An Open Framework for Foundational Proof-Carrying Code

Xinyu Feng[†] Zhaozhong Ni[†] Zhong Shao[†] Yu Guo[‡]

[†]Department of Computer Science
Yale University
New Haven, CT 06520-8285, U.S.A.

[‡]Department of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui 230026, China

Abstract

Software systems usually use many different computation features and span different abstraction levels (*e.g.,* user code level and the runtime system level). To build foundational certified systems, it is hard to have one verification system supporting all computation features. In this paper we present an open framework for foundational proof-carrying code (FPCC). It allows program modules to be specified and certified separately using different type systems or program logics. Certified modules (code + proof) can be linked to compose fully certified systems. The framework supports modular verification and proof reuse. It is extensible, and is expressive enough to allow invariants established in verification systems to be maintained when they are embedded in. Our framework is the first FPCC framework that systematically supports interoperation between different verification systems. It is fully mechanized in the Coq proof assistant with machine-checkable soundness proof.

1. Introduction

Foundational certified systems are packages containing machine code and mechanical proof about safety properties [15, 2]. Building foundational certified systems is hard because software systems usually use many different computation features (stacks and heaps, strong and weak memory update, first- and higher-order function pointers, sequential and concurrent control flows, *etc.*), and span different abstraction levels (*e.g.,* user level code and run-time system code such as thread schedulers and garbage collectors).

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) [14] 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. Recent works on certifying concurrent assembly code [23, 9] apply the rely-guarantee method to prove concurrency properties. They also use abstract machines with abstract instructions such as `fork` and `yield`.

It is hard (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 systems to compose a completely certified system.

Suppose we want to build FPCC package [2] which contains the machine code C and a proof showing that C satisfies the safety policy SP, as shown in Fig. 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.

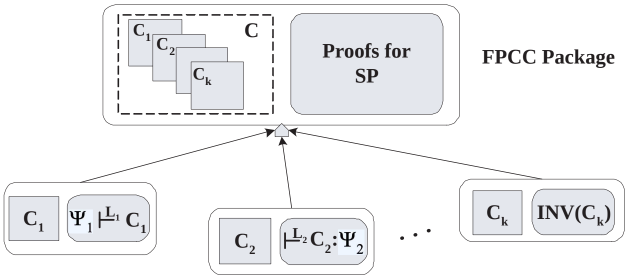

Figure 1. Building FPCC Package by Linking Certified Modules

To compose the certified modules, we need an open FPCC 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 verification 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 FPCC [3, 12, 8] only shows how to construct foundational proof for each specific verification system and does not support interoperation between systems, with the only exception of [11] 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 [12, 8] 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 [3, 4, 19] 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 [4] is defined specifically for type safety (*i.e.,* program never gets stuck). It is hard to use the indexed model for embedded code pointers to support Hoare-style program logics, which usually certifiess the partial correctness of programs with respect to program specifications. More discussion about related work will be given in section 7.

In this paper, we propose an open framework, OCAP, for developing foundational proof carrying code. OCAP is the first FPCC 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. 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. Our work on OCAP builds upon previous work on program verification but makes the following new contributions:

- OCAP is built to reason about real machine code, but it still allows user level code to be specified and certified with higher-level abstractions. Instead of introducing higher-level primitive operations in the machine, we let user code call runtime which implements the required functionality. Runtime code can be fully certified in a different verification system.
- 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, any program specification definable in Coq can be incorporated as OCAP program specification. The heterogeneous program specification also allows OCAP to specify embedded code pointers following the XCAP [16] approach, 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. The soundness of OCAP ensures that these invariants are maintained when foreign systems are embedded in the framework.
- Our applications of OCAP to support interoperation of verification systems are interesting in their own right. In the first application, we show how to link user code in TAL with a simple certified memory management library. TAL only supports weak-memory update and the free memory is invisible to TAL code. The memory management library is specified in SCAP [10], which supports reasoning about operations over free memory and still ensures that the invariants of TAL code is maintained. In our second application, we show how to construct FPCC for concurrent code *without* trusting the scheduler. The user thread code is certified using the rely-guarantee method [13], which supports thread modular verification; the thread scheduler is certified as sequential code in SCAP. They are linked in OCAP to construct FPCC package.

In the rest of this paper, we first present in section 2 the basic settings of the meta-logic and the machine we use to construct FPCC. We propose our OCAP framework in section 3. In section 4 we illustrate the embedding of a specific verification system, SCAP, in the OCAP framework. Then we show our two applications involving interoperation between different systems in section 5 and 6. Finally we discuss related work and conclude in Section 7.

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

Figure 2. The Target Machine TM

2. Basic Settings for FPCC

In the FPCC framework, the operational semantics of machine instructions is formalized in a mechanized meta-logic. Program logics or type systems are formally defined in the meta-logic with machine checkable soundness proof, resulting in smaller TCB for the safety proof. In this Section, we introduce the meta-logic we use for OCAP and present the formulation of our target machine.

2.1 The Mechanized Meta-Logic

We use the calculus of inductive constructions (CiC) [18] as our meta-logic, which is an extension of the calculus of constructions (CC) 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 [6], which we use to implement the results presented in this paper.

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

Syntax of some of mostly common-used CiC terms are shown above, where Prop is the universe of all propositions, and Type is the (stratified) universe of all terms. $A \to B$ represents function spaces. It also means logical implication if A and B have kind Prop. Meanings of other terms will be explained at the time they are used.

2.2 The Target Machine

The syntax of machine programs is defined in Fig. 2. 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. 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. $\mathbb{R}$ is a total function from registers to word values.

To simplify the presentation, we do not model the Von Newman architecture since reasoning about self-modifying code is beyond the scope of this paper. 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 interoperability issue we are trying to address. Also, we only show a small set of common-used instructions. Adding more instructions to the framework is straightforward.

To lay some structure over the flat code heap $\mathbb{C}$, we use the instruction sequence $\mathbb{I}$ to represent a basic code block. $\mathbb{C}[\mathtt{f}]$ extracts from $\mathbb{C}$ a basic block ending with a jump instruction.

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

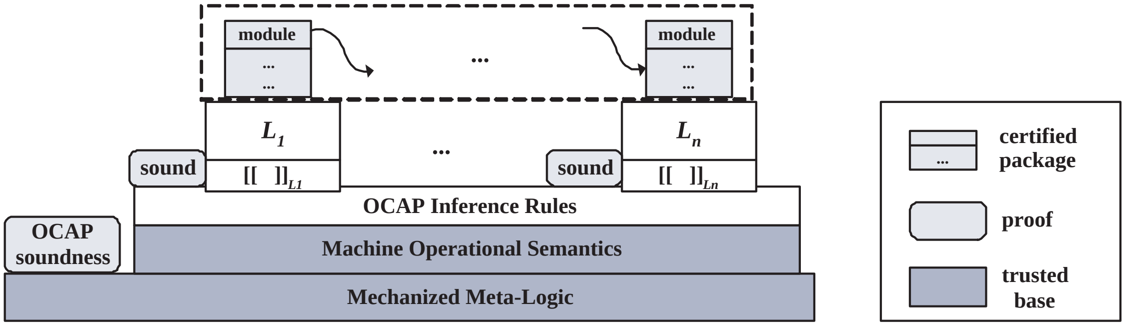

Figure 4. OCAP: an open framework for FPCC

$(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathsf{pc}) \longmapsto \mathbb{P}$		
if $\mathbb{C}(\mathsf{pc}) =$	then $\mathbb{P} =$	if
$\mathsf{j}\ \mathtt{f}$	$(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathtt{f})$	
$\mathsf{jr}\ \mathtt{r}_s$	$(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{R}(\mathtt{r}_s))$	
$\mathsf{jal}\ \mathtt{f}$	$(\mathbb{C},(\mathbb{H},\mathbb{R}\{\mathtt{r}_{31}\leadsto \mathsf{pc}+1\}),\mathtt{f})$	
$\mathsf{bgtz}\ \mathtt{r}_s,\mathtt{f}$	$(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathsf{pc}+1)$ $(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathtt{f})$	$\mathbb{R}(\mathtt{r}_s)\leq 0$ $\mathbb{R}(\mathtt{r}_s)>0$
other ι	$(\mathbb{C},\mathsf{Next}_\iota\ (\mathbb{H},\mathbb{R}),\mathsf{pc}+1)$	

where

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

Figure 3. Operational Semantics of TM

We define the operational semantics of machine programs in Fig. 3. One-step execution of a program is modeled as a transition relation $\mathbb{P}\longmapsto\mathbb{P}'$. $\mathbb{P}\longmapsto^k\mathbb{P}'$ means $\mathbb{P}$ reaches $\mathbb{P}'$ in k steps, and $\longmapsto^*$ is the reflexive and transitive closure of the step-relation. The auxiliary (partial) function $\mathsf{Next}_\iota\,(\_)$ defines the effects of sequential instructions over program states. It is partial because the operational semantics for memory access instructions is undefined if the memory address is not in the domain of $\mathbb{H}$.

2.3 Program Safety

The FPCC framework is used to construct the mechanized proof about program safety. Safety of the program means the execution of the program $\mathbb{P}$ satisfies certain 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 to prove the program safety. We first define a program invariant INV which is stronger than the safety policy. Then we prove that

1. the initial program $\mathbb{P}_0$ satisfies INV, *i.e.,* $\mathsf{INV}(\mathbb{P}_0)$;
2. $\forall\mathbb{P}.\,\mathsf{INV}(\mathbb{P})\rightarrow\exists\mathbb{P}'.\,(\mathbb{P}\longmapsto\mathbb{P}')\wedge\mathsf{INV}(\mathbb{P}')$.

Using CiC as the meta-logic, we can support very general specifications of the safety policy, which may range from simple type 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 type 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 method. On the other hand, suppose we have a program specification Ψ which defines the loop-invariants at certain points of the program. We can define SP as:

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

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 [12]. 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.

3. The OCAP Framework

The OCAP framework, as shown in Fig. 4, lays a set of Hoare-style inference rules over the raw machine semantics. Soundness of these rules are proved in the meta-logic with machine checkable proof, so they are not in the TCB. OCAP rules are expressive enough to embed most existing verification systems for low-level code. To embed a verification system, we define an interpretation which maps specifications in that system to assertions used in OCAP, then we prove system specific rules/axioms as lemmas based on the the interpretation and OCAP rules. Proofs constructed in each system can be incorporated as OCAP proof and be linked to compose the complete safety proof.

3.1 Overview of Certified Assembly Programming

We first give an overview of our previous work on certified assembly programming, upon which we develop our OCAP framework.

3.1.1 The CAP system

Yu *et al.* proposed a simple Hoare-style program logic CAP [22] to certify assembly code. CAP expects a program specification Ψ which collects the loop invariants asserted for each basic code block. Instead of defining its own assertion language in the meta-logic, CAP uses the meta-logic as the assertion language (a.k.a. shallow embedding) and each assertion $\mathtt{p}$ is a predicate over the program state, as shown below.

$$\begin{array}{rrcl}(\mathit{CHSpec}) & \Psi & \in & \mathit{Labels}\rightharpoonup\mathit{StatePred}\\ (\mathit{StatePred}) & \mathtt{p} & \in & \mathit{State}\rightarrow\mathit{Prop}\end{array}$$

CAP inference rules. Fig. 5 shows inference rules in 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.,* $\Psi\vdash\mathbb{C}:\Psi$.

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

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

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

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

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

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

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

Figure 5. Selected CAP Rules

- There exist a precondition $\mathtt{p}$ 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 $\mathtt{p}$ will guarantee the safe execution of $\mathbb{C}[\mathsf{pc}]$, *i.e.*, $\Psi \vdash \{\mathtt{p}\}\,\mathsf{pc} :\ \mathbb{C}[\mathsf{pc}]$.
- The current program state $\mathbb{S}$ satisfy $\mathtt{p}$.

To certify a program, we only need to prove that the initial program $(\mathbb{C},\mathbb{S}_0,\mathsf{pc}_0)$ satisfies the invariant. Soundness of CAP guarantees that the invariant holds at each step of execution.

The CDHP rule defines well formed code heap $\Psi \vdash \mathbb{C}:\Psi'$. 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 $\mathbb{C}$.

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 $\mathtt{p} \Rightarrow \mathtt{p}' \circ \mathsf{Next}_\iota$ is the shorthand for

$$\forall \mathbb{S}.\ \mathtt{p}\ \mathbb{S} \to \exists \mathbb{S}'.(\mathbb{S}' = \mathsf{Next}_\iota\ (\mathbb{S})) \wedge \mathtt{p}'\ \mathbb{S}'.$$

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

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.

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

Soundness. The soundness of CAP ensures that well-formed programs never get stuck, as shown in Theorem 3.2. Proof for the theorem follows the syntactic approach to proving type soundness [21].

Theorem 3.2 (CAP-Soundness)
If $\Psi \vdash \mathbb{P}$, then for all n there exists a $\mathbb{P}'$ such that $\mathbb{P} \longmapsto^n \mathbb{P}'$.

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 [16]. That is because CAP's specification language (predicate over state) is not expressive enough to express $\mathsf{codeptr}(\mathtt{f},\mathtt{p})\ \Psi$, which requires the reference to Ψ. A quick attack to this problem may be extending the specification langauge as follows:

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

and a code pointer $\mathtt{f}$ with specification $\mathtt{a}$ is defined as:

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

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 [16] defined a syntactic specification language. In their XCAP, the program specification is in the following form.

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

The meaning of extended proposition $\mathtt{P}$ 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* → *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 [10]. Then we get the following generic framework:

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

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 $\mathsf{codeptr}$ 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.

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

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

(*LangID*)	ρ	$::=$	n (*nat nums*)		
(*LangTy*)	$\mathcal{L}$	$::=$	(*CiC terms*)	$\in$	Type
(*CdSpec*)	θ	$::=$	(*CiC terms*)	$\in$	$\mathcal{L}$
(*OCdSpec*)	π	$::=$	$\langle \rho, \mathcal{L}, \theta \rangle$	$\in$	$LangID * (\Sigma X.X)$
(*CHSpec*)	Ψ	$\in$	$Labels * OCdSpec$		
(*Assert*)	$\mathtt{a}$	$\in$	$CHSpec \to State \to Prop$		
(*Interp*)	$\llbracket \_ \rrbracket_{\mathcal{L}}$	$\in$	$\mathcal{L} \to Assert$		
(*LangDict*)	$\mathcal{D}$	$\in$	$LangID \rightharpoonup \Sigma X.(X \to Assert)$		

Figure 6. Specification Constructs of OCAP

interpretation $\llbracket \_ \rrbracket_{\mathcal{L}}$ for the language $\mathcal{L}$ maps θ to the OCAP assertion $\mathtt{a}$. 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 $\llbracket \_ \rrbracket_{\mathcal{L}}$ 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 final solution. The final definition of OCAP program specification constructs is shown in Fig. 6. 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 assertions $\mathtt{a}$. To avoid circularity, we do not put the interpretation $\llbracket \_ \rrbracket_{\mathcal{L}}$ 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 $\mathtt{a}$:

$$\llbracket \langle \rho, \mathcal{L}, \theta \rangle \rrbracket_{\mathcal{D}} \triangleq \lambda \Psi, \mathbb{S}.\ \exists \llbracket \_ \rrbracket_{\mathcal{L}}.\ (\mathcal{D}(\rho) = \langle \mathcal{L},\ \llbracket \_ \rrbracket_{\mathcal{L}} \rangle) \wedge (\llbracket \theta \rrbracket_{\mathcal{L}}\ \Psi\, \mathbb{S}).$$

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

Fig. 7 shows OCAP inference rules. The PROG rule is similar to the one for CAP, but with several differences:

- 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 $\mathbb{C}[\mathsf{pc}]$ 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 $(\mathbb{C}, \mathbb{S}_0, \mathsf{pc}_0)$. The precondition $\mathtt{a}$ specifies the initial state $\mathbb{S}_0$. It takes Ψ to be able 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.

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

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

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

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

$$\frac{\mathcal{D}_1; \Psi_1 \vdash \mathbb{C}_1 : \Psi_1' \quad \mathcal{D}_2; \Psi_2 \vdash \mathbb{C}_2 : \Psi_2' \quad \mathcal{D}_1 \# \mathcal{D}_2 \quad \mathbb{C}_1 \# \mathbb{C}_2}{\mathcal{D}_1 \cup \mathcal{D}_2; \Psi_1 \cup \Psi_2 \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1' \cup \Psi_2'} \ (\text{LINK*})$$

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

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

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

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

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

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

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

Figure 7. OCAP Inference Rules

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 $(\llbracket \Psi'(\mathtt{f}) \rrbracket_{\mathcal{D}})$ by applying the corresponding interpretation defined in $\mathcal{D}$. Then we do another lifting $\langle \_ \rangle_{\Psi}$, which is defined as:

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

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

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

We also overload the conjunction connector "$\wedge$" for assertions:

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

Therefore, the lifted assertion $(\langle \llbracket \Psi'(\mathtt{f}) \rrbracket_{\mathcal{D}} \rangle_{\Psi})$ carries the knowledge of the code pointers 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. We use a "*" in the name to distinguish admissible rules from normal rules. The compatibility of partial mappings f and g is defined as

$$f\#g \triangleq \forall x.\, x \in dom(f) \wedge x \in dom(g) \rightarrow f(x) = g(x)\,.$$

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.

Lemma 3.3 is used to prove the admissibility of the LINK rule.

Lemma 3.3
If $\mathcal{D};\Psi \vdash \mathbb{C}:\Psi''$, $\mathcal{D} \subseteq \mathcal{D}'$, and $\Psi \subseteq \Psi'$, we have $\mathcal{D}';\Psi' \vdash \mathbb{C}:\Psi''$.

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 $\mathtt{a} \Rightarrow \mathtt{a}'$ as a shorthand for $\forall \Psi, \mathbb{S}.\ \mathtt{a}\ \Psi\ \mathbb{S} \rightarrow \mathtt{a}'\ \Psi\ \mathbb{S}$.

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

The* WEAKEN *rule. The WEAKEN rule is also admissible in OCAP. 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 5.2.3.

3.4 Soundness of OCAP

The soundness of OCAP inference rules is proved following the syntactic approach [21] to proving type soundness. We need to first prove the progress and preservation lemmas.

Lemma 3.4 (Progress)
If $\mathcal{D};\Psi \vdash \mathbb{P}$, there exists $\mathbb{P}'$ such that $\mathbb{P} \longmapsto \mathbb{P}'$.

Lemma 3.5 (Preservation)
If $\mathcal{D};\Psi \vdash \mathbb{P}$ and $\mathbb{P} \longmapsto \mathbb{P}'$, then we have $\mathcal{D};\Psi \vdash \mathbb{P}'$.

We prove two soundness theorems for OCAP. The first one shows that we can use OCAP to certify type safety (the non-stuckness property); while the second one shows that we can additionally certify the partial correctness of programs.

Theorem 3.6 (Soundness-Type Safety)
If $\mathcal{D};\Psi \vdash \mathbb{P}$, then for n there exists $\mathbb{P}'$ such that $\mathbb{P} \longmapsto^n \mathbb{P}'$.

Before we present Theorem 3.7, we first define $[\![\Psi(\mathtt{f})]\!]_{\mathcal{D}}$ as:

$$[\![\Psi(\mathtt{f})]\!]_{\mathcal{D}} \triangleq \begin{cases} \bigvee_{(\mathtt{f},\pi_i)\in\Psi} [\![\pi_i]\!]_{\mathcal{D}} & \exists\pi.(\mathtt{f},\pi)\in\Psi \\ \mathsf{FALSE} & \neg\exists\pi.(\mathtt{f},\pi)\in\Psi \end{cases}$$

where "$\vee$" is lifted for assertions.

Theorem 3.7 (Soundness-Correctness)
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

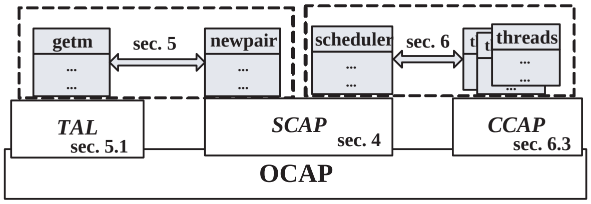

Figure 8. Case Studies for OCAP

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 $[\![\Psi(\mathtt{f})]\!]_{\mathcal{D}}\ \Psi\ \mathbb{S}'$.

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.

A similar soundness theorem was also proved for CAP [22]. Yu *et al.* [22] exploited CAP's support of partial correctness to certify an implementation of malloc and free libraries. CAP and OCAP's ability to support partial correctness of programs benefits from the way we specify codeptr. As we will discuss later, it is unclear how this soundness theorem can be proved using the step-indexed semantic model of codeptr.

3.5 Applicability of OCAP

In the rest of the paper, we will explore the applicability of the OCAP framework by showing how to embed existing type systems and program logics into the framework, and how to support interoperations between different systems at different abstraction levels. As shown in Fig. 8, we embed SCAP into OCAP to certify runtime library code. We also show how to embed TAL as a type system and CCAP as a program logic for concurrency verification. In section 5 we link TAL code with a simple memory management library certified in SCAP. In section 6, user-level threads certified in CCAP is linked with a simple implementation of a scheduler certified in SCAP. Since we mainly focus on interfacing systems, no familiarity of specific systems is required to understand these examples.

4. Case Study: Embedding SCAP in OCAP

In general, it takes three steps to embed a foreign system into OCAP: first identify the invariant enforced in the system; then define an interpretation for code specifications and embed the invariant in the interpretation; finally prove the soundness of the embedding by showing that inference rules in the original system can be proved as lemmas in OCAP based on the interpretation. In this section, we show how to embed SCAP into OCAP.

SCAP is a compositional Hoare-style program logic proposed in [10] for assembly code verification. It supports reasoning about function call/return without requiring specifications of return code pointers (which is a special form of embedded code pointers).

SCAP specification. SCAP uses a pair of predicates $(\mathtt{p},\mathtt{g})$ as code specifications (θ). As shown below, $\mathtt{p}$ is a predicate over a state; the guarantee $\mathtt{g}$ is a predicate over a pair of states. $\mathcal{L}_{\mathrm{SCAP}}$ specifies the type of θ. The code heap specification ψ maps code labels to θs.

$$\begin{array}{rlll} (\mathit{StatePred}) & \mathtt{p} & \in & \mathit{State} \rightarrow \mathit{Prop} \\ (\mathit{Guarantee}) & \mathtt{g} & \in & \mathit{State} \rightarrow \mathit{State} \rightarrow \mathit{Prop} \\ (\mathit{CdSpec}) & \theta & ::= & (\mathtt{p},\mathtt{g}) \quad \in \ \mathcal{L}_{\mathrm{SCAP}} \\ (\mathit{LangTy}) & \mathcal{L}_{\mathrm{SCAP}} & \triangleq & \mathit{StatePred} * \mathit{Guarantee} \\ (\mathit{LocalSpec}) & \psi & ::= & \{\mathtt{f} \rightsquigarrow \theta\}^* \quad \in \ \mathit{Labels} \rightharpoonup \mathcal{L}_{\mathrm{SCAP}} \end{array}$$

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

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

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

$$\frac{\begin{array}{l} (\mathtt{p}',\mathtt{g}') = \psi(\mathtt{f}') \qquad (\mathtt{p}'',\mathtt{g}'') = \psi(\mathtt{f}{+}1) \\ \forall \mathbb{S}.\ \mathtt{p}\,\mathbb{S} \to \mathtt{p}'\ (\mathbb{S}.\mathbb{H}, \mathbb{S}.\mathbb{R}\{\mathtt{r}_{31} \rightsquigarrow \mathtt{f}{+}1\}) \\ \forall \mathbb{S},\mathbb{S}'.\ \mathtt{p}\,\mathbb{S} \to \mathtt{g}'\ (\mathbb{S}.\mathbb{H}, \mathbb{S}.\mathbb{R}\{\mathtt{r}_{31} \rightsquigarrow \mathtt{f}{+}1\})\ \mathbb{S}' \\ \qquad\qquad \to \mathtt{p}''\,\mathbb{S}' \wedge (\forall \mathbb{S}''.\ \mathtt{g}''\,\mathbb{S}'\,\mathbb{S}'' \to \mathtt{g}\,\mathbb{S}\,\mathbb{S}'') \\ \forall \mathbb{S},\mathbb{S}'.\ \mathtt{g}'\,\mathbb{S}\,\mathbb{S}' \to \mathbb{S}.\mathbb{R}(\mathtt{r}_{31}) = \mathbb{S}'.\mathbb{R}(\mathtt{r}_{31}) \end{array}}{\psi \vdash \{(\mathtt{p},\mathtt{g})\}\,\mathtt{f} : \mathsf{jal}\ \mathtt{f}';\ \mathbb{I}} \ (\textsc{call})$$

$$\frac{\forall \mathbb{S}.\ \mathtt{p}\,\mathbb{S} \to \mathtt{g}\,\mathbb{S}\,\mathbb{S}}{\Psi \vdash \{(\mathtt{p},\mathtt{g})\}\,\mathtt{f} : \mathsf{jr}\ \mathtt{r}_{31}} \ (\textsc{ret})$$

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

$$\frac{(\mathtt{p}',\mathtt{g}') = \psi(\mathtt{f}') \quad \mathtt{p} \Rightarrow \mathtt{p}' \quad \forall \mathbb{S},\mathbb{S}'.\ \mathtt{p}\,\mathbb{S} \to \mathtt{g}'\,\mathbb{S}\,\mathbb{S}' \to \mathtt{g}\,\mathbb{S}\,\mathbb{S}'}{\psi \vdash \{(\mathtt{p},\mathtt{g})\}\,\mathtt{f} : \mathsf{j}\ \mathtt{f}'} \ (\textsc{j})$$

Figure 9. Selected SCAP Rules

Program invariant. The idea behind SCAP is very intuitive. The predicate $\mathtt{p}$ is the precondition, which plays the same role as the $\mathtt{p}$ in CAP. We use $\mathtt{g}$ to specify the behavior of code from the specified point to the return point of a function. A function call is made in SCAP by executing the jal instruction. Function returns by jumping to the register $\mathtt{r}_{31}$. The program invariant enforced in SCAP is formalized [10] as

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

where $(\mathtt{p},\mathtt{g})$ is the SCAP specification for the current program point. ψ is the code heap specification. It requires that, at any program point, the state satisfy the current precondition $\mathtt{p}$, and there be a well-formed control stack with certain depth n. The predicate wfst is defined as:

$$\begin{array}{ll} \mathsf{wfst}(0, \mathtt{q}, \psi) & \triangleq \neg\exists \mathbb{S}.\ \mathtt{q}\,\mathbb{S} \\ \mathsf{wfst}(n{+}1, \mathtt{q}, \psi) & \triangleq \forall \mathbb{S}'.\ \mathtt{q}\,\mathbb{S}' \to \exists \mathtt{p}',\mathtt{g}'.\ \psi(\mathbb{S}'.\mathbb{R}(\mathtt{r}_{31})) = (\mathtt{p}',\mathtt{g}') \wedge \\ & \qquad\qquad \mathtt{p}'\,\mathbb{S}' \wedge \mathsf{wfst}(n, \mathtt{g}'\,\mathbb{S}', \psi). \end{array}$$

At the return point of the current function (where $\mathtt{g}$ has been fulfilled), if the stack depth is greater than 0, $\mathtt{r}_{31}$ contains a code pointer with certain specification $(\mathtt{p}',\mathtt{g}')$. After the current function returns, $\mathtt{p}'$ holds so that it is safe to run the return continuation; and the stack is still well-formed with depths decreased by 1. When stack depth is 0, we are executing the topmost function and cannot return (*i.e.*, the guarantee cannot be fulfilled).

Fig. 9 shows selected SCAP rules. These rules ensure that the invariant specified above is maintained during program execution. The call rule (for jal) requires that, if the specification for the callee is $(\mathtt{p}',\mathtt{g}')$ and the return continuation $\mathbb{I}$ is well-formed with specification $(\mathtt{p}'',\mathtt{g}'')$, then

- the precondition $\mathtt{p}'$ of callee be satisfied after jal;
- the precondition $\mathtt{p}''$ for the return continuation be satisfied when the callee returns and has fulfilled its guarantee $\mathtt{g}'$;
- composing the behavior of the callee and the return continuation fulfill the guaranteed behavior $\mathtt{g}$; and
- the callee reinstate the return address when it returns.

The RET rule simply require that a function fulfill its guarantee before it returns. Therefore an identity transition will satisfy the remaining guarantee.

The rest instruction rules are easy to understand. Interested readers can refer to [10] for more details.

Embedding and soundness. To embed SCAP into OCAP, we first use the lifting function $\llcorner\psi\lrcorner_\rho$ to convert the ψ in SCAP to OCAP's specification Ψ, where ρ is the language ID assigned to SCAP.

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

For any ρ, 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 code heap specification Ψ.

$$\begin{array}{l} \mathsf{WFST}(0, \mathtt{q}, \mathcal{D}, \Psi) \triangleq \\ \qquad \forall \mathbb{S}'.\ \mathtt{q}\,\mathbb{S}' \to \exists \pi.\ (\mathsf{codeptr}(\mathbb{S}'.\mathbb{R}(\mathtt{r}_{31}), \pi) \wedge [\![\pi]\!]_{\mathcal{D}})\ \Psi\,\mathbb{S}' \\ \mathsf{WFST}(n{+}1, \mathtt{q}, \mathcal{D}, \Psi) \triangleq \\ \qquad \forall \mathbb{S}'.\ \mathtt{q}\,\mathbb{S}' \to \exists \mathtt{p}',\mathtt{g}'.\ (\mathbb{S}'.\mathbb{R}(\mathtt{r}_{31}),\ \langle\rho, \mathcal{L}_{\text{SCAP}}, (\mathtt{p}',\mathtt{g}')\rangle) \in \Psi \\ \qquad\qquad\qquad\qquad \wedge\ \mathtt{p}'\,\mathbb{S}' \wedge \mathsf{WFST}(n, \mathtt{g}'\,\mathbb{S}', \mathcal{D}, \Psi). \end{array}$$

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 $\mathtt{r}_{31}$ 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 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 rules shown in Fig. 9. The interpretation is *not* used until we want to link the certified SCAP code with code certified in other systems. We instantiate ρ and $\mathcal{D}$ in each specific application scenarios. 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 $\mathcal{D}$ for foreign code, let $\mathcal{D}' = \mathcal{D}\{\rho \rightsquigarrow \langle \mathcal{L}_{\text{SCAP}}, [\![\_]\!]^{(\rho,\mathcal{D})}_{\mathcal{L}_{\text{SCAP}}} \rangle\}$.

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

5. Case II: TAL with Certified Runtime

In this section, we will show how to link TAL code with certified memory allocation libraries. Unlike traditional TALs [14, 7] which are based on abstract machines with primitive operations for memory allocation, we present a variation of TAL for our TM (defined in section 2.2).

We use a simple function `newpair` to do memory allocation. The code for `newpair` is specified and verified in SCAP without knowing about the future interoperation with TAL. User code is certified in TAL. There is also a TAL interface for `newpair` so that the call to `newpair` can be type-checked. To allow the interoperation, we first embed both systems in OCAP. Then we show that, given the interpretations for TAL and SCAP, the TAL interface for `newpair` is compatible with the SCAP interface.

The tricky part is that TAL and SCAP have different views about machine states. As shown in Fig. 10, TAL (the left side) only knows

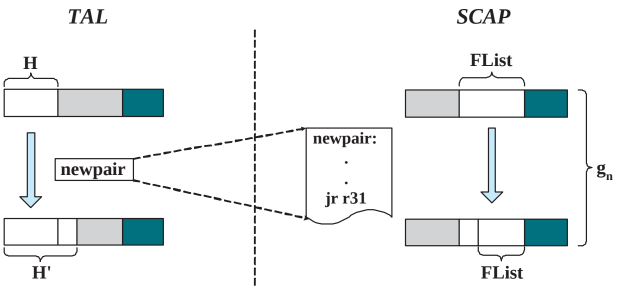

Figure 10. Interoperation with TAL and SCAP

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

Figure 11. Type Definitions of TAL

the heap reachable from the user 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 type. 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 [17].

We first embed into OCAP a TAL over TM. The embedding follows similar steps we did for SCAP.

5.1 Embedding TAL into OCAP

TAL types and typing rules. Figure 11 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 selected typing rules of TAL in Fig. 12 and 13. The TAL typing rules are similar[1] to the original TAL [14] and are not explained in details here. 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 Fig. 12 look very similar to CAP rules shown in Fig. 5. Actually this is exactly how we embed TAL in OCAP below.

The invariant enforced in TAL is that, at any step of execution, the program state is well-typed with respect to the code heap type ψ and certain register file type $[\Delta].\Gamma$. Judgment for well-typed state is represented as $\boxed{\psi \vdash \mathbb{S} : [\Delta].\Gamma}$. The TAL state typings and subtyping rules are shown in Fig. 13.

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

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

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

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

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

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

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

Figure 12. Selected TAL typing rules

Embedding of TAL The code specification θ in TAL is the register file type $[\Delta].\Gamma$. The type of its CiC encoding is $\mathcal{L}_{\text{TAL}}$. Then we define the mapping between the TAL code heap specification ψ and the OCAP code heap specification Ψ:

$$
\begin{aligned}
\llcorner \psi \lrcorner_{\rho} &\triangleq \{(\mathtt{f},\ \langle \rho, \mathcal{L}_{\text{TAL}}, [\Delta].\Gamma \rangle) \ \mid\ (\mathtt{f}, [\Delta].\Gamma) \in \psi\} \\
\ulcorner \Psi \urcorner^{\rho,\mathcal{L}} &= \{(\mathtt{f}, \theta) \ \mid\ (\mathtt{f},\ \langle \rho, \mathcal{L}, \theta \rangle) \in \Psi\}
\end{aligned}
$$

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 $\ulcorner \Psi \urcorner^{\rho,\mathcal{L}}$ collects from OCAP's Ψ the specifications of code certified in language ρ, and constructs a language-specific code heap specification.

To link TAL programs with run-time systems, the interpretation function for TAL specification is defined with an open parameter r, which is the invariant about memory invisible from TAL (the grey blocks in Fig. 10):

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

Here ρ is the language ID assigned to TAL; $f \uplus g$ means union of partial mappings with disjoint domains. Instead of building semantic models for TAL types, we reuse the TAL state typing ($\psi \vdash \mathbb{S} : [\Delta].\Gamma$ as shown in Fig. 13) as the interpretation. Also note that the invariant r is a predicate over Ψ and $\mathbb{H}$ only. Although expressiveness is limited, this should be sufficient for runtime resource because usually runtime does not reserve registers. Also this limitation can be lifted if we model the register file $\mathbb{R}$ as a partial mapping (like data heap).

Soundness. Theorem 5.1 states the soundness of TAL rules and the interpretation for TAL specifictions. It shows that, given the interpretation, TAL rules are derivable as lemmas in OCAP. The soundness is independent with the open parameter r.

Theorem 5.1 (TAL Soundness)
For all ρ and r, let $\mathcal{D} = \{\rho \rightsquigarrow \langle \mathcal{L}_{\text{TAL}},\ \llbracket \_ \rrbracket^{(\rho,r)}_{\mathcal{L}_{\text{TAL}}} \rangle\}$.

1. if $\psi \vdash \{[\Delta].\Gamma\}\,\mathbb{I}$ then $\mathcal{D} \vdash \{\langle \mathsf{a} \rangle_{\Psi}\}\,\mathbb{I}$, where $\mathsf{a} = \llbracket\, [\Delta].\Gamma \,\rrbracket^{(\rho,r)}_{\mathcal{L}_{\text{TAL}}}$ and $\Psi = \llcorner \psi \lrcorner_{\rho}$;

[1] 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.

$$\boxed{\vdash [\Delta].\Gamma \le [\Delta'].\Gamma'} \quad \textbf{(\textit{Subtyping})}$$

$$\frac{\Gamma(\mathtt{r}) = \Gamma'(\mathtt{r}) \quad \forall\, \mathtt{r} \in dom(\Gamma')}{\vdash [].\Gamma \le [].\Gamma'}\ (\textsc{subt}) \qquad \frac{\Gamma(\mathtt{r}) = \forall[\alpha,\Delta'].\Gamma' \qquad \Delta \vdash \tau'}{\vdash [\Delta].\Gamma \le [\Delta].\Gamma\{\mathtt{r}:\forall[\Delta'].\Gamma'[\tau'/\alpha]\}}\ (\textsc{tapp}) \qquad \frac{\Gamma(\mathtt{r}) = \tau[\tau'/\alpha] \qquad \Delta \vdash \tau'}{\vdash [\Delta].\Gamma \le [\Delta].\Gamma\{\mathtt{r}:\exists\alpha.\tau\}}\ (\textsc{pack})$$

$$\frac{\Gamma(\mathtt{r}) = \exists\alpha.\tau}{\vdash [\Delta].\Gamma \le [\alpha,\Delta].\Gamma\{\mathtt{r}:\tau\}}\ (\textsc{unpack}) \qquad \frac{\Gamma(\mathtt{r}) = \tau[\mu\alpha.\tau/\alpha]}{\vdash [\Delta].\Gamma \le [\Delta].\Gamma\{\mathtt{r}:\mu\alpha.\tau\}}\ (\textsc{fold}) \qquad \frac{\Gamma(\mathtt{r}) = \mu\alpha.\tau}{\vdash [\Delta].\Gamma \le [\Delta].\Gamma\{\mathtt{r}:\tau[\mu\alpha.\tau/\alpha]\}}\ (\textsc{unfold})$$

$$\boxed{\Delta \vdash \tau \quad \psi \vdash \mathbb{S}:[\Delta].\Gamma \quad \psi \vdash \mathbb{H}:\Phi \quad \psi;\Phi \vdash \mathbb{R}:\Gamma \quad \psi;\Phi \vdash \mathtt{w}:\tau \quad \psi;\Phi \vdash \mathtt{w}:\tau^{\varphi} \quad \vdash \tau^{\varphi} \le \tau^{\varphi'}}$$

$$\frac{ftv(\tau) \subseteq \Delta}{\Delta \vdash \tau}\ (\textsc{type}) \qquad \frac{\cdot \vdash \tau_i \quad \psi \vdash \mathbb{H}:\Phi \quad \psi;\Phi \vdash \mathbb{R}:\Gamma[\tau_1,\ldots,\tau_n/\alpha_1,\ldots,\alpha_n]}{\psi \vdash \mathbb{S}:[\alpha_1,\ldots,\alpha_n].\Gamma}\ (\textsc{state})$$

$$\frac{\psi;\Phi \vdash \mathbb{H}(\mathtt{l}):\Phi(\mathtt{l}) \quad \forall\, \mathtt{l} \in dom(\Phi)}{\psi \vdash \mathbb{H}:\Phi}\ (\textsc{heap}) \qquad \frac{\psi;\Phi \vdash \mathbb{R}(\mathtt{r}):\Gamma(\mathtt{r}) \quad \forall\, \mathtt{r} \in dom(\Gamma)}{\psi;\Phi \vdash \mathbb{R}:\Gamma}\ (\textsc{rfile})$$

$$\frac{}{\psi;\Phi \vdash \mathtt{w}:\mathsf{int}}\ (\textsc{int}) \qquad \frac{(\mathtt{f},[\Delta].\Gamma) \in \psi}{\psi;\Phi \vdash \mathtt{f}:\forall[\Delta].\Gamma}\ (\textsc{code}) \qquad \frac{\cdot \vdash \tau' \quad \psi;\Phi \vdash \mathtt{f}:\forall[\alpha,\Delta].\Gamma}{\psi;\Phi \vdash \mathtt{f}:\forall[\Delta].\Gamma[\tau'/\alpha]}\ (\textsc{poly}) \qquad \frac{\vdash \Phi(\mathtt{l}+i-1) \le \tau_i^{\varphi_i}}{\psi;\Phi \vdash \mathtt{l}:\langle \tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n} \rangle}\ (\textsc{tup})$$

$$\frac{\cdot \vdash \tau' \quad \psi;\Phi \vdash \mathtt{w}:\tau[\tau'/\alpha]}{\psi;\Phi \vdash \mathtt{w}:\exists\alpha.\tau}\ (\textsc{ext}) \qquad \frac{\psi;\Phi \vdash \mathtt{w}:\tau[\mu\alpha.\tau/\alpha]}{\psi;\Phi \vdash \mathtt{w}:\mu\alpha.\tau}\ (\textsc{rec}) \qquad \frac{\psi;\Phi \vdash \mathtt{w}:\tau}{\psi;\Phi \vdash \mathtt{w}:\tau^{\varphi}}\ (\textsc{init}) \qquad \frac{}{\psi;\Phi \vdash \mathtt{w}:\tau^{0}}\ (\textsc{uninit})$$

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

Figure 13. TAL typing rules – II

2. if $\psi \vdash \mathbb{C}:\psi'$ then $\mathcal{D};\llcorner\psi\lrcorner_{\rho} \vdash \mathbb{C}:\llcorner\psi'\lrcorner_{\rho}$.

Lemma 5.2 is used to prove the soundness theorem. It shows the TAL subtyping relation is sound with respect to the interpretation.

Lemma 5.2 (Subtyping Soundness)
For any ρ, r, $[\Delta].\Gamma$ and $[\Delta'].\Gamma'$, let $\mathsf{a} = \llbracket\, [\Delta].\Gamma \,\rrbracket^{(\rho,r)}_{\mathcal{L}_{\text{TAL}}}$ and $\mathsf{a}' = \llbracket\, [\Delta'].\Gamma' \,\rrbracket^{(\rho,r)}_{\mathcal{L}_{\text{TAL}}}$. If $\vdash [\Delta].\Gamma \le [\Delta'].\Gamma'$, we have $\mathsf{a} \Rightarrow \mathsf{a}'$.

5.2 Linking TAL with `newpair`

Since we have already embedded SCAP, our next step is to link TAL code with an implementation of `newpair` certified 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 $\mathtt{r}_{30}$. The callee-save registers are $\mathtt{r}_1,\ldots,\mathtt{r}_9$.

5.2.1 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 will be passed to `newpair` as the return address.

```
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.

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

From TAL's point of view, `newpair` takes no argument and returns a reference in $\mathtt{r}_{30}$ pointing to two fresh memory cells with types τ and τ' (tagged by 0). Also values of callee safe registers have to be maintained, which is enforced by the polymorphic type.

The user will certify the caller $\mathbb{C}_{\text{TAL}}$ by constructing the following derivations in TAL.

$$\psi_t \vdash \{\psi_t(\mathtt{getm})\}\,\mathtt{getm} : \mathbb{I}_{\mathtt{getm}} \tag{1}$$

$$\psi_t \vdash \{\psi_t(\mathtt{cont})\}\,\mathtt{cont} : \mathbb{I}_{\mathtt{cont}} \tag{2}$$

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

5.2.2 Certifying `newpair` in SCAP.

The following code schema shows the implementation $\mathbb{C}_{\text{SCAP}}$ of `newpair`, which largely follows the `malloc` function in [22]. We omit the actual code here.

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

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

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

Following [22], we use an assertions FList to specify the list of free memory blocks maintained by `newpair`. The SCAP code specification for `newpair` is $(\mathtt{p}_n,\mathtt{g}_n)$ where

$$\begin{aligned}
\mathtt{p}_n &\triangleq \mathsf{FList}\\
\mathtt{g}_n &\triangleq (\forall \mathtt{r} \in \{\mathtt{r}_1,\ldots,\mathtt{r}_9,\mathtt{r}_{31}\}.\, [\mathtt{r}] = [\mathtt{r}]') \wedge\\
&\qquad \begin{pmatrix}\mathsf{FList}\\ \mathsf{FList} * [\mathtt{r}_{30}]' \mapsto \_ * [\mathtt{r}_{30}]'+1 \mapsto \_\end{pmatrix} * \mathsf{ID}.
\end{aligned}$$

Recall that $\mathtt{g}$ in SCAP specifies the guarantee of functions. We use $[\mathtt{r}]$ to represent the value of $\mathtt{r}$ in the first state (the current state), while the primed value $[\mathtt{r}]'$ means the value of $\mathtt{r}$ in the second state (the return state). Here $\mathtt{g}_n$ says the function will reinstate the value of callee-save registers and the return address before it returns. Also, as shown in Fig. 10, the original FList is split into a smaller FList and a memory block of two-word size. The rest of the memory is not changed.

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

$$\psi_s \triangleq \{\mathtt{newpair} \rightsquigarrow (\mathtt{p}_n, \mathtt{g}_n)\}\,.$$

We certify `newpair` by constructing the SCAP derivation of

$$\psi_s \vdash \{(\mathtt{p}_n, \mathtt{g}_n)\}\,\mathtt{newpair} : \mathbb{I}_{\mathtt{newpair}} \tag{3}$$

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

5.2.3 Linking the caller and callee

So far, we have specified and certified the caller and callee independently in TAL and SCAP. Our next step is to link the caller and the callee in OCAP.

Suppose the language ID for TAL and SCAP are ρ and ρ' respectively. We use FList to instantiate the resource invariant r used in the interpretation for TAL. Therefore TAL's interpretation is $\llbracket \_ \rrbracket^{(\rho,\mathsf{FList})}_{\mathcal{L}_{\mathrm{TAL}}}$. The language dictionary $\mathcal{D}_{\mathrm{TAL}}$ is defined as:

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

We feed $\mathcal{D}_{\mathrm{TAL}}$ to the interpretation for SCAP, which is now $\llbracket \_ \rrbracket^{(\rho',\mathcal{D}_{\mathrm{TAL}})}_{\mathcal{L}_{\mathrm{SCAP}}}$ (see section 4 for the SCAP interpretation). The language dictionary for both languages is:

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

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

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

TAL and SCAP specifications are lifted to OCAP spec Ψ_{FULL}:

$$\begin{array}{llll}
\{(& \mathtt{getm}, & \langle \rho, \mathcal{L}_{\mathrm{TAL}}, \psi_t(\mathtt{getm}) \rangle &), \\
(& \mathtt{cont}, & \langle \rho, \mathcal{L}_{\mathrm{TAL}}, \psi_t(\mathtt{cont}) \rangle &), \\
(& \mathtt{newpair}, & \langle \rho', \mathcal{L}_{\mathrm{SCAP}}, \psi_s(\mathtt{newpair}) \rangle &), \\
(& \mathtt{newpair}, & \langle \rho, \mathcal{L}_{\mathrm{TAL}}, \psi_t(\mathtt{newpair}) \rangle &) \quad \}.
\end{array}$$

To certify $\mathbb{C}_{\mathrm{FULL}}$, we need to construct the proof for 4.

$$\mathcal{D}_{\mathrm{FULL}}; \Psi_{\mathrm{FULL}} \vdash \mathbb{C}_{\mathrm{FULL}} : \Psi_{\mathrm{FULL}} \tag{4}$$

By applying the OCAP CDHP rule, we need derivations for the well-formedness of each instruction sequence. By theorems 5.1 and 4.1, we can get most of the derivations for free from derivations (1), (2) and (3). The only tricky part is to show the `newpair` code is well-formed with respect to the TAL specification, *i.e.*,

$$\begin{array}{c}
\mathcal{D}_{\mathrm{FULL}} \vdash \{\langle \mathtt{a} \rangle_{\Psi_{\mathrm{FULL}}}\}\,\mathtt{newpair} : \mathbb{I}_{\mathtt{newpair}} \\
\text{where } \mathtt{a} = \llbracket \langle \rho, \mathcal{L}_{\mathrm{TAL}}, \psi_t(\mathtt{newpair}) \rangle \rrbracket_{\mathcal{D}_{\mathrm{FULL}}}\,.
\end{array} \tag{5}$$

To prove (5), we prove the following implication,

$$\mathtt{a} \Rightarrow \llbracket \langle \rho', \mathcal{L}_{\mathrm{SCAP}}, (\psi_s(\mathtt{newpair})) \rangle \rrbracket_{\mathcal{D}_{\mathrm{FULL}}}\,.$$

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 (5).

6. Case III: Certified Threads and Scheduler

As an important application of OCAP, we show how to construct FPCC for concurrent code *without* putting the thread scheduler code in the TCB, yet still support modular verification.

6.1 The Problem

Almost all work on concurrency verification assumes built-in language constructs for concurrency, including recent work on verification of concurrent assembly code [23, 9].

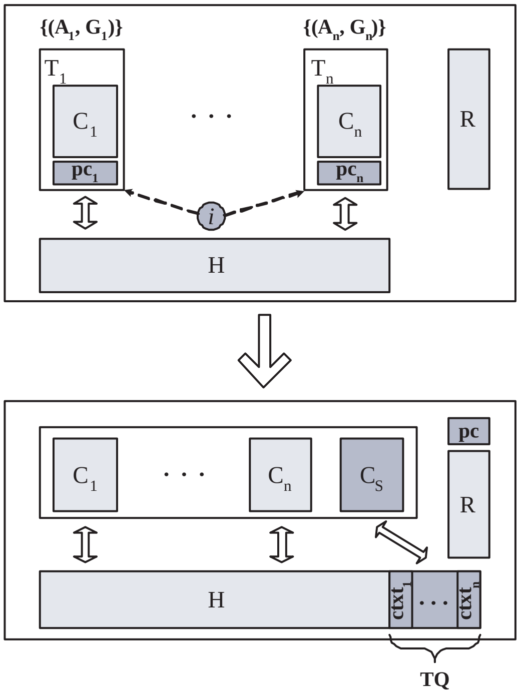

Figure 14. Concurrent Code at Different Abstraction Levels

The top part of Fig. 14 shows a (fairly low-level) abstract machine with built-in support of threads. Each thread T_i has its own code heap and program counter. 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 classic rely-guarantee method [13] allows concurrent code in such a machine to be certified in a thread modular way, as shown in CCAP [23]. The method assigns specification $\mathbb{A}$ 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:

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

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, there is no built-in abstractions for threads. As shown in the bottom part of Fig. 14, 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 hard to use the rely-guarantee method to certify the whole system ($\mathbb{C}_i$ and $\mathbb{C}_S$). We cannot treat $\mathbb{C}_S$ as a special thread because the context-switching behavior cannot be specified unless first-

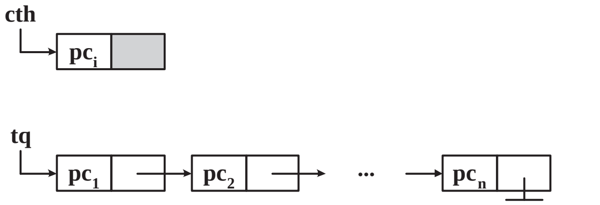

Figure 15. Current thread and the thread queue

class code pointers is supported. We do not know any existing work supporting first-class code pointers in a rely-guarantee-based framework. On the other hand, certifying all the code as sequential code loses thread modularity, thus impractical.

In our approach, we use CCAP to certify user thread code $\mathbb{C}_i$. Although the machine is low-level, the code can be specified and certified as if they are working at the higher-level machine shown in Fig. 14. 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.

6.2 Certifying The Scheduler Code in SCAP

User threads yield by calling the scheduler with the return continuation 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 in CCAP. There is no stack either because CCAP threads do not make function calls. Data structures for the scheduler is thus very simple, as shown in Fig. 15. Each thread context only contains the saved pc. The global constant $\mathtt{cth}$ points to the context of the current thread, and $\mathtt{tq}$ points to the other threads' contexts which are organized in a linked list. We use $\mathsf{TQ}(\mathtt{tq}, Q)$ to represent the linked list pointed by $\mathtt{tq}$ containing Q. Q is a (nonempty) list of code labels $[\mathsf{pc}_1, \ldots, \mathsf{pc}_n]$. Definition of TQ is omitted here.

The scheduler is then given the following specification $(\mathtt{p}_s, \mathtt{g}_s)$, where $|Q|$ represents the set of elements in the list Q.

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

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.

The scheduler code $\mathbb{C}_S$ can be certified using $(\mathtt{p}_s, \mathtt{g}_s)$ in SCAP without knowing about CCAP.

6.3 CCAP for User Thread Code

The code specifications in CCAP is a tuple $(\mathtt{p}, \breve{\mathtt{g}}, \mathbb{A}, \mathbb{G})$, as shown in Fig. 16. $\mathbb{A}$ and $\mathbb{G}$ are the assumption and guarantee. $\mathtt{p}$ is a predicate over the current state. Since the specified program point may be in the middle of yield points, we use $\breve{\mathtt{g}}$ to specify the "local" guarantee from the specified program point to the yield point. If the specified point immediately follows a yield, $\breve{\mathtt{g}}$ will be set to $\mathbb{G}$.

We use $\mathcal{L}_{\text{CCAP}}$ to represent the type of θ (in CiC). The following lift function converts ψ for CCAP to OCAP code heap spec.

$$\llcorner \psi \lrcorner_\rho \triangleq \{(\mathtt{f}, \langle \rho, \mathcal{L}_{\text{CCAP}}, (\mathtt{p}, \breve{\mathtt{g}}, \mathbb{A}, \mathbb{G}) \rangle) \mid \psi(\mathtt{f}) = (\mathtt{p}, \breve{\mathtt{g}}, \mathbb{A}, \mathbb{G})\}$$

$$\begin{array}{rrcl}
(\mathit{StPred}) & \mathtt{p}, \mathtt{q} & \in & \mathit{State} \to \mathit{Prop} \\
(\mathit{Assumption}) & \mathbb{A} & \in & \mathit{State} \to \mathit{State} \to \mathit{Prop} \\
(\mathit{Th\text{-}Guarant.}) & \breve{\mathtt{g}}, \mathbb{G} & \in & \mathit{State} \to \mathit{State} \to \mathit{Prop} \\
(\mathit{CdSpec}) & \theta & ::= & (\mathtt{p}, \breve{\mathtt{g}}, \mathbb{A}, \mathbb{G}) \\
(\mathit{CHSpec}) & \psi & ::= & \{\mathtt{f} \leadsto \theta\}^*
\end{array}$$

Figure 16. Specification Constructs for CCAP

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

$$\frac{\text{for all } \mathtt{f} \in \mathit{dom}(\psi'): \quad \psi \vdash \{\psi'(\mathtt{f})\}\, \mathtt{f} : \mathbb{C}[\mathtt{f}]}{\psi \vdash \mathbb{C} : \psi'} \ (\text{CDHP})$$

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

$$\frac{\begin{array}{ll} \psi \vdash \{(\mathtt{p}', \breve{\mathtt{g}}', \mathbb{A}, \mathbb{G})\}\, \mathtt{f}{+}1 : \mathbb{I} & \iota \in \{\mathsf{addu}, \mathsf{addiu}, \mathsf{lw}, \mathsf{subu}, \mathsf{sw}\} \\ \mathtt{p} \Rightarrow \mathtt{p}' \circ \mathsf{Next}_\iota & \forall \mathbb{S}, \mathbb{S}'.\ \mathtt{p}\, \mathbb{S} \to \breve{\mathtt{g}}'\ (\mathsf{Next}_\iota\ (\mathbb{S}))\, \mathbb{S}' \to \breve{\mathtt{g}}\, \mathbb{S}\, \mathbb{S}' \end{array}}{\psi \vdash \{(\mathtt{p}, \breve{\mathtt{g}}, \mathbb{A}, \mathbb{G})\}\, \mathtt{f} : \iota; \mathbb{I}} \ (\text{SEQ})$$

$$\frac{\begin{array}{l} \forall \mathbb{S}.\ \mathtt{p}\, \mathbb{S} \to \breve{\mathtt{g}}\, \mathbb{S}\, (\mathbb{S}.\mathbb{H}, \mathbb{S}.\mathbb{R}\{\mathtt{r}_{31} \leadsto \mathtt{f}{+}1\}) \\ \forall \mathbb{S}, \mathbb{S}'.\mathtt{p}\, \mathbb{S} \wedge \mathbb{A}\, \mathbb{S}\, \mathbb{S}' \to \mathtt{p}\, \mathbb{S}' \quad (\mathtt{p}, \mathbb{G}, \mathbb{A}, \mathbb{G}) = \psi(\mathtt{f}{+}1) \end{array}}{\psi \vdash \{(\mathtt{p}, \breve{\mathtt{g}}, \mathbb{A}, \mathbb{G})\}\, \mathtt{f} : \mathsf{jal}\ \mathtt{yield}; \mathbb{I}} \ (\text{YIELD})$$

Figure 17. Selected CCAP Rules

Selected CCAP rules are shown in Fig. 17. Since CCAP uses a built-in yield, we revise its original YIELD rule here to adapt to our TM, where yield is done by calling the runtime. To certify the user thread code $\mathbb{C}_i$, we use CCAP rules and construct the following derivation $\psi \vdash \mathbb{C}_i : \psi'$. We will not explain these rules in detail here because they are not essential to understand the interoperation.

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

- $\mathtt{p}$ 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;
- 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.*, $\mathsf{NI}([\ldots, (\mathbb{A}_i, \mathbb{G}_i), \ldots, (\mathbb{A}, \mathbb{G})])$;
- 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.*, $\forall \mathbb{S}, \mathbb{S}'.\ \mathtt{p}_i\, \mathbb{S} \wedge \mathbb{A}_i\, \mathbb{S}\, \mathbb{S}' \to \mathtt{p}_i\, \mathbb{S}'$;
- when we reach a state that $\breve{\mathtt{g}}$ is satisfied (*i.e.*, the current thread can yield), it is safe for all threads in Q to take over the control, *i.e.*, $(\breve{\mathtt{g}}\, \mathbb{S}) \Rightarrow \mathtt{p}_i$ for all i, where $\mathbb{S}$ is the current program state.

The following interpretation for CCAP specification $(\mathtt{p}, \breve{\mathtt{g}}, \mathbb{A}, \mathbb{G})$ simply specifies these invariants.

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

where

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

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

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

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

However, CCAP 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 CCAP derivations after we know the specification of `yield` at the time of linkage.

Theorem 6.1 (CCAP Soundness)
Let $\mathcal{D} = \mathcal{D}_c \cup \{\rho \leadsto \langle \mathcal{L}_{\mathrm{SCAP}},\ [\![\_]\!]^{(\rho,\mathcal{D}_c)}_{\mathcal{L}_{\mathrm{SCAP}}} \rangle\}$ and

$$\Psi_s = \{(\mathtt{yield},\ \langle \rho, \mathcal{L}_{\mathrm{SCAP}}, (\mathtt{p}_s, \mathtt{g}_s) \rangle\ )\}$$

1. If we have $\psi \vdash \{(\mathtt{p}, \breve{\mathtt{g}}, \mathbb{A}, \mathbb{G})\}\,\mathtt{f} : \mathbb{I}$, then $\mathcal{D} \vdash \{\langle \mathtt{a} \rangle_{\Psi}\}\,\mathtt{f} : \mathbb{I}$, where $\Psi = \llcorner \psi \lrcorner_{\rho'} \cup \Psi_s$ and $\mathtt{a} = [\![\,(\mathtt{p}, \breve{\mathtt{g}}, \mathbb{A}, \mathbb{G})\,]\!]^{\rho'}_{\mathcal{L}_{\mathrm{CCAP}}}$.
2. If we have $\psi \vdash \mathbb{C} : \psi'$ in CCAP, then $\mathcal{D}; \Psi \vdash \mathbb{C} : \llcorner \psi' \lrcorner_{\rho'}$, where $\Psi = \llcorner \psi \lrcorner_{\rho'} \cup \Psi_s$.

7. Related Work and Conclusion

Semantic approaches to FPCC. The semantic approach to FPCC [3, 4, 19] 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. For instance, it is challenging to define models for mutable weak references, and the resulting indexed model is heavyweight to use [1].

OCAP also uses a different specification for embedded code pointers than the step-indexed semantic model [4, 19] 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 3.7.

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

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

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 3.7 could be proved if this model is used: when we do an indirect jump to a code pointer $\mathsf{codeptr}(\mathbb{R}(\mathtt{r}), \mathtt{a})$, 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. More detailed discussion of this issue is shown in Appendix A.

Syntactic approaches to FPCC. The OCAP framework is quite different from the original syntactic approach [12, 8] 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 [12] 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 [12] 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 [22] to certify `malloc/free` libraries. The system used there does not have modular support of embedded code pointers. Ni and Shao [16] 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 [10] 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 [5]. 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.

Conclusion. We propose OCAP as an open framework for constructing FPCC. OCAP lays a thin layer of Hoare-style inference rules over a bare meta-logic. Assertions in OCAP rules is expressive enough to specify the invariants enforced in foreign verification systems. We have embedded in OCAP a program logic (SCAP) for certifying run-time code, a type system (TAL) and a program logic for concurrency verification (CCAP). OCAP also supports separate verification of program modules in different foreign systems. We showed two applications of OCAP's support of system interoperations. The first one shows how to use OCAP to link TAL code with certified libraries; the second one shows how to construct FPCC for concurrent code without trusting the scheduler: scheduler code and user thread code are certified in different systems and linked in OCAP. The OCAP framework has been implemented in the Coq proof assistant with machine checkable soundness proof.

References

[1] A. J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[2] A. W. Appel. Foundational proof-carrying code. In *LICS'01*, pages 247–258. IEEE Comp. Soc., June 2001.

[3] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *POPL'00*, pages 243–253. ACM Press, 2000.

[4] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23(5):657–683, 2001.

[5] 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.

[6] Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.0, Oct. 2005.

[7] K. Crary. Toward a foundational typed assembly language. In *Proc. 30th ACM Symp. on Principles of Prog. Lang.*, pages 198–212, 2003.

[8] K. Crary and S. Sarkar. Foundational certified code in a metalogical framework. In *CADE'03*, volume 2741 of *LNCS*, pages 106–120. Springer, 2003.

[9] X. Feng and Z. Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *ICFP'05*, pages 254–267, 2005.

[10] 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.

[11] 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.

[12] 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.

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

[14] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *POPL'98*, pages 85–97, 1998.

[15] G. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Prog. Lang.*, pages 106–119. ACM Press, Jan. 1997.

[16] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *POPL'06*, pages 320–333, 2006.

[17] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL'04*, pages 268–280, 2004.

[18] C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In *Proc. TLCA*, volume 664 of *LNCS*, 1993.

[19] G. Tan. *A Compositional Logic for Control Flow and its Application in Foundational Proof-Carrying Code*. PhD thesis, Princeton University, 2004.

[20] G. Tan and A. W. Appel. A compositional logic for control flow. In *VMCAI'06*, volume 3855 of *LNCS*, pages 80–94. Springer, 2006.

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

[22] 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.

[23] 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.

A. Indexed Model for Code Pointers Revisited

Appel and McAllester proposed an indexed model [4] for type systems. Based on the indexed model, Tan and Appel [19, 20] defined a compositional Hoare-logic system $\mathcal{L}_c$ which serves as an intermediate level to translate LTAL to the meta-logic, but they did not show how first-class code pointers can be supported in $\mathcal{L}_c$.

In this section, we first present an indexed Hoare-logic systems for TM, which is similar to $\mathcal{L}_c$ but with modular support of indirect jumps. Then we explain why such a system, based on the indexed semantic model, cannot be used to prove general partial correctness of programs.

In the indexed Hoare-logic system, the specification for the code heap is a partial mapping from code labels to indexed predicates over the whole machine state ($\mathbb{C}$ and $\mathbb{S}$):

$$\begin{array}{lrcl} (\mathit{CHSpec}) & \Psi & ::= & \{\mathtt{f} \rightsquigarrow \mathtt{a}\}^* \\ (\mathit{CdSpec}) & \mathtt{a} & \in & \mathit{nat} \rightarrow \mathit{CodeHeap} \rightarrow \mathit{State} \rightarrow \mathit{Prop} \end{array}$$

To support general indirect jumps, we need to give a specification for code pointers. Following the indexed model, a valid (up to k steps) code pointer $\mathtt{f}$ with precondition $\mathtt{a}$ is defined as:

$$\begin{array}{l} \mathsf{codeptr}(\mathtt{f},\mathtt{a}) \triangleq \\ \qquad \lambda k,\mathbb{C},\mathbb{S}.\ \forall \mathbb{S}'.\forall j < k.\ \mathtt{a}\ j\ (\mathbb{C},\mathbb{S}') \rightarrow \mathsf{Safen}(i,(\mathbb{C},\mathbb{S}',\mathtt{f}))\,, \end{array}$$

where $\mathsf{Safe}(j,\mathbb{P})$ means the program $\mathbb{P}$ can execute at least j steps:

$$\mathsf{Safe}(k,\mathbb{P}) \triangleq \forall j \le k.\ \exists \mathbb{P}'.\mathbb{P} \longmapsto^{j} \mathbb{P}'\,.$$

We also define $\mathsf{Safe}(\mathbb{P})$ as:

$$\mathsf{Safe}(\mathbb{P}) \triangleq \forall k.\ \mathsf{Safe}(k,\mathbb{P})\,.$$

The code heap $\mathbb{C}$ satisfies its specification Ψ up to k steps only if each code label in Ψ is a valid one up to k steps:

$$\begin{array}{l} \models \mathbb{C}{:}_k\Psi \triangleq \mathit{dom}(\Psi) \subseteq \mathit{dom}(\mathbb{C}) \wedge \\ \qquad\qquad \forall \mathtt{f} \in \mathit{dom}(\Psi).\forall \mathbb{S}.\ \mathsf{codeptr}(\mathtt{f},\Psi(\mathtt{f}))\ k\,\mathbb{C}\,\mathbb{S}\,. \end{array}$$

In the following discussion, we will assume that $\mathit{dom}(\Psi) = \mathit{dom}(\mathbb{C})$. This is not necessary. We want it simply because we want to make our presentation to be as close to [4] as possible.

The safety of code $\mathbb{C}$ at state $\mathbb{S}_0$ and pc_0 can be proved using the following theorem, a paraphrase of the Theorem 40 in [4].

Theorem A.1 (Indexed-Sound)

$$\frac{\forall k.\ \models \mathbb{C}{:}_k\Psi \qquad \mathtt{a} = \Psi(\mathsf{pc}) \qquad \forall k.\ \mathtt{a}_k\,\mathbb{C}\,\mathbb{S}}{\mathsf{Safe}(\mathbb{C},\mathbb{S},\mathsf{pc})}$$

To use above theorem, the challenging part is to prove $\forall k.\ \models \mathbb{C}{:}_k\Psi$. By induction over k, we need to prove (6):

$$\forall k.\ (\models \mathbb{C}{:}_k\Psi)\ \rightarrow\ (\models \mathbb{C}{:}_{k+1}\Psi)\,, \tag{6}$$

which, as suggested in [4] (Theorem 44), can be proved from:

$$\forall \mathtt{f} \in \mathit{dom}(\Psi).\ \mathsf{safe\_at}(\mathbb{C},\Psi,\mathtt{f})\,, \tag{7}$$

where

$$\begin{array}{l} \mathsf{safe\_at}(\mathbb{C},\Psi,\mathtt{f}) \triangleq \\ \quad \forall k,\mathbb{S}.\ (\models \mathbb{C}{:}_k\Psi) \wedge \Psi(\mathtt{f})\ k\,\mathbb{C}\,\mathbb{S} \rightarrow \\ \qquad\quad \exists \mathbb{S}',\mathtt{f}'.(\mathbb{C},\mathbb{S},\mathtt{f}) \longmapsto (\mathbb{C},\mathbb{S}',\mathtt{f}') \wedge \Psi(\mathtt{f}')\ k{-}1\,\mathbb{C}\,\mathbb{S}'\,. \end{array}$$

The framework looks good so far. In addition to the "non-stuckness" property specified by $\mathsf{Safe}(\mathbb{P})$, we can prove partial correctness as a by-product of (7). The partial correctness can be formalized as:

$$\begin{array}{l} \mathsf{Safe}(\mathbb{C},\mathbb{S},\mathsf{pc}) \wedge \\ \quad \forall k,\mathbb{S}',\mathsf{pc}'.\ ((\mathbb{C},\mathbb{S},\mathsf{pc}) \longmapsto^{k} (\mathbb{C},\mathbb{S}',\mathsf{pc}')) \rightarrow \forall n.\Psi(\mathsf{pc}')\ n\,\mathbb{C}\,\mathbb{S}'\,. \end{array}$$

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

$$\frac{\text{for all } \mathtt{f} \in \mathit{dom}(\Psi'): \quad \mathtt{a} = \langle \Psi'(\mathtt{f}) \rangle_\Psi \quad \Psi \vdash \{\mathtt{a}\}\,\mathtt{f} : \mathbb{C}(\mathtt{f})}{\Psi \vdash \mathbb{C} : \Psi'} \; (\textsc{cdhp})$$

$\boxed{\Psi \vdash \{\mathtt{a}\}\,\mathtt{f} : \iota}$ ***(Well-formed instruction)***

$$\frac{\mathtt{a} \Rightarrow \Psi(\mathtt{f}')}{\Psi \vdash \{\mathtt{a}\}\,\mathtt{f} : \mathsf{j}\ \mathtt{f}'} \; (\textsc{j})$$

$$\frac{\begin{array}{l}\mathtt{a} \Rightarrow \lambda k, \mathbb{C}, \mathbb{S}.\ \exists \mathtt{a}'.\mathsf{Monotone}(\mathtt{a}') \wedge \\ \qquad\qquad (\mathsf{codeptr}(\mathbb{S}.\mathbb{R}(\mathtt{r}_s), \mathtt{a}') \wedge \mathtt{a}')\ k\ \mathbb{C}\ \mathbb{S}\end{array}}{\Psi \vdash \{\mathtt{a}\}\,\mathtt{f} : \mathsf{jr}\ \mathtt{r}_s} \; (\textsc{jr})$$

$$\frac{\iota \in \{\mathsf{addu}, \mathsf{addiu}, \mathsf{lw}, \mathsf{sw}\} \quad \mathtt{a} \Rightarrow \lambda k, \mathbb{C}.\ (\Psi(\mathtt{f}+1)\ k\ \mathbb{C}) \circ \mathsf{Next}_\iota}{\Psi \vdash \{\mathtt{a}\}\,\mathtt{f} : \iota} \; (\textsc{seq})$$

Figure 18. Indexed Hoare-style rules

This is not surprising because (7) essentially formulates the progress and preservation properties.

However, Appel and McAllester did not give any guidance to construct the proof of (7). $\mathcal{L}_c$ is supposed to play such a role, but there is no rules for indirect jumps. We extend $\mathcal{L}_c$ and present a set of Hoare-style rules in Figure 18. Here the lifting of predicate $\mathtt{a}$ is defined as:

$$\langle \mathtt{a} \rangle_\Psi \triangleq \lambda k, \mathbb{C}, \mathbb{S}.\ \mathsf{Monotone}(\mathtt{a}) \wedge (\models \mathbb{C} :_k \Psi) \wedge (\mathtt{a}_k\ \mathbb{C}\ \mathbb{S}),$$

where $\mathsf{Monotone}(\mathtt{a})$ means:

$$\mathsf{Monotone}(\mathtt{a}) \triangleq \forall k, \mathbb{C}, \mathbb{S}.\ \mathtt{a}\ k\ \mathbb{C}\ \mathbb{S} \rightarrow \forall j < k.\ \mathtt{a}\ j\ \mathbb{C}\ \mathbb{S}.$$

Readers can see that the definition of $\langle \mathtt{a} \rangle_\Psi$ is similar to the lifted assertion we defined in OCAP (see section 3.3) if we unfold the definition of $\models \mathbb{C} :_k \Psi$. The extra monotonicity requirement for $\mathtt{a}$ corresponds to the requirement for types in [4]. As usual, we use $\mathtt{a} \Rightarrow \mathtt{a}'$ as a shorthand for $\forall k, \mathbb{C}, \mathbb{S}.\ \mathtt{a}\ k\ \mathbb{C}\ \mathbb{S} \rightarrow \mathtt{a}'\ k\ \mathbb{C}\ \mathbb{S}$. The conjunction connector "$\wedge$" is overloaded for assertions.

Instead of proving (7) directly, we want to let the user prove $\Psi \vdash \mathbb{C} : \Psi$ instead. Unfortunately, based on the definition of $\mathsf{codeptr}$ and the $\textsc{jr}$ rule, we cannot prove (7) from ($\Psi \vdash \mathbb{C} : \Psi$). The $\textsc{jr}$ rule only tells us that $\mathtt{r}_s$ is a code pointer with certain precondition $\mathtt{a}'$ and $\mathtt{a}'$ holds at the time of the jump. Since the definition of $\mathsf{codeptr}$ is independent with Ψ, we do not know whether the target address is specified in Ψ or not, and, if specified, whether the specification is the same with (or weaker than) $\mathtt{a}'$ or not.

To solve this problem, we have to use a weaker definition of $\mathsf{safe\_at}$:

$$\begin{array}{l}\mathsf{safe\_at}(\mathbb{C}, \Psi, \mathtt{f}) \triangleq \\ \qquad \forall k, \mathbb{S}.(\models \mathbb{C} :_k \Psi) \wedge \Psi(\mathtt{f})\ k\ \mathbb{C}\ \mathbb{S} \rightarrow \\ \qquad\qquad \exists \mathtt{a}', \mathbb{S}', \mathtt{f}'.(\mathbb{C}, \mathbb{S}, \mathtt{f}) \longmapsto (\mathbb{C}, \mathbb{S}', \mathtt{f}') \wedge \\ \qquad\qquad\qquad (\mathsf{codeptr}(\mathtt{f}', \mathtt{a}') \wedge \mathtt{a}')\ k\ \mathbb{C}\ \mathbb{S}.\end{array}$$

With this weaker definition, we can prove (7) from $\Psi \vdash \mathbb{C} : \Psi$, and (7) still implies (6). Tan [19] essentially uses the weaker version of (7) to prove the soundness of TAL_1's indirect jump rule.

However, the problem with this weaker definition of $\mathsf{safe\_at}$ is that the preservation cannot be proved. As a result, we cannot use the system to prove the partial correctness of programs as formulated above. We can construct a counter example to show that, there exist a $\mathbb{C}$, $\mathbb{S}$, pc and Ψ, even though we have $\Psi \vdash \mathbb{C} : \Psi$ (therefore $\forall k.\ \models \mathbb{C} :_k \Psi$) and $\forall k.\ \Psi(\mathsf{pc})\ k\ \mathbb{C}\ \mathbb{S}$, we can find an $\mathbb{S}'$ and pc' such that $(\mathbb{C}, \mathbb{S}, \mathsf{pc}) \longmapsto^* (\mathbb{C}, \mathbb{S}', \mathsf{pc}')$ but $\forall k.\Psi(\mathsf{pc}')\ k\ \mathbb{C}\ \mathbb{S}'$ does not hold.

```
f:    addiu  r31, r0, cont    ;save return addr
      j      h                ;call function h
cont: j      cont             ;infinite loop

h:    addiu  r1, r0, 1        ;update r1
      jr     r31              ;return
```

For the code heap $\mathbb{C}$ shown above, we first give it a specification Ψ_1:

$$\begin{array}{l}\Psi_1 \triangleq \{\mathtt{f} \rightsquigarrow \lambda k, \mathbb{C}, \mathbb{S}.\mathsf{TRUE}, \ldots, \mathtt{cont} \rightsquigarrow \lambda k, \mathbb{C}, \mathbb{S}.\mathsf{TRUE}, \\ \qquad \mathtt{h} \rightsquigarrow \mathsf{codeptr}(\mathbb{S}.\mathbb{R}(\mathtt{r}_{31}),\ \lambda k, \mathbb{C}, \mathbb{S}.\mathsf{TRUE}), \ldots\}.\end{array}$$

We can prove that $\Psi_1 \vdash \mathbb{C} : \Psi_1$ (which implies $\forall k.\ \models \mathbb{C} :_k \Psi_1$), therefore we know that

$$\forall k, \mathbb{S}.\ \mathsf{codeptr}(\mathtt{cont}, \lambda k, \mathbb{C}, \mathbb{S}.\mathsf{TRUE})\ k\ \mathbb{C}\ \mathbb{S} \qquad (8)$$

Then we define Ψ_2 as:

$$\Psi_2 \triangleq \Psi_1\{\mathtt{cont} \rightsquigarrow \lambda k, \mathbb{C}, \mathbb{S}.\ (\mathbb{S}.\mathbb{R}(\mathtt{r}_1) = 0)\}.$$

We can also prove $\Psi_2 \vdash \mathbb{C} : \Psi_2$ by using (8), which was proved in the last round. However, it is trivial to see that when the program reaches code label $\mathtt{cont}$, the program state cannot satisfy $\Psi_2(\mathtt{cont})$, so we cannot prove the partial correctness of the program with respect to the code specification Ψ_2.

One way to solve the problem is to use a different $\textsc{jr}$ rule, as shown below:

$$\frac{\begin{array}{l}\mathtt{a} \Rightarrow \lambda k, \mathbb{C}, \mathbb{S}.\ \exists \mathtt{a}'.\mathsf{Monotone}(\mathtt{a}') \wedge (\mathtt{a}' \Rightarrow \Psi(\mathbb{S}.\mathbb{R}(\mathtt{r}_s))) \wedge \\ \qquad\qquad (\mathsf{codeptr}(\mathbb{S}.\mathbb{R}(\mathtt{r}_s), \mathtt{a}') \wedge \mathtt{a}')\ k\ \mathbb{C}\ \mathbb{S}\end{array}}{\Psi \vdash \{\mathtt{a}\}\,\mathtt{f} : \mathsf{jr}\ \mathtt{r}_s} \; (\textsc{jr}')$$

Using the $\textsc{jr}$' rule, we cannot certify the code shown above using the specification Ψ_2. However, this rule requires the knowledge about all the possible target addresses of the indirect jump, which breaks modularity of the system [16].

One may want to change the definition of Safe to specify the partial correctness, *e.g.*, to define Safe as:

$$\begin{array}{l}\mathsf{Safe}(k, (\mathbb{C}, \mathbb{S}, \mathsf{pc})) \triangleq \\ \qquad \Psi(\mathsf{pc})\ k\ \mathbb{C}\ \mathbb{S} \wedge \\ \qquad\quad \exists \mathbb{S}', \mathsf{pc}'.\ (\mathbb{C}, \mathbb{S}, \mathsf{pc}) \longmapsto (\mathbb{C}, \mathbb{S}', \mathsf{pc}') \wedge \mathsf{Safe}(k-1, (\mathbb{C}, \mathbb{S}', \mathsf{pc}')).\end{array}$$

However, the definition of Ψ now becomes circular because Ψ uses $\mathsf{codeptr}$, which uses Safe, which uses Ψ.