshop*, pages 213–224, May 2003.

[40] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, pages 338–349, 2003.

[41] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ASE'02: Automated Software Engineering*, pages 3–12, 2002.

[42] D. Grossman. Type-safe multithreading in cyclone. In *Proceedings of the 2003 ACM SIGPLAN International workshop on Types in Languages Design and Implementation*, pages 13–25, 2003.

[43] Y. Guo, X. Jiang, Y. Chen, and C. Lin. A certified thread library for multithreaded user programs. In *Proc. 1st IEEE & IFIP International Symposium on Theoretical Aspects of Software Engineering (TASE'07)*, pages 117–126. IEEE Computer Society, June 2007.

[44] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A principled approach to operating system construction in haskell. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 116–128, New York, NY, USA, September 2005. ACM Press.

[45] N. A. Hamid and Z. Shao. Interfacing hoare logic and type systems for foundational proof-carrying code. In *Proc. 17th International Conference on Theorem Proving in Higher Order Logics*, volume 3223 of *LNCS*, pages 118–135. Springer-Verlag, Sept. 2004.

[46] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *LICS'02*, pages 89–100, July 2002.

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

[48] K. Havelund and T. Pressburger. Model checking Java programs using Java pathfinder. *Software Tools for Technology Transfer (STTT)*, 2(4):72–84, 2000.

[49] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI'04: Programming Language Design and Implementation*, pages 1–13, 2004.

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

[51] C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, editors, *Operating Systems Techniques*, pages 61–71. Academic Press, 1972.

[52] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[53] G. J. Holzmann. Proving properties of concurrent systems with SPIN. In *Proc. 6th International Conference on Concurrency Theory (CONCUR'95)*, volume 962 of *LNCS*, pages 453–455. Springer, 1995.

[54] W. A. Howard. The formulae-as-types notion of constructions. In *To H.B.Curry: Essays on Computational Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[55] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, Redmond, WA, 2005.

[56] G. C. Hunt. Singularity in a nutshell. Talk presented at 2007 Microsoft Research Faculty Summit, Redmond, Washington, July 2007.

[57] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM Symp. on Principles of Prog. Lang.*, pages 14–26, 2001.

[58] 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.

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

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

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

[62] T. Kleymann. Metatheory of verification calculi in lego — to what extent does syntax matter? In *TYPES'98*, volume 1657 of *Lecture Notes in Computer Science*, pages 133–148. Springer, 1998.

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

[64] C. Lin, A. McCreight, Z. Shao, Y. Chen, and Y. Guo. Foundational typed assembly language with certified garbage collection. In *Proc. 1st IEEE & IFIP International Symposium on Theoretical Aspects of Software Engineering (TASE'07)*, pages 326–335. IEEE Computer Society, June 2007.

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

[66] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1982.

[67] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[68] 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.

[69] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *POPL'98*, pages 85–97, 1998.

[70] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. on Programming Languages and Systems*, 21(3):527 – 568, May 1999.

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

[72] G. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Prog. Lang.*, pages 106–119. ACM Press, Jan. 1997.

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

[74] 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.

[75] 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, 2006.

[76] Z. Ni and Z. Shao. A translation from typed assembly languages to certified assembly programming. Technical report, Dept. of Computer Science, Yale University, New Haven, CT, Oct. 2006. Available at `http://flint.cs.yale.edu/flint/publications/talcap.html`.

[77] Z. Ni, D. Yu, and Z. Shao. Using xcap to certify realistic systems code: Machine context management. In *Proc. 20th International Conference on Theorem Proving in Higher Order Logics*, volume 4732 of *LNCS*, pages 189–206. Springer-Verlag, Sept. 2007.

[78] P. W. O'Hearn. Resources, concurrency and local reasoning. In *Proc. 15th Int'l Conf. on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 49–67, 2004.

[79] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007. Journal version of [78].

[80] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL'04*, pages 268–280, 2004.

[81] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[82] M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. In *Proc. 34rd ACM Symp. on Principles of Prog. Lang.*, pages 297 – 302. ACM Press, Jan. 2007.

[83] C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In *Proc. TLCA*, volume 664 of *LNCS*, 1993.

[84] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS'05: Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, 2005.

[85] 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.

[86] J. H. Reppy. CML: A higher concurrent language. In *Proc. 1991 Conference on Programming Language Design and Implementation*, pages 293–305, Toronto, Ontario, Canada, 1991. ACM Press.

[87] 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.

[88] J. C. Reynolds. Toward a grainless semantics for shared-variable concurrency. In *Proc. FSTTCS'04*, volume 3328 of *LNCS*, pages 35–48, 2004.

[89] 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.

[90] E. W. Stark. A proof technique for rely/guarantee properties. In *Proc. 5th Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *LNCS*, pages 369–391, 1985.

[91] R. Stata and M. Abadi. A type system for java bytecode subroutines. In *Prin. of Prog. Lang. (POPL'98)*, pages 149–160. ACM Press, 1998.

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

[93] K. Stølen. A method for the development of totally correct shared-state parallel programs. In *Proc. 2nd International Conference on Concurrency Theory (CONCUR'91)*, pages 510–525, 1991.

[94] G. Tan. *A Compositional Logic for Control Flow and its Application in Foundational Proof-Carrying Code*. PhD thesis, Princeton University, 2004.

[95] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *Proc. 18th International Conference on Concurrency Theory (CONCUR'07)*, page to appear, 2007.

[96] 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.

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

[98] D. P. Walker. *Typed Memory Management*. PhD thesis, Cornell University, Ithaca, NY, Jan. 2001.

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

[100] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proc. 28th ACM Symp. on Principles of Prog. Lang.*, pages 27–40, New York, NY, USA, 2001. ACM Press.

[101] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. In *Proc. 2003 European Symposium on Programming (ESOP'03)*, April 2003.

[102] 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.

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