A Certified Thread Library for Multithreaded User Programs

Yu Guo Xinyu Jiang Yiyun Chen Chunxiao Lin
Department of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui 230026, China
{guoyu,wewewe,cxlin3}@mail.ustc.edu.cn yiyun@ustc.edu.cn

Abstract

Ensuring the safety of multithreaded software is a task both important and challenging. Currently, most approaches focus on the safety of multithreaded programs rather than the runtime based on which those concurrent programs run. In order to fundamentally solve this problem, a method of ensuring the safety of the runtime should be developed. Such a runtime could be organized as a thread library typically.

This paper presents the development and certification of a simple but realistic thread library. The thread library provides common multi-threading features such as dynamic thread creation, termination and joining as well. This library also carries machine-checkable proof which guarantees the library does not violate the safety policies. This paper also presents an approach to link the library to existing certified multithreaded user programs to form an integrated foundational proof-carrying code (FPCC) package. Comparing with the uncertified libraries, our work makes multithreaded applications much more reliable.

1. Introduction

Multithreaded software is widely employed in realistic applications. For example, in a web browser, it is common that while one thread is displaying images or text, another thread is retrieving data from the Internet. However, the safety of multithreaded programs is hard to ensure, since the interference between the simultaneously executing threads must be taken into account.

Many efforts have been devoted to the verification of concurrent programs. Jones [13] introduced the compositional rely-guarantee method (or A-G method) [26, 7, 5] to describe the state changes performed by the environment and by the program respectively. Lamport proposed the Temporal Logic of Action (TLA) [14] as a logic for specifying and reasoning about concurrent programs at the high-language level. Xu et al. [24] proposed a logic system to verify deadlock freedom and convergence by rely-guarantee method. Model checkers [9] are developed to verify concurrent programs with a fixed number of threads. Flanagan et al. [10] used A-G method to check java multithreaded programs. CCAP [26] applied the A-G method to the assembly code based on a concurrent abstract machine with a built-in thread scheduler. CMAP [7] proposed by Feng and Shao supports thread-modular reasoning with dynamic thread creation and termination.

However, most of the previous work concentrates on safe multithreaded programming, not the runtime (thread library). Nonetheless, only when the safety of underlying thread library is guaranteed, can the programs verified by their methods be safe. The thread library often contains lurking flaws which are subtle and hard to detect and fix due to its complexity, so a fully certified thread library is of urgent necessity. However, the certifying task is full of challenges. A thread library generally involves sophisticated manipulations on memory and machine context. The invariants are subtle and hard to specify. Furthermore, the control flow transfers between threads and the scheduler of thread library increase the certification burden.

In this paper, we propose a simple but realistic certified thread library, named CTL, which implements dynamic thread creation, termination and joining. CTL is written in low-level code and certified thoroughly in the program logic SCAP [8]. The certification convinces us that the semantics of CTL conform to its formal specifications. The code and its specifications, as well as corresponding safety proof are all encoded in a foundational mathematical logic and packed together to form a foundational proof-carrying code (FPCC) package [1, 12], in which a neat and consistent logic system instead of the entire complicated thread library has to be trusted. In this way, not only multithreaded programs but also CTL itself can be reasoned about and verified on a solid and rigorous base.

Even if we have the individual safety proof of them, it is still inadequate to ensure the safety of the interactions be-
tween CTL and multithreaded programs due to the absence of safety proof for the code with respect to linkage. Although the construction of such safety proof is possible, it is non-trivial because of the differences between specification languages, as well as certification methods’ variety.

Recently, Feng et al. [6] proposed an open certification framework (OCAP) to support inter-operation of different certification systems. OCAP serves as a common layer in which different program logics can be embedded.

Based on OCAP, we demonstrate in this paper the linkage between CTL certified in SCAP and user programs certified in CMAP, and construct the extra safety proof of interaction, as shown in Figure 1.

The main contributions of our work are as follows:

- We describe, specify, and certify a simple but realistic thread library CTL. It provides common multithreading features including thread scheduling, dynamic thread creation, termination and joining.

- We define formal specifications to capture the semantics of CTL routines. This library carries machine-checkable proof which ensures that the library does not violate the safety policies.

- Meanwhile, we adapt and embed CMAP in a simplified OCAP framework. Thus, CTL can be linked to the user programs certified in CMAP to construct an integrated mechanized FPCC package. As far as we know, CTL is the first thread library which can be safely linked to multithreaded user programs.

We have formalized the work presented in this paper, including CTL, OCAP, CMAP, and their soundness proof, in the Coq proof assistant [3]. Interested readers may find them on our web site [11].

The remainder of this paper is organized as follows: Section 2 presents the formalization of basic settings. In Section 3 we describe and certify the CTL library. We discuss the verification framework CMAP in which the multithreaded programs are certified in Section 4. Section 5 shows the linkage of CTL and user programs in a simplified OCAP framework. Section 6 is related work and the conclusion.

2. Basic settings

In order to certify CTL in the FPCC framework, we formalize all the related concepts into a mechanized meta-logic. In other words, the machine model, the code of CTL, its specifications, program logics and related safety proof are all based on a common formal logic, resulting in smaller trusted computing base for safety. In this section, we will present these basic settings.

The mechanized meta-Logic. We use the calculus of inductive constructions (CiC) [21] as our meta-logic. CiC is supported by the Coq proof assistant [3], which we use to implement the work presented in this paper.

The target machine. A MIPS-style [15] target machine (TM) is chosen as our machine model on which our thread library runs. We omit some physical machine features irrelevant to threading, such as address alignment, bit-arithmetic etc. The target machine and its operational semantics are formally defined in Figure 2. A program \( P \) contains a code heap \( C \) and an updatable state \( S \). A code heap \( C \) is a segment of memory mapping addresses to machine instructions, but \( C \) is read-only and isolated from the mutable data heap \( H \). The state \( S \), which specifies the execution of the program, consists of a register file, mutable data heap \( H \) and the program counter \( pc \). The target machine has 32 general-purpose registers. Following the MIPS convention, the table below shows the register alias and usage.

<table>
<thead>
<tr>
<th>Register</th>
<th>Alias</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_0 )</td>
<td></td>
<td>always zero</td>
</tr>
<tr>
<td>( r_0 )</td>
<td></td>
<td>always zero</td>
</tr>
<tr>
<td>( a_0 )</td>
<td></td>
<td>arguments</td>
</tr>
<tr>
<td>( r_4 )</td>
<td></td>
<td>arguments</td>
</tr>
<tr>
<td>( r_7 )</td>
<td></td>
<td>arguments</td>
</tr>
<tr>
<td>( t_0 )</td>
<td></td>
<td>return values</td>
</tr>
<tr>
<td>( t_7 )</td>
<td></td>
<td>return values</td>
</tr>
<tr>
<td>( r_8 )</td>
<td></td>
<td>return values</td>
</tr>
<tr>
<td>( r_15 )</td>
<td></td>
<td>return values</td>
</tr>
<tr>
<td>( k_0 )</td>
<td></td>
<td>reserved</td>
</tr>
<tr>
<td>( k_1 )</td>
<td></td>
<td>reserved</td>
</tr>
<tr>
<td>( r_26 )</td>
<td></td>
<td>reserved</td>
</tr>
<tr>
<td>( r_27 )</td>
<td></td>
<td>reserved</td>
</tr>
<tr>
<td>( r_{31} )</td>
<td></td>
<td>return address</td>
</tr>
</tbody>
</table>

The basic TM instruction set covers the common MIPS instructions for arithmetics, jump, conditional branch and load/store. It is easy to add more instructions in our TM.

The relation \( P \xrightarrow{i} P' \) indicates that the program state \( P \) steps to the program state \( P' \). The relation \( P \xrightarrow{\sigma}^k P' \) means that \( P \) reaches \( P' \) in \( k \) steps, and \( \xrightarrow{\sigma} \) is the reflexive and transitive closure of the step relation. The auxiliary function \( \text{Next}(\cdot) \) specifies the state transition according to the operational semantics instruction \( i \). We use the notation \( \mathcal{S}_{i} \) to denote \( \text{Next}(\mathcal{S}) \).

Safety. Safety of the program means that:

1. the execution of the programs in CTL will not go stuck;
2. the code of CTL satisfies certain safety specifications. The code heap specification \( \Psi \) is a map from code labels \( \ell \) to code specifications \( \theta \), as defined below:

\[
(CdSpec) \quad \theta ::= \cdots \quad (CHSpec) \quad \Psi ::= \{ \ell \sim \theta \}^\ast
\]
then
(lock(TCB). Each TCB
asserts a heap with only one memory cell, at address
A
. We use \( H \models A \) if the heap predicate \( A \) is valid with \( H \).
\( A_1 \land A_2 \) asserts the heap \( H_1 \cup H_2 \), in which \( H_1 \models A_1 \) and \( H_2 \models A_2 \) hold respectively. \( \top \) is valid with any heap. \( l \mapsto w \) asserts a heap with only one memory cell, at address \( l \) with content \( w \).

3. CTL: a certified thread library

In this section, we give a detailed description of CTL. Because of space limitations, we cannot fully discuss the
certifications of every routine of CTL. So we concentrate on certifying the yielding routine, which is an essential part of
the thread library.

3.1. Overview

CTL is a lightweight implementation of thread library whose thread model is similar to GNU Pthreads [4] or FSU
threads [16], in which threads are implemented in the userspace and the machine context switching is performed by an
application library without knowledge of the kernel. This model does not rely on the kernel threads and is adaptable to
various platforms. CTL supports non-preemptive scheduling and can directly run on the processors without interrupts
handling, for example, the SPIM simulator [20]. CTL is fully certified at the assembly level, so the certification of
the compiler correctness can be spared.

3.2. Thread model

We use pseudo C code to illustrate the prototype of the core data structures and API of CTL.

The core data structures in the CTL space are shown in
Figure 3. The main data structure is a queue, whose elements are called thread control block (TCB). Each TCB
identifies one dynamic thread and contains one thread id and corresponding executing context. The TCB is prototyped by:

```c
struct tcb {
    word id; /* unique id */
    word pc; /* program counter */
    word state; /* ready, dead or wait */
    word wait_id; /* id of thread to wait */
    word regfile[28]; /* array for saving registers */
};
```

Meanwhile, an isolated TCB identifies the current running thread. Note that the global pointers referring to these data
structures are stored in a global pointer array.

CTL provides a threading API:

```c
void ctl_yield(void);
word ctl_spawn((* void)());
word ctl_exit();
word ctl_join(int thread_id);
```
σdead or w Specification constructs. by the entry state. can also be treated as a general postcondition parameterized a code to the return state of the corresponding procedure. g organized. The specification constructs of SCAP and some specified thread terminates. It takes a thread id as argument. ctl_join() the current thread and marking its TCB with dead flag. The if it does not terminate. The role of new thread id. Lastly, the new TCB is marked with ready machine context is cloned and stored into the TCB with a cate a new thread control block (TCB). Then the current cates a new thread control block (TCB). Then the current

Figure 4. SCAP

The routine ctl_yield() causes the current thread to yield its execution in favor of another thread with certain scheduling policy. ctl_yield() stores the execution context in the queue of the TCBs and picks up one TCB, loads it and switches the control to the new continuation. ctl_yield() adopts the round-robin scheduling algorithm.

Thread creation is achieved by using ctl_spawn() with a parameter of start code pointer. This function first allocates a new thread control block (TCB). Then the current machine context is cloned and stored into the TCB with a new thread id. Lastly, the new TCB is marked with ready flag and put into the queue. A thread will be running forever if it does not terminate. The role of ctl_exit() is stopping the current thread and marking its TCB with dead flag. The ctl_join() function suspends the calling thread until the specified thread terminates. It takes a thread id as argument.

3.3. Certification of CTL

SCAP. We use SCAP [8] to certify CTL. SCAP supports modular certification of assembly code with function call/return abstraction, making CTL routines well-organized. The specification constructs of SCAP and some selected SCAP rules are shown in Figure 4. A code specification θSCAP is a pair of two predicates p and g. p is the precondition, while g is a predicate over the entry state and the future return state after jr ra. g relates the entry state of a code to the return state of the corresponding procedure. g can also be treated as a general postcondition parameterized by the entry state.

Specification constructs. Figure 5 describes the specification constructs of CTL. We use the word value v to specify the thread id σ. The state ts of a thread may be ready, dead or wait. A thread control block Tcb consists of a pc, a

\[(\text{Thread-id}) \ σ := w\]
\[(\text{RegFileX}) \ θ := \mathbb{R} \setminus \{r0 \sim \ldots, k0 \sim \ldots, k1 \sim \ldots, ra \sim \ldots\}\]
\[(\text{TCB}) \ Tcb := (pc, θ, ts, σ)\]
\[(\text{Th State}) \ ts := \text{ready | dead | wait}\]
\[(\text{Th Queue}) \ Q := \{\} | \{σ \sim \text{Tcb} \} | Q\]

Figure 5. Specification constructs of CTL

partial registers file θ and a thread state ts. A partial registers file θ is a registers file θ excluding r0, k0, k1 and ra. The registers k0 and k1 are preserved for thread scheduling and then unusable in user programs. Because the register r0 always holds the value of zero, it needn’t to be saved. When ra is used to call the routines of CTL, its value is saved in the Tcb.pc field. A thread queue Q is a dictionary mapping σ to Tcb and constructed inductively.

Formal core data structures. We formally specify the core data structures that are used in our thread library CTL. As shown in Figure 3, the memory space for CTL is divided into four parts, the global pointer array, the isolated TCB for current thread, the thread queue and the free-block list.

\[\text{Core}(σ, Tcb, Q) \triangleq \exists gbd. gbd \rightarrow \text{ctlhd}, \text{ctlid}, fr\]
\[\ast QNode(ctlhd, σ, Tcb, NULL) \ast TQ(ctlhd, tl, Q) \ast \text{GoodL}(fl)\]

The global pointers (ctlhd, ctlid, fl) are saved in the memory starting from gbd. ctlhd points to the current thread TCB, ctlid and tl point to the thread queue. fl points to a memory block list, which is used for dynamic heap allocation.

\[QNode(p, σ, (pc, θ, ts, σo), q) \triangleq (p \rightarrow q, 34)\]
\[\ast (p \rightarrow σ, pc, ts, σo)\]
\[\ast (p + 4 \rightarrow θ, x1, \ldots, θ, x25, θ, x26, θ, x29, θ, x30)\]

\[\text{Cth}(p, σ, Tcb) \triangleq \text{QNode}(p, σ, Tcb, NULL)\]

We define the predicate TQ to model the queue for threads. TQ has three arguments, the formal queue Q which is isomorphism to the concrete memory data, a queue-head pointer hd and a queue-tail pointer tl. TQseg models a queue segment in the middle of the queue, with a non-null pointer pointing to the next node. TQseg is defined inductively on the structure of Q.

\[\text{TQseg}(\{\}, hd, tl, q) \triangleq (hd = tl) \land (hd = q)\]
\[\text{TQseg}(\{σ \rightarrow \text{Tcb}\} \cup Q, hd, tl, q) \triangleq \text{TQseg}(hd, σ, Tcb, q)\]
\[\text{TQseg}(\{σ \rightarrow \text{Tcb}\} \cup Q, hd, tl, q) \triangleq \exists q'. \text{QNode}(hd, σ, Tcb, q') \ast \text{TQseg}(Q, q', tl, q)\]
\[\text{TQ}(Q, hd, tl) \triangleq \text{TQseg}(Q, hd, tl, NULL)\]
As defined above, a queue segment consists of several nodes modeled by QNode. The routines in the CTL may traverse, search, add a TCB node or delete one in the queue, whose structure should be preserved.

When a thread is created, the scheduler will allocate a heap block from the free block list. If a thread is joined by another one, the scheduler will free its TCB. The definition of free block-list Goodl follows the work by Yu et al. [25] and Xiang et al. [23], and it can be ported to our framework directly.

Implementation of yielding. The yielding routine is used to schedule threads and perform the context switching. The code body of the \texttt{ctl\_yield()} are presented in Figure 6.

The first phase of yielding (from \texttt{YIELD} to \texttt{APPENDTCB}) consists in loading address of the current TCB to ra, saving the machine context to the current TCB and loading other global pointers. The code address of the next instruction of the current thread has already been saved in ra. ra has to be saved into TCB firstly, because ra will be used to call \texttt{SAVECTX}.

The second phase of yielding is appending the current TCB (by calling \texttt{APPENDTCB}) to the thread queue, searching for a ready TCB through the queue, and fetching it out (by calling \texttt{FETCHTCB}). Since there is at least one ready TCB (consider the one just appended) in the queue, the routine \texttt{FETCHTCB} never fails to return. The algorithm of searching ready thread depends on the scheduling policy. Here the naive FIFO method is used and then the code of searching is a loop over the thread queue.

The role of the last switch phase (from \texttt{SWITCH}) of yielding is to switch the machine context from the old context to the new context fetched by \texttt{FETCHTCB}. Concretely, it sets ko with the address of the new TCB, then makes a tail call to \texttt{LOADCTX}. Inside \texttt{LOADCTX}, the registers file are restored including ra. The last step is tricky, and when program returns, the control flow is transferred to the new thread.

Certification of yielding. Part of the formal specifications of \texttt{ctl\_yield()} are also presented in Figure 6. Note that the guarantees in the middle of code body are not listed because they are unnecessary for the readers to understand the specification, although indispensable to the certification process. For the same reason, the initial state \((R, H, pc)\) and the final state \((R', H', pc')\), playing their roles as parameters of \texttt{p} and \texttt{g}, are omitted as well.

The specification of \texttt{ctl\_yield()} \((p, g)\) is defined below:

\[
p, g \triangleq \exists Q. ([\text{Core}(\ldots, Q) \cdot \text{Top})
\]

\[
g, p \triangleq \lambda \left( (R, H, pc) \right) \cdot \forall A, Q, Tcb. \exists \sigma_i.
\]

where

\[
\left( \left( \lambda \left( \text{Core}(\sigma, Tcb, Q) \right) \cdot A \right) \right.
\]

\[
\left( \lambda \left( \text{Core}(\sigma, (pc, R, \text{ready}_r), Q') \right) \cdot A \right)
\]

\texttt{ctl\_yield()} mainly performs context switching. Its precondition \texttt{p}, requires that the memory space of CTL should be well-formed, specified by the predicate \texttt{Core}. \texttt{g} is a condition which specifies the actions performed by the whole \texttt{ctl\_yield()} routine. We assume that the control flow is transferred to thread \(\sigma\), whose TCB is \(Tcb_i = (pc, R, \text{ready}_r)\). Then \(g\) specifies the postconditions which \texttt{ctl\_yield()} must satisfies that: (i) at the exiting point of \texttt{ctl\_yield()}, the old machine context is added into \(Q\); (ii) a ready control block \(Tcb_i\), which contains the current register file \(R\), and the code pointer \(pc\), has been fetched out of \(Q\); (iii) the memory space of \texttt{Core} is still well-formed while the irrelevant heap \(A\) is unchanged. Obviously, the specification of \texttt{ctl\_yield()} is independent of CTL scheduling algorithm. \texttt{Core} is related to the implementation, which should be hidden from user programs.

By the SCAP rules presented in Figure 4, it can be proved that the yielding routine satisfies these specifications.

Certification of CTL. The certification of \texttt{ctl\_spawn()}, \texttt{ctl\_exit()} and \texttt{ctl\_join()} follows this method of certifying the \texttt{ctl\_yield()} routine. We take \(\xi_{\text{CTL}}\) as the code heap of CTL and \(\Psi_{\text{CTL}}\) as the code heap specification. By the CDHP rule, we have:

\[
\Psi_{\text{CTL}} \vdash_{\text{SCAP}} \xi_{\text{CTL}} : \Psi_{\text{CTL}}
\]

4. CMAP: a concurrent program logic

CMAP is a program logic for certifying multithreaded assembly code with unbounded dynamic thread creation and termination. It assumes that the register files are thread local, i.e., saving and loading registers during context switching. CMAP is based on an abstract machine with built-in thread operations, e.g., \texttt{yield}, \texttt{spawn} and \texttt{exit} are treated as atomic primitives. This approach simplify the certification of user programs. However, the operations are not directly supported by most existing hardwares.

In Figure 7, we present a simplified CMAP system, which is adapted to our thread model. Notice that the whole
system is unchanged except for the primitives of yield, spawn and exit are replaced with function calls in CTL.

The code specification $\theta_{\text{CMAP}}$ is a quadruple $(p, g, A, G)$. The predicate $p$ and the local guarantee $g$ describe the states and transitions of the program. When checking concurrent properties during the interleaving execution, we rely on the A-G method, in which the assumption $A$ and the guarantee $G$ are used. In one thread, the assumption $A$ gives information of what atomic transitions may be performed by other threads, while the guarantee $G$ holds on every atomic transition performed by the thread itself. So long as the environment (i.e., the collection of all the rest of the threads) satisfies $A$, the thread’s behavior to the environment will satisfy its $G$. Furthermore, every thread should be verified to ensure that the guarantee of any other thread satisfies its assumption.

By this method we could then certify each thread separately without worrying about the rest of the threads. That is how we achieve thread-modular reasoning. And if all threads satisfy their specifications, the following non-interference property results in correct collaboration between threads.

$$
N!(((A_1, G_1, \sigma_1, \overline{R}_1), \ldots, (A_n, G_n, \sigma_n, \overline{R}_n))) \triangleq \\
\forall i \neq j, \sigma_i \neq \sigma_j \land \forall \overline{H}, \overline{H}', (G_i, \overline{R}_i, \overline{H} \rightarrow A_j, \overline{R}_j, \overline{H}')$$

Suppose the program will yield its control by executing $\text{jal yield}$, when $pc$ points to $f$. In order to ensure the yielding safety, the premises of YIELD rule are: (i) the precondition and assumption at the address $f+1$ are the same in each thread; (ii) the local guarantee $g_f$ at the address $f+1$ is equal to the guarantee $G$; the state of yielding thread always satisfies the current precondition if it would satisfy the assumption $A$ after any state transition; (iii) the current thread completes the required state transition specified by the guarantee $G$.

**Example.** We give an example to explain how to certify user multithreaded programs. The example adapted from [7] is a program for unbounded dynamic thread creation as shown in Figure 8. When running, the main thread initialize 100 pieces of data with 0 to 99 respectively, and distributes them to 100 child threads. These child threads add their own data by $\text{one}$ in parallel. To ensure the safety property of the program, each child thread assumes that no other threads will touch its own working data and guarantees that it will not change other threads’ data. The assumptions and guarantees of the main thread and its child threads, defined in Figure 8, reflect these ideas.

Suppose the example code is $C_{\text{ex}}$ and its specification is $\psi_{\text{ex}}$, we have the safety of the code heap $C_{\text{ex}}$ with CMAP inference rules.
neous code specifications down to the OCAP level and then linked with CTL. Program certified in foreign logic systems can be translated and prove system specific rules/axioms as lemmas based on heterogeneous program specification. OCAP rules are ex-

grams, just providing a certified library is insufficient. In

5. Linking CTL to user programs

The main purpose of CTL is to provide a certified runtime for multithreaded user programs. In Section 3.3, we have already certified a thread library CTL and a multithreaded program. As to build safety multithreaded programs, just providing a certified library is insufficient. In this section, we will complete our work by linking the certified multithreaded user programs with CTL.

As stated in Section 1, we choose OCAP as our common certification framework. OCAP uses an extensible and heterogeneous program specification. OCAP rules are expressible enough to embed most existing verification logic systems for low-level code. We can embed a program logic and prove system specific rules/axioms as lemmas based on an interpretation function and OCAP rules. Thus, the safety program certified in foreign logic systems can be translated down to the OCAP level and then linked with CTL.

5.1. OCAP-light

For simplicity, we present a light-weight OCAP (OCAP-light) framework in Figure 9. OCAP-light uses heterogeneous code specifications $\theta$ to support specification languages of both SCAP and CMAP. The code heap specification $\Psi$ is defined as a map from code labels $t$ to their specifications $\theta$. Note that code labels $t$ may be mapped to $\theta_{SCAP}$ or $\theta_{CMAP}$.

In OCAP-light, the code specifications written in different languages should have interaction to form a cooperative system. Accordingly, the interpretation function $[\cdot]$ is used to translate the specification $\theta$ to assertion $a$ which is used at OCAP-light level. Each assertion $a$ is a CiC predicate over $\Psi$ and machine state $S$. With interpretation functions, the specific inference rules of program logics can be proved as lemmas based on a thin layer of Hoare-style inference rules over meta-logic. Then the soundness of a program logic is reduced to the soundness of OCAP-light.

The soundness and correctness theorem of OCAP-light ensures safety, stated in Section 2. It says that any well-formed program will run forever without reaching any undefined state in Figure 2, and any reachable states satisfy the corresponding assertions in $\Psi$.

**Theorem 1 (OCAP-light - Soundness and Correctness).** If $\Psi \vdash ([C,S])$, for all natural number $n$ there exists a $S' = (R',H',pc')$, such that $(C,S) \mapsto^n (C,S')$; and if $pc' \in \Psi$, then $[\Psi(t)] \Psi S'$.

5.2. Embedding SCAP in OCAP-light.

As stated in Section 3.3, $\theta_{SCAP}$ is a pair of predicates $(p,g)$. The predicate $p$ is the precondition and the guarantee $g$ specifies the state at the (function-call) return point and the relationship between the current point and return point. In our TM, the jal instruction is used to perform function
call, while the return action is performed by \texttt{jr r}}. The invariant of the abstract control stack is captured by the predicate \texttt{(WFST)}, which tells us what is a well-form abstract control stack. Certainly, a safe function call \texttt{jal} won’t break the well-formedness of the abstract control stack.

We define the interpretation function of SCAP according to these intuitive ideas in Figure 9. Through the interpretation function, we can build SCAP instruction rules as lemmas in OCAP-light. With these lemmas, safety proof can be constructed directly at OCAP-light level, while still reasoning at the SCAP level. Furthermore, we can prove the soundness of SCAP semantically by this interpretation function.

**Theorem 2 (SCAP - Soundness).**
If $\Psi \vdash_{\text{SCAP}} C : \Psi'$, then $\Psi \vdash C : \Psi'$.

### 5.3. Embedding CMAP in OCAP-light

Like SCAP, an interpretation function of CMAP is also defined in Figure 9. Through the interpretation function, we know that the whole data heap $H$ is separated into two parts, $H_1$ and $H_2$. $H_1$ is for user programs, while $H_2$ is for thread library CTL. $\hat{R}$ is a registers file excluding the registers $r0$, $k0$, $k1$ and ra. The remainder of the interpretation function is a complicated predicate WFTQ. Similar to WFST, WFTQ describes the well-formedness of the thread queue. Upon a well-formed thread queue, the thread library can run the scheduling routine safely. The well-formedness of thread queue is specified by the following invariants:

- each Tcb in the thread queue contains a valid code pointer with code specification ($p, g, A, G$);
- assumptions and guarantees of all the threads are non-interference;
- the precondition of a waiting thread still holds after any state transitions satisfying the assumption, i.e., $\forall R, H, H', p R H \rightarrow A R H H' \rightarrow p R H'$; when calling the routines in CTL, the state satisfies preconditions of all the waiting thread.

Next, we prove the soundness theorem of CMAP to complete the embedding process. Firstly, we prove that the programs certified by CMAP logic system call the yield routine of CTL safely. Informally, the safety proof of \texttt{ctl\_yield()} is divided into two subgoals. One subgoal is to prove that the well-formed state before yielding satisfies the pre-condition of \texttt{ctl\_yield()} $p_r$. The alternative is to prove that the state after the program returns from \texttt{ctl\_yield()} is still well-formed. The difficulties of proving this subgoal come
from the indeterminable transfer of control flow. Fortunately, we could solve these difficulties by knowing that the thread queue satisfies WFTQ. The following lemma specifies the safety of \texttt{ctl\_yield}().

**Lemma 1 (Yielding - Safety).** 
If $\Psi; C \vdash_{\text{CMAP}} \langle \{p, g, A, G\} \rangle f: \text{jal\_yield}$, $\Psi(f+1) = (p, g, A, G)$ then $\langle \{p, g, A, G\} \rangle_y \Rightarrow \langle \{p_y, g_y\} \rangle_y$

By the \texttt{JAL} rule presented in Figure 9 and Lemma 1, we can prove the CMAP \texttt{JAL} rule presented in Figure 7 as lemmas at OCAP-light level.

**Lemma 2 (Yielding - OCAP-light).** 
If $\Psi; C \vdash_{\text{CMAP}} \{\theta_{\text{CMAP}}\} f: \text{jal\_yield}$, then: $\Psi \cup \{\text{yield} \sim \{p_y, g_y\}\}; C \vdash \{\theta_{\text{CMAP}}\} f: \text{jal\_yield}$

Following the same pattern of \texttt{ctl\_yield}(), we can prove that all the routines are safe to call by the user programs. Then, we can have the soundness theorem of CMAP by the CDHP rule in Figure 9 immediately.

**Theorem 3 (CMAP - Soundness).** 
If $\Psi \vdash_{\text{CMAP}} C: \Psi$, then $\Psi \cup \Psi_{\text{CTL}} \vdash C : \Psi$.

**Discussion.** Benefiting from the underlying OCAP-light, we have bridged the two different program logics, CMAP and SCAP. It is possible to embed other concurrent program logics in OCAP-light with the same method presented in this section. The essential step is to define a corresponding interpretation function, which expresses the global invariants of the program logic in another point of view. In the original paper [7], the soundness of CMAP is proved by PROG and DTHRS rules, which is similar to our interpretation function of CMAP actually — all of them do the same job checking whether the thread queue is well-formed. The two rule express the global invariants of the CMAP logic.

As observed, the linkage does not increase the cost of proof construction of user program. The safety proof of linkage is hidden from the programmer writing and certifying multithreaded applications.

The thread queue in the original CMAP is abstract and similar to our Q, while the interpretation function in our framework interprets the abstract queue Q into the concrete one in real TM memory. Between the concrete queue of CTL and the abstract one of the CMAP logic, there is a one-to-one map which makes the bridging possible.

We believe that CTL can be linked with other concurrent certification logics, such as AGL, CSL, etc. Because their thread model is similar to ours, they can also be embedded in OCAP-light by the similar techniques we used.

### 5.4. Example

In Section 3.3, we have proved that CTL is well-formed in SCAP, $\Psi_{\text{CTL}} \vdash_{\text{SCAP}} C_{\text{CTL}} : \Psi_{\text{CTL}}$. By Theorem 2, we have

CTL is well-formed in OCAP-light $\Psi_{\text{CTL}} \vdash C_{\text{CTL}} : \Psi_{\text{CTL}}$. On the other hand, from Section 4, the unbounded thread creation is well-formed in CMAP, $\Psi_{\text{EX}} \vdash_{\text{CMAP}} C_{\text{EX}} : \Psi_{\text{EX}}$. Hence by Theorem 3, the example is well-formed in OCAP-light $\Psi_{\text{EX}} \cup \Psi_{\text{CTL}} \vdash C_{\text{EX}} : \Psi_{\text{EX}}$. Finally, CTL and the example are all well-formed at OCAP-light level. By the CDHP rule of OCAP-light, we can link $C_{\text{EX}}$ with our thread library $C_{\text{CTL}}$ and have the conclusion that:

$$\Psi_{\text{EX}} \cup \Psi_{\text{CTL}} \vdash C_{\text{EX}} \cup C_{\text{CTL}} : \Psi_{\text{EX}} \cup \Psi_{\text{CTL}}$$

From this, a complete multithreaded FPCC package can be constructed by the PROG rule of OCAP-light easily.

### 6. Related work and conclusion

**Thread library.** Ni et al. have certified a thread library named Mth [18], whose aim is quite different from our CTL, they use a machine model that is a strict subset of x86 to ensure that their certified code is runnable on real CPU without any changes, while we concentrate on the simplicity to link our CTL to other certification frameworks with ease, we take an abstract machine model as our platform. But still, our code is MIPS-32 compatible. Mth may be capable of linking to programs certified in other logics, but the linking has not been done yet. The program logic employed by Ni is a variant of XCAP [17], which makes intensive use of general embedded code pointer to support modular reasoning, and results in larger proof size.

**FPCC framework.** Our work is based on the OCAP framework proposed by Feng et al. [6]. In the original OCAP paper, an example was shown to bridge a naive yielding routine to the CCAP logic. Compared to our CTL, the concurrency model of CCAP is rather simple. For example, their model lacks machine context switching and thread management. Our CTL is a big extension to their work, and well illustrates the expressiveness and openness of OCAP.

Chang et al. proposed an open verifier for verifying untrusted code [2]. Their framework produces foundational verifiers using untrusted extensions to customize the safety enforcement mechanism. However, it is unclear whether their extensions support concurrent verification.

**Concurrency verification.** SAGL [5] and concurrent separation logic (CSL) [19] improve the modularity of A-G reasoning method and make the definition of assumptions and guarantees easier. In their machine models, the holding and releasing of locks are primitive operations. So it would be safe to link the programs certified in SAGL or CSL framework with our library just like linking CMAP programs, as long as embedded in OCAP.
Conclusion. We introduce in this paper the design, interfaces and implementation of a certified thread library, which is implemented at assembly level and strictly proved. Dynamic thread management, thread scheduling and synchronization mechanics are also covered, what is more, the modularity of the library endows it with high scalability.

In the open framework OCAP, we show that the thread library can be linked to safe user programs certified in the concurrent certification logic CMAP to form complete multithreaded FPCC packages.

Our long-term goals include the building of a mature thread library with applicative perspectives, as well as the certification of a tiny operating system kernel, whose concurrency model resembles CTL.

Acknowledgments

We would like to thank Zhong Shao and Xinyu Feng for their many inspirational suggestions and comments on earlier drafts of this paper. We would also like to thank Zhaozhong Ni and Hai Fang for help using Coq. Cheng Liu, Yan Guo, Wei Hu and the anonymous reviewers provided many useful suggestions of areas where the paper could be improved.

This work is supported by the National Natural Science Foundation of China under Grant No. 60673126. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of this agency.

References