Formal Reasoning about Concurrent Assembly Code with Reentrant Locks

Ming Fu[1,2] Yu Zhang[1,2] Yong Li[1,2]

[1]*Department of Computer Science & Technology*
University of Science & Technology of China
Hefei, 230027, China
{*fuming, liyong*}*@mail.ustc.edu.cn*

[2]*Software Security Laboratory*
Suzhou Institute for Advanced Study, USTC
SuZhou, 215123, China
yuzhang@ustc.edu.cn

***Abstract*—This paper focuses on the problem of reasoning about concurrent assembly code with reentrant locks. Our verification technique is based on concurrent separation logic (CSL). In CSL, locks are treated as non-reentrant locks and each lock is associated with a resource invariant, the lock-protected resources are obtained and released through acquiring and releasing the lock respectively. In order to accommodate for reentrancy, we introduce some additional notions into our specification language to describe reentrant level for each acquiring and releasing lock operation. Keeping track of the reentrant level for each lock in the pre- and post-conditions enables the program logic to ensure that resources are not re-acquired upon reentrancy, thus resources owned by a thread are prevented from reintroducing in the postcondition. Our framework is fully mechanized. Its soundness has been verified using the Coq proof assistant. We demonstrate the usage of our framework through giving a safety proof of a simple program.**

***Keywords*-reentrant locks, concurrent separation logic, safety, program logic**

I. Introduction

It is difficult to write correct concurrent programs due to potential inter-thread interference at every program point. In order to reduce the complexity of concurrent programming, most popular modern languages – Java and C# provide high-level reentrant locking primitives, which ease concurrent programming. However, it is difficult to use reentrant locks correctly and the incorrect usage can result in nasty concurrent errors like data races or deadlocks. Existing high-level languages do not provide any effective mechanisms to avoid such errors, thus it is important to develop a verification technique for reasoning about concurrent programs with reentrant locks. The reentrant mechanism allows a thread to re-acquire a lock that it already holds. It is important because it eliminates the possibility of a single thread deadlocking itself on a lock that it already holds.

Concurrent separation logic(CSL) [1] is an extension of *separation logic* [2] for reasoning about shared memory race-free concurrent programs. Separation logic is a program logic which is tailored to reason about the heap manipulating programs. In CSL, the shared memory is partitioned and each part is protected by a unique mutual exclusive lock. For each part of the partition, an invariant is assigned to specify its well-formedness. When a thread acquires one of the mutual exclusive locks, it treats the part of shared memory protected by the lock as private. Before releasing the lock, it must ensure that the part of shared memory is well-formed with regard to the corresponding invariant. The ownership of lock-protected shared memory can be dynamically transferred among threads, the verification system ensures that a piece of shared memory is only accessed when the associated lock is held. However, in the invariants of CSL, locks are non-reentrant, we cannot directly apply CSL to reason about concurrent programs with reentrant locks.

In order to adapt CSL to reasoning about concurrent programs with reentrant locks, we build an abstract machine model based on an assembly language with RISC-style instructions and built-in "lock l" and "unlock l" primitives, and introduce additional specification constructs to trace the reentrant level for each lock. Instead of using the high-level parallel language proposed by Hoare [3], we use the assembly language because it has cleaner semantics, which makes our formulation much simpler. For instance, we do not use variables, instead we only use register files and memory. Therefore we can have a quick formulation in Coq [4] without worrying about variable renaming issues. Also we do not have to formalize the complicated syntactic constraints enforced in CSL over shared variables. Another important reason is that our work at low level can be easily applied to generate proof-carrying code (PCC) [5]. It seems unavoidable that the proof rule for acquiring a lock distinguishes between initial acquires and re-acquires. This is needed because it is quite obviously unsound to simply assume the resource invariant after a re-acquire. Thus, a verification system for reentrant locks must keep track of the reentrant level for each lock that the current thread holds in the pre- and post conditions, and we have to enrich our specification language to achieve this requirement. Our system addresses the safety issues at assembly-level as PCC systems do. So we do not need to trust the complicated compilation and optimization and can have a smaller trusted computing base to build executable PCC package for programs using reentrant locks. Furthermore, our formal model for reentrant locks is still general and similar to high-level ones. The verification technique we describe at assembly-level can be lifted up to higher levels. This paper makes the following contributions:

1) As far as we know, this paper first proposes a method to adapt CSL to fitting for reasoning about concurrent assembly code with reentrant locks. We present a program logic for reasoning about properties of concurrent assembly code with reentrant locks, and we prove it sound with respect to the semantics of reentrant locks.
2) We implement our framework using the Coq proof assistant, and prove an example under the framework. The result shows that the adapted inference rules can be easily applied to verify the concurrent assembly code with reentrant locks.

The rest of this paper is organized as follows: In section II, we explain the CSL and its limitation for verifying reentrant locks. We describe the abstract machine we model and the program logic for reasoning about concurrent assembly code with reentrant locks in section III. Section IV presents an example that are written and proved under our framework. Finally we discuss the related work and conclusion in section V and VI.

II. Preliminaries

We give a brief description of separation logic. A more careful treatment is in [1] and [6]. A simplified syntax for separation logic is shown in Fig. 1.

Here we briefly demonstrate the logical semantics for each construct in the syntax. Both A and B are assertions that describe the heap. $\mathtt{l} \mapsto v$ holds if the heap consists entirely of the binding of location $\mathtt{l}$ to value v. $\mathtt{emp}$ holds only on the empty heap. $A * B$ holds if the heap can be split into two disjoint parts such that A holds on one and B on the other. $A \wedge B$ holds if both A and B hold on the entire heap. $A \vee B$ holds if either A or B holds on the heap. $\exists\ x.\ B$ holds if there exists an x that B holds on the heap. $\forall\ x.\ A$ holds on a heap that satisfies A for all x.

$$\begin{array}{rl} A, B\ ::= & \mathtt{l} \mapsto v \mid \mathsf{emp} \mid A * B \mid A \wedge B \mid A \vee B \\ & \mid \exists\ x.\ B \mid \forall\ x.\ A \end{array}$$

Figure 1. Syntax of Separation Logic

The *frame property* of separation logic requires that if a program does not go wrong in a particular state with heap $\mathbb{H}$, then it will not go wrong in a larger state with heap $\mathbb{H} \uplus \mathbb{H}'$ (the notion "$\uplus$" is used to merge two disjoint heaps into a larger one, we give its formal definition in Fig. 6); the effect will still be taken on $\mathbb{H}$, leaving the added heap $\mathbb{H}'$ completely unaffected. Thus the separation conforms to the following frame rule:

$$\frac{\{Q\}C\{R\}}{\{P * Q\}C\{P * R\}}$$

(no variable occurring free in P is modified by C)

If C cannot modify the variables of P, and if the heap it manipulates is disjoint from that of P, then we can reason about C and its effect separately from P.

CSL introduces the concurrency rule based on separation logic for reasoning about concurrent programs. The concurrency rule given below

$$\frac{\{Q_1\}C_1\{R_1\} \quad \{Q_2\}C_2\{R_2\}}{\{Q_1 * Q_2\}C_1 \| C_2\{R_1 * R_2\}}$$

describes how concurrent threads with disjoint heap resources can be treated separately. As a concurrent program executes, heap resources must remain separated but the separation need not be fixed : the ownership can be transferred among threads through locking operation. The rule below is used to deal with the non-reentrant locks for transferring the ownership of shared resources.

$$\frac{I \text{ is a resource invariant associated with a lock } \ l}{\Gamma, l \rightsquigarrow I \ \vdash \{\mathsf{emp}\}\ \mathsf{lock}\ l\{I\}}$$

However, we can not directly apply this rule to reason about concurrent programs with reentrant locks. The main problem is that a verification system for reentrant locks has to distinguish between initial lock entries and reentries, because only after initial entries is it sound to assume a lock's resource invariant. This means that initial lock entries need precondition requiring that the current thread has not yet held the acquired lock. In Fig. 2, a simple code sequence is made up of two consecutive statements that acquire the same lock l. Both the first and the second acquiring lock l operations lead to obtain additional resource satisfying invariant I. According to the frame rule and the above rule , the second acquiring lock operation requires that the postcondition be $I * I$. However separation logic treats $I * I$ as a false assertion, and this leads to incorrect verification. The following sections show our technique for solving this problem and adapting CSL to reasoning about concurrent programs with reentrant locks.

```
        {emp}
(1)     lock l ;
          {I}       (I is lock l 's resource invariant )
(2)     lock l ;
        {I * I}     (Wrong!!!)
          ...
```

Figure 2. CSL does not Support Reentrant Locks

III. The Framework

A. Abstract Machine

Fig. 3 defines the abstract machine and the syntax of an assembly language. We extend CAP [7], [8] by adding built-in primitives "lock l" and "unlock l" for reentrant locks. The whole world $\mathbb{W}$ consists of a code heap $\mathbb{C}$, a shared data heap

(*World*)	$\mathbb{W}$	::=	$(\mathbb{C},\mathbb{H},\mathbb{TS},\mathbb{L})$
(*ThreadSet*)	$\mathbb{TS}$	::=	$(\mathbb{T}_1,\ldots,\mathbb{T}_n)$
(*ThreadState*)	$\mathbb{S}$	:=	$(\mathbb{H},\mathbb{T},\mathbb{L})$
(*Thread*)	$\mathbb{T}_i$	::=	$(\mathbb{R},\mathtt{pc},\mathtt{tid})$
(*ThrdID*)	$\mathtt{tid}$	::=	m (*nat nums*, *and* $m>0$)
(*CodeHeap*)	$\mathbb{C}$	::=	$(\mathtt{f}\rightsquigarrow\iota)^*$
(*Heap*)	$\mathbb{H}$	::=	$\{\mathtt{l}\rightsquigarrow\mathtt{w}\}^*$
(*LockMap*)	$\mathbb{L}$	::=	$\{l\rightsquigarrow(\mathtt{tid},\mathtt{n})\}^*$
(*ReentrantLevel*)	$\mathtt{n}$	::=	i(*nat nums*, *and* $i>0$)
(*RegFile*)	$\mathbb{R}$	::=	$\{\mathtt{r}\rightsquigarrow\mathtt{w}\}^*$
(*Register*)	$\mathtt{r}$	::=	$\mathtt{r}_0 \mid\ldots\mid \mathtt{r}_{31}$
(*Labels*)	$\mathtt{f},\mathtt{l},\mathtt{pc}$	::=	i (*nat nums*)
(*Locks*)	l	::=	i (*nat nums*)
(*Word*)	$\mathtt{w}$	::=	i (*nat nums*)
(*Instr*)	ι	::=	add $\mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t$ $\mid$ addi $\mathtt{r}_d,\mathtt{r}_s,\mathtt{w}$
		$\mid$	sub $\mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t$ $\mid$ ld $\mathtt{r}_d,\mathtt{w}(\mathtt{r}_s)$
		$\mid$	st $\mathtt{r}_d,\mathtt{w}(\mathtt{r}_s)$ $\mid$ beq $\mathtt{r}_s,\mathtt{r}_t,\mathtt{f}$
		$\mid$	lock l $\mid$ unlock l
(*InstrSeq*)	$\mathbb{I}$	::=	$\iota;\mathbb{I}$ $\mid$ j $\mathtt{f}$ $\mid$ jr $\mathtt{r}_s$

Figure 3. The Abstract Machine

$\mathbb{H}$, a thread set $\mathbb{TS}$ which contains n threads $(\mathbb{T}_1,\ldots,\mathbb{T}_n)$ and a shared lock mapping $\mathbb{L}$.

The code heap $\mathbb{C}$ is a partial mapping from code labels to instructions. The global shared heap $\mathbb{H}$ is modeled as a finite partial mapping from heap locations $\mathtt{l}$ (natural numbers) to word values $\mathtt{w}$ (natural numbers). The locking map $\mathbb{L}$ is a finite mapping from reentrant locks to lock value pairs "$(\mathtt{tid},\mathtt{n})$", where the integer $\mathtt{tid}$ identifies the thread holding the lock exclusively and the integer $\mathtt{n}$ is the reentrant level counting how often it currently holds the lock.

The abstract machine has a fixed number of threads. Each thread $\mathbb{T}_i$ contains a register file $\mathbb{R}$, a program counter $\mathtt{pc}$ and its thread identifier $\mathtt{tid}$. Here we allow each thread to have its own register file and program counter, which is consistent with most implementation of thread library where the register file is saved to the execution context when a thread is preempted. The register file $\mathbb{R}$ is represented as a total function from registers to words. Each thread's program counter $\mathtt{pc}$ points to its current command in a shared code heap $\mathbb{C}$. The set of instructions we present here are the commonly used subset in RISC machines with additional reentrant "lock l" and "unlock l" primitives for synchronization.

We define the instruction sequence $\mathbb{I}$ as a sequence of sequential instructions ending with jump or return instructions. $\mathbb{C}[\mathtt{pc}]$ extracts an instruction sequence starting from $\mathtt{pc}$ in $\mathbb{C}$, as defined in Fig. 4. $(F\{a\rightsquigarrow b\})(x)$ is used to formalize memory update in our operational semantics. Macros $\mathbb{S}\,|_{\mathbb{H}'}$ and $\mathbb{S}\,|_{\mathbb{L}'}$ are defined for constructing thread states by replacing the heap and the lock set respectively.

$$\mathbb{C}[\mathtt{pc}] \overset{\text{def}}{=} \begin{cases} \iota & \iota=\mathbb{C}(\mathtt{pc}) \text{ and } \iota = \text{j } \mathtt{f} \text{ or jr } \mathtt{r}_s \\ \iota;\mathbb{I} & \iota=\mathbb{C}(\mathtt{pc}) \text{ and } \mathbb{I}=\mathbb{C}[\mathtt{pc+1}] \end{cases}$$

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

$$\mathbb{S}\,|_{\mathbb{H}'} \overset{\text{def}}{=} (\mathbb{H}',\mathbb{S}.\mathbb{T},\mathbb{S}.\mathbb{L})$$

$$\mathbb{S}\,|_{\mathbb{L}'} \overset{\text{def}}{=} (\mathbb{S}.\mathbb{H},\mathbb{S}.\mathbb{T},\mathbb{L}')$$

Figure 4. Definition of Representations

B. *Operational Semantics*

The operational semantics of each instruction is defined in Fig. 5. The relation $\mathtt{NextS}_\iota$ shows the transition of thread states by executing instruction ι. The operational semantics for most instructions are quite straightforward. Note the execution of instruction for acquiring locks. It allows a lock to be re-acquired by one thread and does not lead to deadlock. There exist three different cases for executing lock l: when l is not in the domain of $\mathbb{L}$ (means that l is free), the current thread $\mathtt{tid}$ can exclusively and successfully acquire the lock l, and set lock l with pair value $(\mathtt{tid},1)$. $\mathtt{tid}$ denotes that lock l is held by thread $\mathtt{tid}$ and the reentrant level "1" shows that the current program point is at the initial lock entry of the lock l. Shared resource can be obtained by the thread through the initial lock acquiring. When l is in the domain of $\mathbb{L}$ and held by the current thread $\mathtt{tid}$, thread $\mathtt{tid}$ tries to re-acquired a held lock. Non-reentrant locking mechanism makes the current thread block and leads to deadlock, while our model avoids deadlock through setting the reentrant level in the lock pair value with the increment of "1". When l is held by the other thread, the current thread blocks. The semantics for releasing locks is straightforward, the reentrant level makes acquiring and releasing operation on the same lock keep in pair.

Fig. 5 also defines $(\mathbb{C},\mathbb{S})\rightsquigarrow(\mathbb{C},\mathbb{S}')$ and $(\mathbb{W}\longmapsto\mathbb{W}')$ for the thread execution and the whole world execution respectively. Note that relation $(\mathbb{C},\mathbb{S})\rightsquigarrow(\mathbb{C},\mathbb{S}')$ is deterministic but our semantics of the abstract machine $(\mathbb{W}\longmapsto\mathbb{W}')$ is not deterministic: the state transition may be made by executing any thread in $\mathbb{W}$. Also, given a $\mathbb{W}$, there may not always exist a $\mathbb{W}'$ such that $(\mathbb{W}\longmapsto\mathbb{W}')$ holds. If there is no such $\mathbb{W}'$, we say the program gets stuck at $\mathbb{W}$. One important goal of our program logic is to show that verified programs never get stuck.

C. *Program Logic*

1) Assertion Language: Fig. 6 shows the syntax and semantics of the assertion language. We use the predicate $\mathtt{m}$ over a heap and separation logic connectors $*$ in our assertion language. The assertion $\mathtt{a}$ is a predicate over a thread state.

Most of the definitions are simple and straightforward. Here we explain some special ones. The assertion "$\mathtt{l}\mapsto v$"

$\mathtt{NextS}_\iota\ \mathbb{S}\ \mathbb{S}'$ where $\mathbb{S} = (\mathbb{H},(\mathbb{R},\mathtt{pc},\mathtt{tid}),\mathbb{L})$	
if $\iota =$	$\mathbb{S}' =$
add $\mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t$	$(\mathbb{H},(\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s)+\mathbb{R}(\mathtt{r}_t)\},\mathtt{pc+1},\mathtt{tid}),\mathbb{L})$
addi $\mathtt{r}_d,\mathtt{r}_s,\mathtt{w}$	$(\mathbb{H},(\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s)+\mathtt{w}\},\mathtt{pc+1},\mathtt{tid}),\mathbb{L})$
sub $\mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t$	$(\mathbb{H},(\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s)-\mathbb{R}(\mathtt{r}_t)\},\mathtt{pc+1},\mathtt{tid}),\mathbb{L})$
ld $\mathtt{r}_d,\mathtt{w}(\mathtt{r}_s)$	$(\mathbb{H},(\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{H}(\mathbb{R}(\mathtt{r}_s)+\mathtt{w})\},\mathtt{pc+1},\mathtt{tid}),\mathbb{L})$ if $\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \in \mathrm{dom}(\mathbb{H})$
st $\mathtt{r}_d,\mathtt{w}(\mathtt{r}_s)$	$(\mathbb{H}\{\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \rightsquigarrow \mathbb{R}(\mathtt{r}_d)\},(\mathbb{R},\mathtt{pc+1},\mathtt{tid}),\mathbb{L})$ if $\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \in \mathrm{dom}(\mathbb{H})$
lock l	$(\mathbb{H},(\mathbb{R},\mathtt{pc+1},\mathtt{tid}),\mathbb{L}\{l \rightsquigarrow (\mathtt{tid},1)\})$ if $l \notin \mathrm{dom}(\mathbb{L})$ $(\mathbb{H},(\mathbb{R},\mathtt{pc+1},\mathtt{tid}),\mathbb{L}\{l \rightsquigarrow (\mathtt{tid},n+1)\})$ if $\mathbb{L}(l) = (\mathtt{tid},n)$ $(\mathbb{H},(\mathbb{R},\mathtt{pc},\mathtt{tid}),\mathbb{L})$ otherwise
unlock l	$(\mathbb{H},(\mathbb{R},\mathtt{pc+1},\mathtt{tid}),\mathbb{L}\{l \rightsquigarrow (\mathtt{tid},n-1)\})$ if $\mathbb{L}(l) = (\mathtt{tid},n) \wedge n > 1$ $(\mathbb{H},(\mathbb{R},\mathtt{pc+1},\mathtt{tid}),\mathbb{L}/\{l\})$ if $\mathbb{L}(l) = (\mathtt{tid},n) \wedge n = 1$
j $\mathtt{f}$	$(\mathbb{H},(\mathbb{R},\mathtt{f},\mathtt{tid}),\mathbb{L})$
jr $\mathtt{r}_s$	$(\mathbb{H},(\mathbb{R},\mathbb{R}(\mathtt{r}_s),\mathtt{tid}),\mathbb{L})$
beq $\mathtt{r}_s,\mathtt{r}_t,\mathtt{f}$	$(\mathbb{H},(\mathbb{R},\mathtt{f},\mathtt{tid}),\mathbb{L})$ if $\mathbb{R}(\mathtt{r}_s) = \mathbb{R}(\mathtt{r}_t)$ $(\mathbb{H},(\mathbb{R},\mathtt{pc+1},\mathtt{tid}),\mathbb{L})$ if $\mathbb{R}(\mathtt{r}_s) \neq \mathbb{R}(\mathtt{r}_t)$

$$\frac{\iota = \mathbb{C}(\mathtt{pc}) \quad \mathtt{NextS}_\iota\ \mathbb{S}\ \mathbb{S}'}{(\mathbb{C},\mathbb{S}) \rightsquigarrow (\mathbb{C},\mathbb{S}')}\ \text{THREADSTEP}$$

$$\frac{\exists \mathbb{T}_k.\mathbb{T}_k \in \mathbb{TS} \wedge (\mathbb{C},(\mathbb{H},\mathbb{T}_k,\mathbb{L})) \rightsquigarrow (\mathbb{C},(\mathbb{H}',\mathbb{T}_k',\mathbb{L}'))}{(\mathbb{C},\mathbb{H},\mathbb{TS},\mathbb{L}) \longmapsto (\mathbb{C},\mathbb{H}',(\mathbb{T}_1\ldots,\mathbb{T}_k',\ldots\mathbb{T}_n),\mathbb{L}')}\ \text{WORLDSTEP}$$

Figure 5. Operational Semantics

(*ThrdStatePred*)	$\mathtt{a}$	$\in$	*ThreadState* $\rightarrow$ *Prop*
(*HeapPred*)	$\mathtt{m}$	$\in$	*Heap* $\rightarrow$ *Prop*

$$\begin{aligned}
\mathtt{m} \ ::=&\ \mathtt{l} \mapsto v \mid \mathtt{true} \mid \mathsf{emp} \mid \mathtt{m}_1 * \mathtt{m}_2 \\
&\mid \mathtt{m}_1 \wedge \mathtt{m}_2 \mid \mathtt{m}_1 \vee \mathtt{m}_2 \mid \exists\, x.\ \mathtt{m} \mid \forall\, x.\ \mathtt{m} \\
\mathtt{a} \ ::=&\ \lfloor \mathtt{m} \rfloor \mid \mathsf{own}_k(l,n) \mid \mathtt{r} = \mathtt{v} \\
&\mid \mathtt{a}_1 \wedge \mathtt{a}_2 \mid \mathtt{a}_1 \vee \mathtt{a}_2 \mid \exists\, x.\ \mathtt{a} \mid \forall\, x.\ \mathtt{a}
\end{aligned}$$

$$\begin{aligned}
\mathtt{true} &\overset{\text{def}}{=} \lambda\mathbb{H}.\mathtt{True} \\
\mathsf{emp} &\overset{\text{def}}{=} \lambda\mathbb{H}.\mathrm{dom}(\mathbb{H}) = \emptyset \\
\mathbb{H}_1 \perp \mathbb{H}_2 &\overset{\text{def}}{=} \mathrm{dom}(\mathbb{H}_1) \cap \mathrm{dom}(\mathbb{H}_2) = \emptyset \\
\mathtt{l} \mapsto v &\overset{\text{def}}{=} \lambda\mathbb{H}.\mathbb{H} = \{\mathtt{l} \rightsquigarrow v\} \\
\mathbb{H}_1 \uplus \mathbb{H}_2 &\overset{\text{def}}{=} \begin{cases} \mathbb{H}_1 \cup \mathbb{H}_2 & \text{if } \mathbb{H}_1 \perp \mathbb{H}_2 \\ \mathit{undefined} & \mathtt{otherwise} \end{cases} \\
\mathtt{m}_1 * \mathtt{m}_2 &\overset{\text{def}}{=} \lambda\mathbb{H}.\exists\mathbb{H}_1,\mathbb{H}_2.(\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{H}) \wedge \mathtt{m}_1\mathbb{H}_1 \wedge \mathtt{m}_2\mathbb{H}_2 \\
\lfloor \mathtt{m} \rfloor &\overset{\text{def}}{=} \lambda\mathbb{S}.\mathtt{m}\ \mathbb{S}.\mathbb{H} \\
\mathsf{own}_k(l,n) &\overset{\text{def}}{=} \lambda\mathbb{S}.(k = \mathbb{S}.\mathbb{T}.\mathtt{tid}) \wedge \mathbb{S}.\mathbb{L}(l) = (k,n) \\
\mathtt{r} = \mathtt{v} &\overset{\text{def}}{=} \lambda\mathbb{S}.\mathbb{S}.\mathbb{T}.\mathbb{R}(\mathtt{r}) = \mathtt{v}
\end{aligned}$$

Figure 6. Syntax and Semantics of the Assertion Language

holds only if the heap has only one cell at location $\mathtt{l}$ containing value v. $\mathtt{m}_1 * \mathtt{m}_2$ means the heap can be split into two disjoint parts, and $\mathtt{m}_1$ and $\mathtt{m}_2$ hold over each of them respectively. $\lfloor \mathtt{m} \rfloor$ means predicate over a thread state containing a heap satisfying $\mathtt{m}$, we use this syntax to lift predicates over heap to assertions specifying a thread state. Predicate $\mathsf{own}_k(l,n)$ is used to specify that l is held by the thread k with corresponding reentrant level n. Here, we omit the semantics of some straightforward connectors, such as $\wedge, \vee, etc$.

2) Program Specification: We use the mechanized *meta-logic* implemented in the Coq proof assistant as our specification language. The logic corresponds to higher-order logic with inductive definitions.

(*WorldSpec*)	ϕ	$:=$	$([\psi_1,\ldots,\psi_n],\Gamma)$
(*CdHpspec*)	ψ	$:=$	$\{(\mathtt{f}_1,\mathtt{a}_1),\ldots,(\mathtt{f}_n,\mathtt{a}_n)\}$
(*LockINV*)	Γ	$:=$	$\{l \rightsquigarrow \mathtt{m}\}^*$

(*Well-formed World*)	$\phi,[\mathtt{a}_1,\ldots,\mathtt{a}_n] \vdash (\mathbb{C},\mathbb{H},\mathbb{TS},\mathbb{L})$
(*Well-formed Thread*)	$\psi,\Gamma \vdash \{\mathtt{a}\}(\mathbb{C},\mathbb{H},\mathbb{T},\mathbb{L})$
(*Well-formed Code Heap*)	$\psi,\Gamma \vdash \mathbb{C} : \psi'$
(*Well-formed Instr. Seq.*)	$\psi,\Gamma \vdash \{\mathtt{a}\}\mathtt{pc} : \iota;\mathbb{I}$
(*Well-formed Instruction*)	$\psi,\Gamma \vdash \{\mathtt{a}\}\mathtt{pc} : \iota$

Figure 7. Specification Constructs for the Program Logic

The specification constructs of our logic are presented in Fig. 7. The world specification ϕ contains a collection of code heap specifications for each thread and a specification

Γ for lock-protected heap. Code heap specification ψ maps a code label to a predicate $\mathtt{a}$ over thread state $\mathbb{S}$ as the precondition of corresponding instruction sequence. The specification Γ of a lock-protected heap maps a lock to an invariant $\mathtt{m}$ specifying the shared heap.

The last five judgments are used to define the well-formed world, well-formed thread, well-formed code heap, well-formed instruction sequence and well-formed instruction respectively. The inference rules for these judgments will be presented in the following subsection.

$$\begin{aligned}
\mathtt{a}_1 \Rightarrow \mathtt{a}_2 &\stackrel{\text{def}}{=} \lambda\mathbb{S}.\mathtt{a}_1\ \mathbb{S} \rightarrow \mathtt{a}_2\ \mathbb{S} \\
\mathtt{a} * \mathtt{m} &\stackrel{\text{def}}{=} \lambda\mathbb{S}.\exists\mathbb{H}_1,\mathbb{H}_2,(\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{S}.\mathbb{H}) \wedge \mathtt{a}\ \mathbb{S}|_{\mathbb{H}_1} \wedge \mathtt{m}\ \mathbb{H}_2 \\
\psi \circ \mathtt{NextS}_\iota &\stackrel{\text{def}}{=} \lambda\mathbb{S}.\exists\mathbb{S}',\mathtt{NextS}_\iota\mathbb{S}\mathbb{S}' \wedge \psi(\mathbb{S}'.\mathbb{T}.\mathtt{pc})\ \mathbb{S}' \\
\forall_* x \in S.\ P(x) &\stackrel{\text{def}}{=} \begin{cases} \mathsf{emp} & \text{if } S = \emptyset \\ P(x_i) * \forall_* x \in S'.\ P(x) & \text{if } S = S' \uplus \{x_i\} \end{cases}
\end{aligned}$$

Figure 8. Auxiliary Definition

3) Inference Rules: The inference rules for a program and instructions are presented in Fig. 9.

A world is well-formed with regard to a world specification ϕ and thread state predicates $\mathtt{a}_1,\ldots,\mathtt{a}_n$ for each thread when the following conditions hold:

- There is a partition of the global heap into $n+1$ disjoint parts, where the shared heap $\mathbb{H}_s$ satisfies the invariants specified in Γ and $\mathbb{H}_1,\ldots,\mathbb{H}_n$ satisfy each thread state predicate $\mathtt{a}_k$ respectively. As in O'Hearn's original work on CSL [1], we also require invariants specified in Γ to be precise, denoted as $\mathtt{Precise}(\Gamma)$ defined as below. Every thread of the world is required to be well-formed. Thus our system support thread-modular verification by decomposing the verification of multi-threaded program into that of its component threads.

$$\begin{aligned}
\mathtt{Precise}(\mathtt{m}) &\stackrel{\text{def}}{=} \forall\mathbb{H}_1,\mathbb{H}_2,\mathbb{H}.\mathbb{H}_1 \subseteq \mathbb{H} \rightarrow \mathbb{H}_2 \subseteq \mathbb{H} \rightarrow \\
&\qquad \mathtt{m}\ \mathbb{H}_1 \wedge \mathtt{m}\ \mathbb{H}_2 \rightarrow \mathbb{H}_1 = \mathbb{H}_2 \\
\mathtt{Precise}(\Gamma) &\stackrel{\text{def}}{=} \forall l \in \mathsf{dom}(\Gamma).\ \mathtt{Precise}(\Gamma(l))
\end{aligned}$$

- The shared state $(\mathbb{H}_s, \_, \mathbb{L})$ satisfies the predicate $\mathtt{a}_\Gamma$, which is defined below. The definition of $\mathtt{a}_\Gamma$ is the separating conjunction of invariants assigned to the locks which are free (not in the domain of the global $\mathbb{L}$). It ensures that the shared heap are well-formed outside critical region. Here $\forall_*$ is an indexed, finitely iterated separating conjunction, which is formalized in Fig. 8.

$$\mathtt{a}_\Gamma \stackrel{\text{def}}{=} \lambda\mathbb{S}.(\forall_* l \in \{l \mid l \notin \mathsf{dom}(\mathbb{S}.\mathbb{L})\}.\ \Gamma(l))\mathbb{S}.\mathbb{H}$$

A thread is well-formed if the current thread state satisfies the precondition $\mathtt{a}$ and both the code heap and the instruction sequence are required to be well-formed. Since $\mathtt{a}$ only specifies the private resource, we use "filter" operator "$\mathbb{L}|_{\mathtt{tid}}$" formalized below to prevent $\mathtt{a}$ from having access to the ownership information of locks not held by the current thread.

$$(\mathbb{L}|_{\mathtt{tid}})(l) \stackrel{\text{def}}{=} \begin{cases} (\mathtt{tid}, n) & \text{if } \mathbb{L}(l) = (\mathtt{tid}, n) \\ \mathtt{undefined} & \mathtt{otherwise} \end{cases}$$

The rule CDHP shows that a code heap is well-formed only if each instruction sequence specified in ψ' is well-formed with respect to the imported interfaces specified with ψ and the lock specification Γ.

The rule INSQ shows that an instruction sequence is well-formed if it is composed of a single instruction ι and another instruction sequence $\mathbb{I}$, both of which are well-formed.

A well-formed instructions includes the following cases with the order of Fig. 9.

- The rule INSN - If the instruction ι is not $\mathsf{lock}\ l$ or $\mathsf{unlock}\ l$, it can execute for all thread states specified by the current thread state predicate $\mathtt{a}$, and the new modified thread state must satisfy the thread state predicate for the target address of instruction ι given by ψ. We use $\psi \circ \mathtt{NextS}_\iota$ defined in Fig. 8 to specify the new modified state generated by executing instruction ι.
- The rule LOCK - We have a unified rule for reasoning about instruction $\mathsf{lock}\ l$ which may be executed at either the initial entry or reentry. The reentrant locks are handy in the presence of polymorphism, i.e. where a given routine that executes lock is called both in a context where the lock is free and where the lock was previously acquired. In that sense, whether the locking operation happens at the initial entry or reentry cannot be established statically, and the unified rule LOCK can support reasoning about the either case automatically. The rule applies when lock l is acquired at the initial entry or reentry. The thread predicate $((\mathtt{En}^l_{\mathtt{lk}} \wedge \mathtt{m}) \vee (\mathtt{En}^l_{\mathtt{relk}} \wedge \mathtt{emp}))$ is used to enforces the ownership transfer under the following two cases:
  - If the current state satisfies the predicate $\mathtt{En}^l_{\mathtt{lk}}$ (defined in Fig. 10) which ensures the lock is free and enables safely locking operation at the initial entry, we can carry the knowledge $\mathtt{m}$ in the postcondition given by ψ at the target address of instruction ι. The global invariant ensures that the part of heap protected by l satisfies the invariant $\mathtt{m}$.
  - If the current state satisfies the predicate $\mathtt{En}^l_{\mathtt{relk}}$ (defined in Fig. 10) which ensures the lock is held by itself and enables safely locking operation at the reentry, we can use the empty heap predicate emp to represent nothing is acquired at the reentry. The part of heap protected by l will not be reintroduced into the result state.
- The rule UNLOCK is similar with the rule LOCK, we use the predicate $((\mathtt{En}^l_{\mathtt{unlk}} \wedge \mathtt{m}) \vee (\mathtt{En}^l_{\mathtt{reunlk}} \wedge \mathtt{emp}))$ to represent two different cases, one is that the invariant gets established and the lock l's current reentrancy level

$\boxed{\phi, [\mathtt{a}_1, \ldots, \mathtt{a}_n] \vdash (\mathbb{C}, \mathbb{H}, \mathbb{TS}, \mathbb{L})}$ ***(Well-formed World)***

$$\frac{\phi = ([\psi_1, \ldots, \psi_n], \Gamma) \quad \mathbb{H} = \mathbb{H}_s \uplus \mathbb{H}_1 \uplus \ldots \uplus \mathbb{H}_n \quad \mathtt{Precise}(\Gamma) \qquad \mathtt{a}_\Gamma(\mathbb{H}_s, \_, \mathbb{L}) \quad \psi_k, \Gamma \vdash \{\mathtt{a}_k\}(\mathbb{C}, \mathbb{H}_k, \mathbb{T}_k, \mathbb{L}) \;\; \mathtt{for\ all}\; k \in \{1, \ldots, n\}}{\phi.[\mathtt{a}_1, \ldots, \mathtt{a}_n] \vdash (\mathbb{C}, \mathbb{H}, [\mathbb{T}_1, \ldots, \mathbb{T}_n], \mathbb{L})} \text{ WORLD}$$

$\boxed{\psi, \Gamma \vdash \{\mathtt{a}\}(\mathbb{C}, \mathbb{H}, \mathbb{T}, \mathbb{L})}$ ***(Well-formed Thread)***

$$\frac{\mathtt{a}\ (\mathbb{H}, (\mathbb{R}, \mathtt{pc}, \mathtt{tid}), \mathbb{L}|_{\mathtt{tid}}) \quad \psi, \Gamma \vdash \mathbb{C} : \psi \quad \psi, \Gamma \vdash \{\mathtt{a}\}\mathtt{pc} : \mathbb{C}[\mathtt{pc}]}{\psi, \Gamma \vdash \{\mathtt{a}\}(\mathbb{C}, \mathbb{H}, (\mathbb{R}, \mathtt{pc}, \mathtt{tid}), \mathbb{L})} \text{ THRD}$$

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

$$\frac{\forall(\mathtt{pc}, \mathtt{a}) \in \psi' : \quad \psi, \Gamma \vdash \{\mathtt{a}\}\mathtt{pc} : \mathbb{C}[\mathtt{pc}]}{\psi, \Gamma \vdash \mathbb{C} : \psi'} \text{ CDHP}$$

$\boxed{\psi, \Gamma \vdash \{\mathtt{a}\}\mathtt{pc} : \mathbb{I}}$ ***(Well-formed Instr. Sequence)***

$$\frac{\psi, \Gamma \vdash \{\mathtt{a}'\}\mathtt{pc+1} : \mathbb{I} \quad \psi\{\mathtt{pc+1} \rightsquigarrow \mathtt{a}'\}, \Gamma \vdash \{\mathtt{a}\}\mathtt{pc} : \iota}{\psi, \Gamma \vdash \{\mathtt{a}\}\mathtt{pc} : \iota; \mathbb{I}} \text{ INSQ}$$

$\boxed{\psi, \Gamma \vdash \{\mathtt{a}\}\mathtt{pc} : \iota}$ ***(Well-formed Instruction)***

$$\frac{\iota \notin \{\mathsf{lock}\ l, \mathsf{unlock}\ l\} \quad \mathtt{a} \Rightarrow \psi \circ \mathtt{NextS}_\iota}{\psi, \Gamma \vdash \{\mathtt{a}\}\mathtt{pc} : \iota} \text{ INSN}$$

$$\frac{\mathtt{a} * ((\mathtt{En}^l_{\mathtt{lk}} \wedge \mathtt{m}) \vee (\mathtt{En}^l_{\mathtt{relk}} \wedge \mathtt{emp})) \Rightarrow \psi \circ \mathtt{NextS}_{\mathsf{lock}\ l}}{\psi, \Gamma\{l \rightsquigarrow \mathtt{m}\} \vdash \{\mathtt{a}\}\mathtt{pc} : \mathsf{lock}\ l} \text{ LOCK}$$

$$\frac{\mathtt{a} \Rightarrow (\psi \circ \mathtt{NextS}_{\mathsf{unlock}\ l}) * ((\mathtt{En}^l_{\mathtt{unlk}} \wedge \mathtt{m}) \vee (\mathtt{En}^l_{\mathtt{reunlk}} \wedge \mathtt{emp}))}{\psi, \Gamma\{l \rightsquigarrow \mathtt{m}\} \vdash \{\mathtt{a}\}\mathtt{pc} : \mathsf{unlock}\ l} \text{ UNLOCK}$$

Figure 9. Inference Rules

is 1; the other is that the specified heap is empty and the lock l's current reentrancy level is bigger than 1. The predicate enforces that the ownership of the well-formed shared heap protected by the lock l only be transferred from private part to the shared part at the last releasing and the middle unlocking operations do not change the domain of thread private heap. The predicate $\mathtt{En}^l_{\mathtt{unlk}}$ and $\mathtt{En}^l_{\mathtt{reunlk}}$ defined in Fig. 10 enables safely unlocking action taken on the current state.

$$\begin{aligned}
\mathtt{En}^l_{\mathtt{lk}} &\overset{\text{def}}{=} \lambda\mathbb{S}.l \notin \mathsf{dom}(\mathbb{S}.\mathbb{L}) \\
\mathtt{En}^l_{\mathtt{relk}} &\overset{\text{def}}{=} \lambda\mathbb{S}.(\mathbb{S}.\mathbb{T}.\mathtt{tid}, \_) = \mathbb{S}.\mathbb{L}(l) \\
\mathtt{En}^l_{\mathtt{unlk}} &\overset{\text{def}}{=} \lambda\mathbb{S}.(\mathbb{S}.\mathbb{T}.\mathtt{tid}, 1) = \mathbb{S}.\mathbb{L}(l) \\
\mathtt{En}^l_{\mathtt{reunlk}} &\overset{\text{def}}{=} \lambda\mathbb{S}.\exists n, (\mathbb{S}.\mathbb{T}.\mathtt{tid}, n) = \mathbb{S}.\mathbb{L}(l) \wedge n > 1
\end{aligned}$$

Figure 10. Predicates for Enabling Instructions

4) Soundness: The soundness of these inference rules with respect to the operational semantics of the abstract machine is proved following the syntactic approach [9]. From the ”progress” and ”preservation” lemmas, we can guarantee that given a well-formed program under the compatible preconditions, the current instruction sequence will be able to execute without getting ”stuck”. The soundness of our framework is formally stated as Theorem III.3.

Lemma III.1 (Progress) *For any world* $\mathbb{W} = (\mathbb{C}, \mathbb{H}, (\mathbb{T}_1, \ldots, \mathbb{T}_n), \mathbb{L})$*, and if* $\psi, [a_1, \ldots, a_n] \vdash \mathbb{W}$*, then for any thread* $\mathbb{T}_k$*, there exist* $\mathbb{H}', \mathbb{T}'_k$ *and* $\mathbb{L}'$ *such that* $(\mathbb{C}, (\mathbb{H}, \mathbb{T}_k, \mathbb{L})) \rightsquigarrow (\mathbb{C}, (\mathbb{H}', \mathbb{T}'_k, \mathbb{L}'))$.

Lemma III.2 (Preservation) $\phi = ([\psi_1, \ldots, \psi_n], \Gamma)$. *If* $\phi, [a_1, \ldots, a_n] \vdash \mathbb{W}$ *and* $\mathbb{W} \longmapsto \mathbb{W}'$*, then there exist* $a'_1, \ldots, a'_n$ *such that* $\phi, [a'_1, \ldots, a'_n] \vdash \mathbb{W}'$.

Theorem III.3 (Soundness) $\phi = ([\psi_1, \ldots, \psi_n], \Gamma)$. *If there exist* $a_1, \ldots, a_n$*, such that* $\phi, [a_1, \ldots, a_n] \vdash \mathbb{W}$*, then for any* $n \geq 0$*, there exist a world* $\mathbb{W}'$ *and* $a'_1, \ldots, a'_n$ *such that* $\mathbb{W} \longmapsto^n \mathbb{W}'$ *and* $\phi, [a'_1, \ldots, a'_n] \vdash \mathbb{W}'$.

We have mechanized the complete soundness proof in the Coq proof assistant. Interested readers can check out our Coq implementation [10] for more detail.

IV. Example

In this section, we give an example to demonstrate the mechanized verification of safety properties(usually the shared memory invariant in parallel program) for concurrent assembly code with reentrant locks.

A simple example is present in Fig. 11, which is the concurrent code that computes the next even number according to the current value stored in the shared memory location x. The shared location x is protected by a reentrant lock l and the value stored in it is initialized with 0. In high level code, we unfold the inlined synchronized method located from line 3 to line 5. The inlined method leads to acquiring the same lock which has been held by the caller. The lock l is a reentrant lock, so this code will run correctly without deadlock.

The corresponding assembly code and assertions are given in Fig. 12. We verify the code in our framework. Following MIPS convention, we assume r_0 always contains 0. Assertions are shown as annotations enclosed in "$-\{\}$", the shared memory location x protected by the reentrant lock l specified by the invariants $\mathtt{m}$ that requires the value stored in shared location is even ($\exists a, b. x \mapsto a \wedge a = 2b$). According to CSL, the shared memory is well-formed and conforms to the invariant $\mathtt{m}$ when the corresponding lock is free. The precondition and postcondition for instructions in the example are straightforward, it is trivial to apply the inference rules in our framework to verify the assembly code with assertions. Note that the rule LOCK is applied to reason about the acquiring lock operation at the initial entry and the rule RELOCK is applied to reason about the second reacquiring lock operation. Only the first locking operation transfers the shared memory location x from the shared part to its private part and the second locking operation acquires nothing but increasing the reentrancy level by 1. We use the rules REUNLOCK and UNLOCK to reason about the first and second releasing operations respectively.

```
       Initially : [x] = 0;
Thread ID : k
       //x protected by lock l
      1: lock l;
      2: [x] := [x] + 1;
      3: lock l;
      4: [x] := [x] + 1;
      5: unlock l;
      6: unlock l;
```

Figure 11. Reentrant Lock Example

$$\mathtt{m} \overset{\text{def}}{=} \exists a, b. x \mapsto a \wedge a = 2b$$
$$\Gamma \overset{\text{def}}{=} \{l \rightsquigarrow \mathtt{m}\}$$

$-\{\lfloor \mathtt{emp} \rfloor\}$
`lock      l;`
$-\{\lfloor \mathtt{m} \rfloor \wedge \mathsf{own}_k(l,1)\}$
`ld        r1,x(r0);`
$-\{\exists a, b. \lfloor x \mapsto a \rfloor \wedge \mathsf{own}_k(l,1) \wedge \mathsf{r}_1 = a \wedge a = 2b\}$
`addi      r1,r1,1;`
$-\{\exists a, b. \lfloor x \mapsto a \rfloor \wedge \mathsf{own}_k(l,1) \wedge \mathsf{r}_1 = a+1 \wedge a = 2b\}$
`st        r1,x(r0);`
$-\{\exists a, b. \lfloor x \mapsto a+1 \rfloor \wedge \mathsf{own}_k(l,1) \wedge \mathsf{r}_1 = a+1 \wedge a = 2b\}$
`lock      l;`
$-\{\exists a, b. \lfloor x \mapsto a+1 \rfloor \wedge \mathsf{own}_k(l,2) \wedge \mathsf{r}_1 = a+1 \wedge a = 2b\}$
`addi      r1,r1,1;`
$-\{\exists a, b. \lfloor x \mapsto a+1 \rfloor \wedge \mathsf{own}_k(l,2) \wedge \mathsf{r}_1 = a+2 \wedge a = 2b\}$
`st        r1,x(r0);`
$-\{\exists a, b. \lfloor x \mapsto a+2 \rfloor \wedge \mathsf{own}_k(l,2) \wedge \mathsf{r}_1 = a+2 \wedge a = 2b\}$
$-\{\exists a', b'. \lfloor x \mapsto a' \rfloor \wedge \mathsf{own}_k(l,2) \wedge \mathsf{r}_1 = a' \wedge a' = 2b'\}$
$-\{\lfloor \mathtt{m} \rfloor \wedge \mathsf{own}_k(l,2)\}$
`unlock    l;`
$-\{\lfloor \mathtt{m} \rfloor \wedge \mathsf{own}_k(l,1)\}$
`unlock    l;`
$-\{\lfloor \mathtt{emp} \rfloor\}$

Figure 12. Assembly Code with Assertions

V. Related Work

Many approaches have been proposed for reasoning about properties of both sequential and concurrent programs [3], [11], [12], [13]. But most efforts on concurrent programs focus on the non-reentrant lock-based programs and do not consider the reentrant locks. As we present in this paper, there exist some differences between reasoning about concurrent programs with non-reentrant locks and those with reentrant locks.

Peter O'Hearn [1], [6] proposed CSL, which applies the local-reasoning idea from separation logic [14], [2] to verify shared-state concurrent programs with memory pointers. Separation logic assertions are used to capture ownerships of resources. Separating conjunction enforces the partition of resources. Verification of sequential threads in CSL is no different from verification of sequential programs. Memory modularity is supported by using separating conjunction and frame rules. However, the rule for acquiring and releasing resource in CSL cannot be directly applied to verify concurrent programs with reentrant mutual exclusive locks. We adapt CSL to a concurrent assembly language with reentrant locks.

In recent years, Shao *et al.* have developed CCAP[8], CMAP[15] and SAGL[13] as extensions to the PCC framework to verify properties of concurrent programs using locks, which are treated as non-reentrant locks. And we present an extension to enable verification of concurrent program using reentrant locks.

A recent work [16] proposes a verification technique for a concurrent Java-like language with reentrant locks. The verification technique is based on permission accounting separation logic. The essential differences between [16] and our paper are: we focus on verifying concurrent assembly code with reentrant locks and develop an extension to the PCC framework; instead of using hand-writing proof, we provide machine-checkable proof for our framework.

VI. Conclusion

In this paper we have presented a system for verifying concurrent programs using reentrant locks. We modeled an assembly level machine with built-in reentrant locking primitives. We adapted concurrent separation logic to verifying concurrent assembly code with reentrant locks. We also used a simple example to demonstrate the effectiveness of our framework.

Acknowledgement

We thank Prof. Zhong Shao (Yale University) and anonymous referees for their suggestions and comments on an earlier version of this paper. This research was supported in part by the National Natural Science Foundation of China under Grant No. 90718026 and gifts from Intel Corporation. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] P. W. O'Hearn, "Resources, concurrency and local reasoning," in *Proc. 15th Int'l Conf. on Concurrency Theory (CONCUR'04)*, ser. LNCS, vol. 3170, 2004, pp. 49–67.

[2] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proc. LICS'02*, Jul. 2002, pp. 55–74.

[3] C. A. R. Hoare, "Towards a theory of parallel programming," in *Operating Systems Techniques*, C. A. R. Hoare and R. H. Perrott, Eds. Academic Press, 1972, pp. 61–71.

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

[5] G. Necula, "Proof-carrying code," in *Proc. 24th ACM Symp. on Principles of Prog. Lang.* ACM Press, Jan. 1997, pp. 106–119.

[6] P. W. O'Hearn, "Resources, concurrency, and local reasoning," *Theoretical Computer Science*, vol. 375, no. 1-3, pp. 271–307, 2007.

[7] 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.

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

[9] A. K. Wright and M. Felleisen, "A syntactic approach to type soundness," *Information and Computation*, vol. 115, no. 1, pp. 38–94, 1994.

[10] M. Fu, Y. Zhang, , and Y. Li, "Formal reasoning about concurrent assembly code with reentrant locks. Coq implementation." `http://ssg.ustcsz.edu.cn/vsync/papers/frcacrl`, Jan 2009.

[11] S. Owicki and D. Gries, "Verifying properties of parallel programs: an axiomatic approach," *Commun. ACM*, vol. 19, no. 5, pp. 279–285, 1976.

[12] A. Cotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv, "Local reasoning for storable locks and threads," in *Proc. Fifth Asian Symp. on Prog. Lang. and Sys. (APLAS'07), LNCS 4807*. Springer-Verlag, November 2007.

[13] X. Feng, R. Ferreira, and Z. Shao, "On the relationship between concurrent separation logic and assume-guarantee reasoning," Dept. of Computer Science, Yale University, New Haven, CT, Tech. Rep. YALEU/DCS/TR-1374 and Formulation in Coq, January 2007.

[14] S. S. Ishtiaq and P. W. O'Hearn, "BI as an assertion language for mutable data structures," in *Proc. 28th ACM Symp. on Principles of Prog. Lang.*, 2001, pp. 14–26.

[15] X. Feng and Z. Shao, "Modular verification of concurrent assembly code with dynamic thread creation and termination," in *Proc. ICFP'05*, 2005, pp. 254–267.

[16] C. Haack, M. Huisman, and C. Hurlin, "Reasoning about java's reentrant locks," in *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 171–187.