Front. Comput. Sci. China, 2008, 2(3)
DOI 10.1007/s11704-009-0000-0

REVIEW ARTICLE

Formal Verification of Concurrent Programs with Read-Write Locks

Ming Fu, Yu Zhang, Yong Li

Department of Computer Science & Technology University of Science & Technology of China, Hefei, 230027,China
Suzhou Institute for Advanced Study University of Science & Technology of China, SuZhou,215123,China

Abstract Read-write locking is an important mechanism to improve concurrent granularity, but it is difficult to reason about the safety of concurrent programs with read-write locks. Concurrent Separation Logic(CSL) provides a simple but powerful technique for locally reasoning about concurrent programs with mutual exclusive locks. Unfortunately, CSL cannot be directly applied to reason about concurrent programs with read-write locks due to the different concurrent control mechanisms.

This paper focuses on extending CSL and present a proof-carrying code(PCC) system for reasoning about concurrent programs with read-write locks. We extend the heap model with a writing permission set, denoted as logical heap, then define "strong separation" and "weak separation" over logical heap. Following CSL's local-reasoning idea, we develop a novel program logic to enforces weak separations of heap between different threads and support verification of concurrent programs with read-write locks.

Keywords verification, concurrent separation logic, mutual exclusive locks, read-write locks

1 Introduction

Read-write locking is an important mechanism to improve concurrent granularity. It is widely employed in realistic applications. Ensuring the safety of concurrent programs with read-write locks is an essential but challenging task.

A mutual exclusion lock (also called a mutex) is used to enforce that only one thread can access a certain set of shared heap locations at a given time. Lock has two operations: **lock** and **unlock**. The lock operation denotes the beginning of a critical section while the unlock operation denotes the ending. The most basic lock can only be locked one time by a given thread (non-reentrant) and can be implemented with a boolean and an atomic test-set operation. O'Hearn [1] proposed concurrent separation logic(CSL) to reason about concurrent programs with mutual exclusion locks. CSL is a logic based on the notion that separate parts of a program depending on separated heap can be dealt with independently. Proofs in CSL consider the heap of each thread separately and adjust the ownership of heap protected by mutual exclusion lock among threads. The use of the separating conjunction * in pre- and post- conditions allows assertion to specify heap of program state and transfer of ownership of shared heap between threads. However, CSL supports only strong separation which exclusively partitions the shared heap among threads and does not provide a mechanism of sharing read-only ownership of shared heap among several threads. So it is not sufficient for verification of concurrent programs with read-write locks.

A read-write lock functions differently from a mutual exclusive lock: it either allows multiple threads to access the shared heap in a read-only way, or it allows one, and only one, thread to have full access (both read and write) to the shared heap. It is fundamentally different from the normal mutex (since it allows multiple threads to obtain a read lock). By using this kind of threads the program can be faster by increasing the concurrent granularity. In this paper, we attempt to extend CSL to develop a framework for the verification of concurrent programs with read-write locks. In our extension of CSL, we apply both thread modular reasoning and heap modular reasoning to read-write locks. According to the semantics of the read-write locks, the separation of share heap becomes more complicate in concurrent programs with read-write locks. Therefore we introduce access permission into the heap model, and then define "strong sepa-

Received month dd.yyyy; accepted month dd.yyyy

E-mail: *yuzhang@ustc.edu.cn*

ration" and "weak separation" under the modified heap model. The strong separation exclusively partitions heap among threads, while the weak separation allow heap partitions with overlaps, but all the overlaps only have read permissions. We add an additional weak separating conjunction operator $\circledast$ to allow for weak separation in our extension of CSL. We also study the relationship between the two kinds of separation, and conclude that strong separation in CSL is not always applicable for all cases.

Our study is based on an assembly language with RISC-style instructions and built-in rlock/unrlock/wlock/unwlock primitives. Instead of using the high-level parallel language proposed by Hoare [2], 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 heap. Therefore we can have a quick formulation in Coq [3] 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 [4]. The extension of CSL method studied in this paper is adapted to the low-level language. The relationship between the low-level extension over CSL and the original logic by O'Hearn [1] is discussed in section 6.

Our paper makes the following contributions:

1. We model an abstract machine with read-write lock primitives supported at assembly-level. We also give the operational semantics for the synchronized primitives which is much more complex than the mutual-exclusion lock primitives.
2. We extend CSL to certify concurrent programs based on the abstract machine. Our extension of CSL is significant since it is a novel program logic that can successfully support modular verification of concurrent programs synchronized with read-write locks.
3. We implement our framework using Coq proof assistant, and prove examples under the framework. The result shows that our extension of CSL can be easily applied for verification of concurrent programs synchronized with read-write locks.

The rest of this paper is organized as follows: In section 2, we briefly introduce CSL and informally explain our ownership transfer technique for reasoning about read-write locks. In section 3, we describe the abstract machine we model and the program logic based on extension of CSL we use to reason. Section 4 presents examples that are written and proved under our framework. We discuss the implementation in Section 5. Finally we discuss the related work and conclude.

2 Preliminaries

Before giving the formal description of our framework for verifying concurrent programs with read-write locks, we first explain the the limitation of the original CSL in verifying critical sections with read-write locks, then informally describe our ownership transfer technique for reasoning about programs with read-write locks.

2.1 Concurrent Separation Logic(CSL)

In this subsection, we give a brief description of CSL and demonstrate the necessity of extending CSL to support local reasoning about concurrent programs with read-write locks. CSL is an extension of *separation logic* [5] for reasoning about shared heap race-free concurrent programs. Separation logic is a programming logic which is tailored to reason about heap-manipulating programs. A simplified syntax for separation logic is shown in Fig. 1.

$$\begin{aligned} P, Q \ ::= \ & \mathtt{l} \mapsto v \mid \mathtt{emp} \mid P * Q \mid P \wedge Q \mid P \vee Q \\ & \mid \exists\, x.\ Q \mid \forall\, x.\ P \end{aligned}$$

Fig. 1 Syntax of Separation Logic

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

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

$$\frac{\{P_1\}C_1\{Q_1\} \quad \{P_2\}C_2\{Q_2\}}{\{P_1 * P_2\}C_1 \| C_2\{Q_1 * Q_2\}}$$

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 exclusive locking operations.

However, the separating and local reasoning mechanism for mutual exclusive locks is not suitable for read-write locks, since concurrent programs synchronized with read-write locks allow heaps to be safely shared among concurrent threads provided they all promise only to read, never to write. From the concurrency rule in original CSL we know that the separating conjunction "*" in separation logic does not allow the different concurrent threads to see the same heap and read from the same locations of it simultaneously, so "*" is too strong to enable the read sharing.

In order to allow several threads read from the same heap locations simultaneously, we introduce a weaker separating conjunction "$\circledast$" which is defined over an extended logical heap and allows read sharing. The corresponding concurrency rule may be written as follow:

$$\frac{\{P_1\}C_1\{Q_1\} \quad \{P_2\}C_2\{Q_2\}}{\{P_1 \circledast P_2\}C_1 \| C_2\{Q_1 \circledast Q_2\}}$$

In the above rule, our proposed separating conjunction "$\circledast$" allows the heap specified by P_1 and P_2 respectively to contain the same sub heap with read-only permission, so threads C_1 and C_2 can read the locations of the sub heap concurrently.

2.2 Ownership Transfer for Read-Write Locks

In Fig. 2, we describe the partition of the whole heap and use it to demonstrate the ownership transfer of acquiring and releasing read-write locks. The whole heap space is partitioned into several thread-private heaps and the shared heap. The shared heap is partitioned and each part is protected by a read-write lock. For each part of the partition, an invariant is assigned to specify its well-formedness.

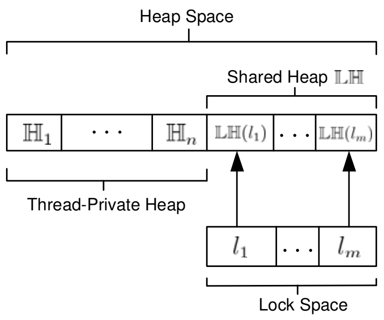

Fig. 2 Partition of Heap

In Fig 3, we give the ownership transfer of acquiring write lock, which is the same with that in the original CSL. ie, the ownership transfer must ensure the shared heap be acquired exclusively. The heap $\mathbb{H}_k$ enclosed in a dashed box is the private heap of a thread. When the lock l_i is acquired in write mode, the thread takes advantage of mutual exclusion provided by acquiring write lock l_i and treats the lock-protected heap $\mathbb{LH}(l_i)$ as private. Before releasing the lock l_i, it must ensure that the part of heap $\mathbb{LH}(l_i)$ is well-formed with regard to the corresponding invariant.

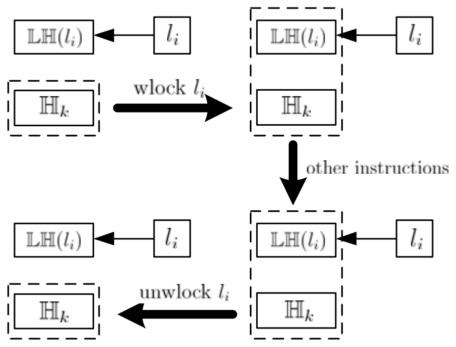

Fig. 3 Ownership Transfer for Acquiring and Releasing Write Lock

However, the ownership transfer for acquiring and releasing read locks is different. In Fig 4, when the lock l_i is acquired in read-only mode, the thread copies the part of shared heap $\mathbb{LH}(l_i)$ with read-only permission (The heap with a shadow box represents that the heap is read-only) and added it into the thread's private part. Before releasing the lock l_i, because the part of heap $\mathbb{LH}(l_i)$ has never been written, it is well-formed with regard to the corresponding invariant and can be removed from thread's private part.

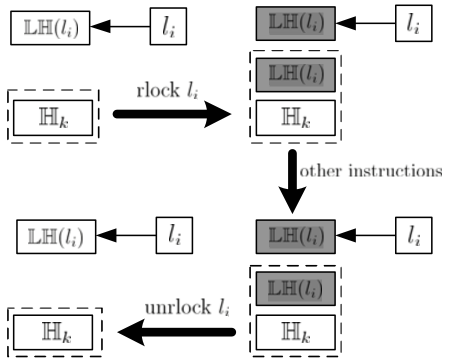

Fig. 4 Ownership Transfer for Acquiring and Releasing Read Lock

Since the original heap model cannot directly support shared heap copying with read-only permission, it is necessary to extend the heap with permission. We defines the logical heap, which contains writing permission sets in addition to the normal heaps. The heap locations without writing permissions are read-only. Acquiring a write lock gets the write permission of the heap protected by the lock, while acquiring a read lock gets read-only permission of the heap. The formal description of ownership transfer under acquiring read-write lock is in the

$$
\begin{array}{rcll}
(\mathit{PogramState}) & \mathbb{P} & ::= & (\mathbb{H}, [\mathbb{T}_1, \ldots, \mathbb{T}_n], \mathbb{L}) \\
(\mathit{Thread}) & \mathbb{T}_{\mathtt{t}} & ::= & (\mathbb{C}, \mathbb{R}, \mathbb{I}, \mathtt{t}) \\
(\mathit{CodeHeap}) & \mathbb{C} & \in & \mathit{Labels} \rightharpoonup \mathit{InstrSeq} \\
(\mathit{Heap}) & \mathbb{H} & \in & \mathit{Labels} \rightharpoonup \mathit{Word} \\
(\mathit{RegFile}) & \mathbb{R} & \in & \mathit{Register} \rightarrow \mathit{Word} \\
(\mathit{LockMap}) & \mathbb{L} & \in & \mathit{Locks} \rightarrow \mathit{WBit} \times \mathit{RBit} \\
(\mathit{WBit}) & \mathtt{u} & := & \varepsilon \,|\, \mathtt{t} \\
(\mathit{RBit}) & \mathbb{Q} & := & \{\mathtt{t}\}^* \\
(\mathit{Register}) & \mathtt{r} & ::= & \mathtt{r}_0 \,|\, \ldots \,|\, \mathtt{r}_{31} \\
(\mathit{Labels}) & \mathtt{f}, \mathtt{l} & ::= & i \;\; (\mathit{nat\ nums}) \\
(\mathit{Locks}) & l & ::= & i \;\; (\mathit{nat\ nums}) \\
(\mathit{ThrdID}) & \mathtt{t} & ::= & 1 \,|\, \ldots \,|\, n \\
(\mathit{Word}) & \mathtt{w} & ::= & i \;\; (\mathit{nat\ nums}) \\
(\mathit{InstrSeq}) & \mathbb{I} & ::= & \mathsf{j}\ \mathtt{f} \,|\, \mathsf{jr}\ \mathtt{r}_s \,|\, \iota; \mathbb{I} \\
(\mathit{Instr}) & \iota & ::= & \mathsf{add}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \mid \mathsf{addi}\ \mathtt{r}_d, \mathtt{r}_s, i \\
 & & & \mid \mathsf{st}\ \mathtt{r}_t, i(\mathtt{r}_s) \mid \mathsf{ld}\ \mathtt{r}_t, i(\mathtt{r}_s) \\
 & & & \mid \mathsf{wlock}\ l \mid \mathsf{rlock}\ l \mid \mathsf{sub}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \\
 & & & \mid \mathsf{unrlock}\ l \mid \mathsf{unwlock}\ l
\end{array}
$$

Fig. 5 The Abstract Machine

next section.

3 The Framework

In this section, we present our abstract machine model and its operational semantics. Then a program logic extended CSL with access permission is presented for the verification of assembly concurrent programs synchronized with read-write locks, its structure is similar to other CAP [6] systems.

3.3 The Abstract Machine

Fig. 5 defines the abstract machine and the syntax of an assembly language. We extend CAP by adding several built-in instructions for read-write locks. A program state $\mathbb{P}$ on the abstract machine consists of a shared heap $\mathbb{H}$, a lock mapping $\mathbb{L}$ and n threads $[\mathbb{T}_1, \ldots, \mathbb{T}_n]$.

The global shared heap $\mathbb{H}$ is modeled as a finite partial mapping from heap locations $\mathtt{l}$ (natural numbers) to word values (natural numbers). The locking map $\mathbb{L}$ is a finite mapping from read-write locks to its corresponding pair $(\mathtt{u}, \mathbb{Q})$. We implement a read-write lock with a pair $(\mathtt{u}, \mathbb{Q})$, where the integer $\mathtt{u}$ identifies the thread holding the lock in write mode(or ε if no such thread exists) and the integer set $\mathbb{Q}$ contains identifiers of all threads holding the lock in read mode. The default value of the pair is $(\varepsilon, \emptyset)$.

The abstract machine has a fixed number of threads. Each thread $\mathbb{T}_{\mathtt{t}}$ contains its own code heap $\mathbb{C}$, register file $\mathbb{R}$, instruction sequence $\mathbb{I}$ currently being executed, and its thread id $\mathtt{t}$. Here we allow each thread to have its own register file, which is consistent with most implementation of thread libraries where the register file is saved in the execution context when a thread is preempted. The register file $\mathbb{R}$ is represented as a total function from registers to words. The code heap $\mathbb{C}$ maps code labels to instruction sequences, which is a list of assembly instructions ending with a jump instruction. The set of instructions we present here are the commonly used subsets in RISC machines with additional wlock/unwlock/rlock/unrlock primitives for synchronization.

The step function ($\longmapsto$) of program state $\mathbb{P}$ is defined in Fig. 6. We use the auxiliary relation $(\mathbb{H}, \mathbb{T}, \mathbb{L}) \rightsquigarrow (\mathbb{H}', \mathbb{T}', \mathbb{L}')$ to define the effects of the execution of the thread $\mathbb{T}$. Here we follow the preemptive thread model where execution of threads can be preempted at any program point, but execution of individual instructions is *atomic*. The operational semantics for most instructions are quite straightforward. Note that the execution of instruction for acquiring locks. A thread attempts to acquire lock l in write mode by executing wlock l instruction, there must be no other threads holding l in read or write mode, and the corresponding value of lock l is $(\varepsilon, \emptyset)$. The result state of this operation is to replace $\mathtt{u}$ with the thread identifier which owns the lock. Similarly, acquiring the lock in read mode using the rlock l instruction must make sure no other threads hold l in write mode, and the result state of the operation is to put the thread identifier into the set $\mathbb{Q}$. In this model, we do not support re-entrant locks. If the lock l has been held in read mode or write mode, execution of the "wlock l" or "rlock l" instruction will be blocked even if the lock is held by the current thread. The release operations are straightforward. All of these locking operation should respect the invariant: $\forall l. \mathbb{L}(l) = (\mathtt{u}, \mathbb{Q}) \wedge (\mathtt{u} = \varepsilon \vee \mathbb{Q} = \emptyset)$. The function Next_ι defines the effects of the sequential instruction ι over heap and register files.

3.4 Extension of CSL

In this subsection, we introduce an extension of CSL that supports two different kinds of separations in the logical heap model. In order to specify access permission for each heap location, we associate heap block with a writing permission set and modify both the syntax and logical semantics of separation logic based on the logical heap model. With the help of extended logical heap model, we can describe heap with read-only or read-write permission transferred between different threads.

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

where

$(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}, k), \mathbb{L}) \rightsquigarrow (\mathbb{H}', \mathbb{T}', \mathbb{L}')$		
if $\mathbb{I} =$	then $(\mathbb{H}', \mathbb{T}', \mathbb{L}') =$	
j $\mathtt{f}$	$(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L})$	where $\mathbb{I}' = \mathbb{C}(\mathtt{f})$
jr $\mathtt{r}_s$	$(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L})$	where $\mathbb{I}' = \mathbb{C}(\mathbb{R}(\mathtt{r}_s))$
wlock $l; \mathbb{I}'$	$(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L}\{l \rightsquigarrow (k, \emptyset)\})$ $(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}, k), \mathbb{L})$	if $\mathbb{L}(l) = (\varepsilon, \emptyset)$ otherwise
unwlock $l; \mathbb{I}'$	$(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L}\{l \rightsquigarrow (\varepsilon, \emptyset)\})$	if $\mathbb{L}(l) = (k, \emptyset)$
rlock $l; \mathbb{I}'$	$(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L}\{l \rightsquigarrow (\varepsilon, \mathbb{Q} \cup \{k\})\})$ $(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}, k), \mathbb{L})$	if $\mathbb{L}(l) = (\varepsilon, \mathbb{Q}) \wedge k \notin \mathbb{Q}$ otherwise
unrlock $l; \mathbb{I}'$	$(\mathbb{H}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L}\{l \rightsquigarrow (\varepsilon, \mathbb{Q} \setminus \{k\})\})$	if $\mathbb{L}(l) = (\varepsilon, \mathbb{Q}) \wedge k \in \mathbb{Q}$
$\iota; \mathbb{I}'$ for other ι	$(\mathbb{H}', (\mathbb{C}, \mathbb{R}', \mathbb{I}', k), \mathbb{L})$	where $(\mathbb{H}', \mathbb{R}') = \mathsf{Next}_\iota\ (\mathbb{H}, \mathbb{R})$

and

if $\iota =$	then $\mathsf{Next}_\iota\ (\mathbb{H}, \mathbb{R}) =$	
add $\mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) + \mathbb{R}(\mathtt{r}_t)\})$	
addi $\mathtt{r}_d, \mathtt{r}_s, i$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) + i\})$	
ld $\mathtt{r}_t, i(\mathtt{r}_s)$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_t \rightsquigarrow \mathbb{H}(\mathbb{R}(\mathtt{r}_s) + i)\})$	when $\mathbb{R}(\mathtt{r}_s) + i \in \mathsf{dom}(\mathbb{H})$
sub $\mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) - \mathbb{R}(\mathtt{r}_t)\})$	
st $\mathtt{r}_t, i(\mathtt{r}_s)$	$(\mathbb{H}\{\mathbb{R}(\mathtt{r}_s) + i \rightsquigarrow \mathbb{R}(\mathtt{r}_t)\}, \mathbb{R})$	when $\mathbb{R}(\mathtt{r}_s) + i \in \mathsf{dom}(\mathbb{H})$

Fig. 6 Operational Semantics of the Machine

3.4.1 Logical Heap Model

The logical heap model $\mathbb{M}$ extends the heap structure with a special writing permission set which is used to denote the access permission for each heap location. According to the logical heap model we formalize extended thread state and extended program state below:

$$\begin{array}{rrcl}
(\mathit{LogicalHeap}) & \mathbb{M} & ::= & (\mathbb{H}, \mathbb{D}) \\
(\mathit{PermissionSet}) & \mathbb{D} & ::= & \{\mathtt{l}\}^* \\
(\mathit{XState}) & \mathbb{X} & ::= & (\mathbb{M}, \mathbb{R}, \mathtt{t}, \mathbb{L}) \\
(\mathit{ExtProgState}) & \mathbb{W} & ::= & (\mathbb{M}, [\mathbb{T}_1, \ldots, \mathbb{T}_2], \mathbb{L})
\end{array}$$

The logical heap model $\mathbb{M}$ contains the data heap block $\mathbb{H}$ and the corresponding writing permission set $\mathbb{D}$ which is always a subset of the domain of $\mathbb{H}$, denoted as $\mathsf{dom}(\mathbb{H})$. The writing permission set $\mathbb{D}$ is used to represent the access permission of each heap location. A heap location in writing permission set $\mathbb{D}$ can be read and written. A heap location in $\mathsf{dom}(\mathbb{H})$ but not in $\mathbb{D}$ is read-only. Instead of using real heap structure, we use logical heap model to formalize the extended thread state $\mathbb{X}$ which contains the local information of the thread, including the private logical heap $\mathbb{M}$ owned by the thread, the thread's register file, identifier and the lock set. The extended program state $\mathbb{W}$ use logical heap $\mathbb{M}$ to trace the access permission of each heap location.

3.4.2 Strong Separation VS. Weak Separation

By using the logical heap model, we can describe two different partitions on shared heap. One is similar with the partition in standard separation logic, which strictly partitions shared heap into disjoint part, and we denote it as *strong separation*. The other partitions shared heap in a relaxed way allowing overlap among different threads with read-only permission, we denote this as *weak separation*. The formal definitions are displayed below:

$$\mathbb{M}_1 \perp \mathbb{M}_2 \overset{\text{def}}{=} \mathsf{dom}(\mathbb{M}_1.\mathbb{H}_1) \cap \mathsf{dom}(\mathbb{M}_2.\mathbb{H}_2) = \emptyset$$

$$\begin{array}{rl}\mathbb{M}_1 \veebar \mathbb{M}_2 \overset{\text{def}}{=} & (\forall \mathtt{l}.\mathtt{l} \in \mathsf{dom}(\mathbb{M}_1.\mathbb{H}_1) \cap \mathsf{dom}(\mathbb{M}_2.\mathbb{H}_2) \rightarrow \\ & \mathbb{M}_1.\mathbb{H}_1(\mathtt{l}) = \mathbb{M}_2.\mathbb{H}_2(\mathtt{l}) \wedge \mathtt{l} \notin \mathbb{M}_1.\mathbb{D}_1 \cup \mathbb{M}_2.\mathbb{D}_2) \\ & \wedge(\mathbb{M}_1.\mathbb{D}_1 \cap \mathbb{M}_2.\mathbb{D}_2 = \emptyset)\end{array}$$

$$\mathbb{M}_1 \uplus \mathbb{M}_2 \overset{\text{def}}{=} \begin{cases} (\mathbb{M}_1.\mathbb{H}_1 \cup \mathbb{M}_2.\mathbb{H}_2, \mathbb{M}_1.\mathbb{D}_1 \cup \mathbb{M}_2.\mathbb{D}_2) & \\ & \text{if } \mathbb{M}_1 \perp \mathbb{M}_2 \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

$$\mathbb{M}_1 \oplus \mathbb{M}_2 \overset{\text{def}}{=} \begin{cases} (\mathbb{M}_1.\mathbb{H}_1 \cup \mathbb{M}_2.\mathbb{H}_2, \mathbb{M}_1.\mathbb{D}_1 \cup \mathbb{M}_2.\mathbb{D}_2) & \\ & \text{if } \mathbb{M}_1 \veebar \mathbb{M}_2 \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

We use $\mathbb{M}_1 \perp \mathbb{M}_2$ to represent a strong separation relation between $\mathbb{M}_1$ and $\mathbb{M}_2$, which means there does not exist a heap location which is both in the domain of $\mathbb{M}_1.\mathbb{H}_1$ and in the domain of $\mathbb{M}_2.\mathbb{H}_2$. $\mathbb{M}_1 \veebar \mathbb{M}_2$ denotes a weak separation relation between $\mathbb{M}_1$ and $\mathbb{M}_2$, it allows the two heap blocks $\mathbb{M}_1.\mathbb{H}_1$ and $\mathbb{M}_2.\mathbb{H}_2$ contain the same heap locations neither in $\mathbb{M}_1.\mathbb{D}_1$ nor in $\mathbb{M}_2.\mathbb{D}_2$. $\mathbb{M}_1 \uplus \mathbb{M}_2$ and $\mathbb{M}_1 \oplus \mathbb{M}_2$ respectively define heap merge operations under strong separation and weak separation. According to the definition of the logical heap model, weak separation ensures the overlapping heap locations never to be written. The following lemmas present the relationship between strong separation and weak separation.

Lemma 3.1 (Strong Separation Weakening) *If* $\mathbb{M}_1 = (\mathbb{H}_1, \mathbb{D}_1), \mathbb{D}_1 \subseteq \mathsf{dom}(\mathbb{H}_1)$, $\mathbb{M}_2 = (\mathbb{H}_2, \mathbb{D}_2)$, $\mathbb{D}_2 \subseteq \mathsf{dom}(\mathbb{H}_2)$*, and* $\mathbb{M}_1 \perp \mathbb{M}_2$*, then* $\mathbb{M}_1 \veebar \mathbb{M}_2$.

Proof. According to the definition of strong separation, we can conclude $\mathsf{dom}(\mathbb{H}_1) \cap \mathsf{dom}(\mathbb{H}_2) = \emptyset$ from $\mathbb{M}_1 \perp \mathbb{M}_2$. We prove the lemma by destructing the definition of weak separation and giving the proof of the following two proposition:

- $(\forall \mathtt{l}.\mathtt{l} \in \mathsf{dom}(\mathbb{H}_1) \cap \mathsf{dom}(\mathbb{H}_2) \rightarrow \mathbb{H}_1(\mathtt{l}) = \mathbb{H}_2(\mathtt{l}) \wedge \mathtt{l} \notin \mathbb{D}_1 \cup \mathbb{D}_2)$, because there exist not any heap location in the empty set, so the premise is false and we can prove this proposition trivially by inversion the false premise.
- $(\mathbb{D}_1 \cap \mathbb{D}_2 = \emptyset)$, we can easily prove this by applying the conditions $\mathbb{D}_1 \subseteq \mathsf{dom}(\mathbb{H}_1)$, $\mathbb{D}_2 \subseteq \mathsf{dom}(\mathbb{H}_2)$ and $\mathsf{dom}(\mathbb{H}_1) \cap \mathsf{dom}(\mathbb{H}_2) = \emptyset$.

Qed. □

Lemma 3.2 (Weak Separation Strengthening) *If* $\mathbb{M}_1 = (\mathbb{H}_1, \mathbb{D}_1), \mathbb{D}_1 = \mathsf{dom}(\mathbb{H}_1)$, $\mathbb{M}_2 = (\mathbb{H}_2, \mathbb{D}_2), \mathbb{D}_2 \subseteq \mathsf{dom}(\mathbb{H}_2)$*, and* $\mathbb{M}_1 \veebar \mathbb{M}_2$ *,then* $\mathbb{M}_1 \perp \mathbb{M}_2$.

Proof. We prove this lemma by considering the two following different cases :

- if $\mathsf{dom}(\mathbb{H}_1) \cap \mathsf{dom}(\mathbb{H}_2) = \emptyset$, then according to the definition of strong separation, we can conclude $\mathbb{M}_1 \perp \mathbb{M}_2$.
- if $\mathsf{dom}(\mathbb{H}_1) \cap \mathsf{dom}(\mathbb{H}_2) \neq \emptyset$, then there exists a heap location $\mathtt{l} \in \mathsf{dom}(\mathbb{H}_1) \cap \mathsf{dom}(\mathbb{H}_2)$. By applying $(\forall \mathtt{l}.\mathtt{l} \in \mathsf{dom}(\mathbb{H}_1) \cap \mathsf{dom}(\mathbb{H}_2) \rightarrow \mathbb{H}_1(\mathtt{l}) = \mathbb{H}_2(\mathtt{l}) \wedge \mathtt{l} \notin \mathbb{D}_1 \cup \mathbb{D}_2)$ inferred from the definition of $\mathbb{M}_1 \veebar \mathbb{M}_2$, we obtain $\mathtt{l} \notin \mathbb{D}_1 \cup \mathbb{D}_2$, we can replace $\mathbb{D}_1$ with $\mathsf{dom}(\mathbb{H}_1)$ because of $\mathbb{D}_1 = \mathsf{dom}(\mathbb{H}_1)$, then $\mathtt{l} \notin \mathsf{dom}(\mathbb{H}_1)$ which is contradict with the existing premise $\mathtt{l} \in \mathsf{dom}(\mathbb{H}_1) \cap \mathsf{dom}(\mathbb{H}_2)$. Since this case will never happen.

Qed. □

3.4.3 Operational Semantics with Logical Heap

We give the concurrent operational semantics based on the logical heap model and extended state in Fig. 7. The simulation between the logical operational semantics for program executions with the logical heap and in the real machine semantics shown in Fig. 6 is obvious and we do not show it in the paper.

In the operational semantics with the logical heap, we use a merged logical heap to depict the state transition. The transfer of the ownership described in section 2 is not expressed in the logical operational semantics, and the logical partition over the merged logical heap will be enforced by the top rule of the program logic. The function $\overset{lg}{\longmapsto}$ is defined for stepping over extended program state, and we use the auxiliary relation $(\mathbb{M}, \mathbb{T}, \mathbb{L}) \overset{lg}{\rightsquigarrow} (\mathbb{M}', \mathbb{T}', \mathbb{L}')$ to define the effects of the execution of the thread $\mathbb{T}$ over the logical heap. The state transitions over the logical heap for most of the instructions can be easily obtained from the operation semantics shown in Fig. 6. The function $\mathsf{Next'}_\iota$ is used to describe the effects made by some local actions over logical heap and register files. The safe execution of instruction $\mathtt{st}\ \mathtt{r}_t, i(\mathtt{r}_s)$ requires that the written heap location should be contained in the writing permission set of the current logical heap.

3.4.4 Assertion Language

Fig. 8 shows the syntax of assertion language for reasoning about concurrent programs with read-write locks under our proposed framework. We treat $\mathtt{m}$ as a predicate over real heap, and $\mathtt{v}$ is predicate over logical heap. $\mathtt{a}$ is a predicate over extended thread state to ensure the safe execution of the instruction sequence. In addition to the usual formulate of predicate calculus, we introduce five new forms of predicates that describe the heap with access permission. Since the abstract machine is built with both locking primitives and register at low level, we define some predicates to specify locking map and register files.

We give the definitions of logical semantics for each assertion construct based on the abstract machine model in Fig. 8. The semantics for the predicates $\mathtt{m}$ over real heap is similar to the semantics of separation logic, we have given the formal explanation for them in subsection 2.2. $\lfloor \mathtt{m} \rfloor_r$ and $\lfloor \mathtt{m} \rfloor_w$ are predicates over logical heap, the former asserts read-only heap block satisfying $\mathtt{m}$, the latter asserts read-write heap block satisfying $\mathtt{m}$. $\mathtt{v}_1 * \mathtt{v}_2$ is a predicate over logical heap that can be split into two parts with strong separation relation, the first satisfying $\mathtt{v}_1$ and the second $\mathtt{v}_2$. A new separating conjunction "$\circledast$" is introduced to specify the weak separation relationship between logical heaps. $\mathtt{v}_1 \circledast \mathtt{v}_2$ is interpreted as a predicate over logical heap which can be split into two parts with weak separation relation, the first satisfying $\mathtt{v}_1$ and the second $\mathtt{v}_2$. $[\mathtt{v}]$ is a predicate over extended thread state that contains logical heap satisfying $\mathtt{v}$. $r = v$ is a predicate specifying the register files status in the extended thread state. $\mathsf{own}_r(l, \mathtt{t})$ and $\mathsf{own}_w(l, \mathtt{t})$ are predicates over extended thread state holding the

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

$(\mathbb{M}, (\mathbb{C}, \mathbb{R}, \mathbb{I}, k), \mathbb{L}) \overset{lg}{\rightsquigarrow} (\mathbb{M}', \mathbb{T}', \mathbb{L}')$		
if $\mathbb{I} =$	then $(\mathbb{M}', \mathbb{T}', \mathbb{L}') =$	
j f	$(\mathbb{M}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L})$	where $\mathbb{I}' = \mathbb{C}(\mathtt{f})$
jr $\mathtt{r}_s$	$(\mathbb{M}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L})$	where $\mathbb{I}' = \mathbb{C}(\mathbb{R}(\mathtt{r}_s))$
wlock $l; \mathbb{I}'$	$(\mathbb{M}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L}\{l \rightsquigarrow (k, \emptyset)\})$ $(\mathbb{M}, (\mathbb{C}, \mathbb{R}, \mathbb{I}, k), \mathbb{L})$	if $\mathbb{L}(l) = (\varepsilon, \emptyset)$ otherwise
unwlock $l; \mathbb{I}'$	$(\mathbb{M}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L}\{l \rightsquigarrow (\varepsilon, \emptyset)\})$	if $\mathbb{L}(l) = (k, \emptyset)$
rlock $l; \mathbb{I}'$	$(\mathbb{M}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L}\{l \rightsquigarrow (\varepsilon, \mathbb{Q} \cup \{k\})\})$ $(\mathbb{M}, (\mathbb{C}, \mathbb{R}, \mathbb{I}, k), \mathbb{L})$	if $\mathbb{L}(l) = (\varepsilon, \mathbb{Q}) \wedge k \notin \mathbb{Q}$ otherwise
unrlock $l; \mathbb{I}'$	$(\mathbb{M}, (\mathbb{C}, \mathbb{R}, \mathbb{I}', k), \mathbb{L}\{l \rightsquigarrow (\varepsilon, \mathbb{Q} \setminus \{k\})\})$	if $\mathbb{L}(l) = (\varepsilon, \mathbb{Q}) \wedge k \in \mathbb{Q}$
$\iota; \mathbb{I}'$ for other ι	$(\mathbb{M}', (\mathbb{C}, \mathbb{R}', \mathbb{I}', k), \mathbb{L})$	where $(\mathbb{M}', \mathbb{R}') = \mathsf{Next'}_\iota\ (\mathbb{M}, \mathbb{R})$

and

if $\iota =$	then $\mathsf{Next'}_\iota\ ((\mathbb{H}, \mathbb{D}), \mathbb{R}) =$	
add $\mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$	$((\mathbb{H}, \mathbb{D}), \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) + \mathbb{R}(\mathtt{r}_t)\})$	
addi $\mathtt{r}_d, \mathtt{r}_s, i$	$((\mathbb{H}, \mathbb{D}), \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) + i\})$	
ld $\mathtt{r}_t, i(\mathtt{r}_s)$	$((\mathbb{H}, \mathbb{D}), \mathbb{R}\{\mathtt{r}_t \rightsquigarrow \mathbb{H}(\mathbb{R}(\mathtt{r}_s) + i)\})$	when $\mathbb{R}(\mathtt{r}_s) + i \in \mathsf{dom}(\mathbb{H})$
sub $\mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$	$((\mathbb{H}, \mathbb{D}), \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) - \mathbb{R}(\mathtt{r}_t)\})$	
st $\mathtt{r}_t, i(\mathtt{r}_s)$	$((\mathbb{H}\{\mathbb{R}(\mathtt{r}_s) + i \rightsquigarrow \mathbb{R}(\mathtt{r}_t)\}, \mathbb{D}), \mathbb{R})$	when $\mathbb{R}(\mathtt{r}_s) + i \in \mathbb{D}$

Fig. 7 Operational Semantics of the Machine with Logical Heap

lock l in read mode and write mode respectively. Here, we omit explanation for some straightforward assertion constructs, such as $\mathtt{m}_1 \wedge \mathtt{m}_2$, *etc.* Some straightforward axioms for weak separation conjunction are shown in Fig. 9. Weak separation conjunction $\circledast$ has some same properties as strong separation conjunction $*$ in original separation logic. Most of them are easy to be proven through the semantics of the assertion language. We omit the proof of them here due to the space limitation.

3.5 Program Specification

We use the mechanized meta-logic implemented in the Coq proof assistant [3] as our specification language. The logic corresponds to a higher-order predicate logic with inductive definitions.

Fig. 10 shows the specification constructs for our calculus. The program specification ϕ consists of 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 an predicate $\mathtt{a}$ over extended thread state $\mathbb{X}$ as the precondition of corresponding instruction sequence. The specification Γ of lock-protected heap maps a lock to invariant $\mathtt{m}$ specifying shared heap. We also give five different judgements to represent well-formed program, well-formed thread, well-formed code heap, well-formed instruction sequences and well-formed instructions respectively, which are used to construct the inference rules. The semantics for each proposition will be explained in subsection 3.6.

3.6 Inference Rules

The inference rules for a program and instructions are presented in Fig. 13. The PROG rule requires that there be a partition of the global logical heap into $n+1$ parts satisfying weak separation. The data heap of shared logical heap $\mathbb{M}_s$ must satisfy the invariants specified in Γ.

We give the definition of predicate $\mathtt{a}_\Gamma$ below, which is the separating conjunction of invariants assigned to the locks which are read-free (locks held in read mode by some threads) or write-free (locks not held in any mode by any threads). It ensures that shared heap are well-formed outside critical regions or inside read-only critical regions which start with rlock instruction and end with unrlock instruction. Here $\forall_*$ is an indexed, finitely iterated separating conjunction, which is formalized in Fig. 11.

$$\begin{aligned}\mathtt{a}_\Gamma \overset{\text{def}}{=} \ & \lambda(\mathbb{M}, (\mathbb{R}, k), \mathbb{L}).\ \exists \mathbb{M}_1, \mathbb{M}_2. \mathbb{M}_1 \uplus \mathbb{M}_2 = \mathbb{M} \wedge \\ & \lfloor \forall_* l \in \{l \mid \mathbb{L}(l) = (\varepsilon, \mathbb{Q}) \wedge \mathbb{Q} \neq \emptyset\}.\ \Gamma(l) \rfloor_r\ \mathbb{M}_1 \ \wedge \\ & \lfloor \forall_* l \in \{l \mid \mathbb{L}(l) = (\varepsilon, \emptyset)\}.\ \Gamma(l) \rfloor_w\ \mathbb{M}_2\end{aligned}$$

As in O'Hearn's original work on CSL [7], we also re-

$$\begin{array}{lrcl}
(\mathit{HeapPred}) & \mathtt{m} & \in & \mathit{Heap} \rightarrow \mathit{Prop} \\
(\mathit{LogicalHeapPred}) & \mathtt{v} & \in & \mathit{LogicalHeap} \rightarrow \mathit{Prop} \\
(\mathit{StPred}) & \mathtt{a} & \in & \mathit{XState} \rightarrow \mathit{Prop}
\end{array}$$

$$\begin{array}{lcl}
\mathtt{m} & ::= & \mathtt{l} \mapsto v \mid \mathsf{emp} \mid \mathtt{m}_1 * \mathtt{m}_2 \\
 & \mid & \mathtt{m}_1 \wedge \mathtt{m}_2 \mid \mathtt{m}_1 \vee \mathtt{m}_2 \mid \exists\, x.\ \mathtt{m} \mid \forall\, x.\ \mathtt{m} \\
\mathtt{v} & ::= & \lfloor \mathtt{m} \rfloor_r \mid \lfloor \mathtt{m} \rfloor_w \mid \mathtt{v}_1 * \mathtt{v}_2 \mid \mathtt{v}_1 \circledast \mathtt{v}_2 \\
 & \mid & \mathtt{v}_1 \wedge \mathtt{v}_2 \mid \mathtt{v}_1 \vee \mathtt{v}_2 \\
\mathtt{a} & ::= & [\mathtt{v}] \mid r = v \mid \mathsf{own}_r(l, \mathtt{t}) \mid \mathsf{own}_w(l, \mathtt{t}) \\
 & \mid & \mathtt{a}_1 \wedge \mathtt{a}_2 \mid \mathtt{a}_1 \vee \mathtt{a}_2 \mid \exists\, x.\ \mathtt{a} \mid \forall\, x.\ \mathtt{a}
\end{array}$$

$$\begin{array}{lcl}
\mathsf{emp} & \overset{\text{def}}{=} & \lambda\mathbb{H}.\mathbb{H} = \emptyset \\
\mathtt{l} \mapsto v & \overset{\text{def}}{=} & \lambda\mathbb{H}.\mathsf{dom}(\mathbb{H}) = \{\mathtt{l}\} \wedge \mathbb{H}(\mathtt{l}) = v \\
\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 \\
\mathtt{m}_1 \wedge \mathtt{m}_2 & \overset{\text{def}}{=} & \lambda\mathbb{H}.\mathtt{m}_1\ \mathbb{H} \wedge \mathtt{m}_2\ \mathbb{H} \\
\mathtt{m}_1 \vee \mathtt{m}_2 & \overset{\text{def}}{=} & \lambda\mathbb{H}.\mathtt{m}_1\ \mathbb{H} \vee \mathtt{m}_2\ \mathbb{H} \\
\exists\, x.\ \mathtt{m} & \overset{\text{def}}{=} & \lambda\mathbb{H}.\exists x.\mathtt{m}\ \mathbb{H} \\
\forall\, x.\ \mathtt{m} & \overset{\text{def}}{=} & \lambda\mathbb{H}.\forall x.\mathtt{m}\ \mathbb{H} \\
\lfloor \mathtt{m} \rfloor_r & \overset{\text{def}}{=} & \lambda\mathbb{M}.\mathbb{M} = (\mathbb{H}, \mathbb{D}) \wedge \mathtt{m}\ \mathbb{H} \wedge \mathbb{D} = \emptyset \\
\lfloor \mathtt{m} \rfloor_w & \overset{\text{def}}{=} & \lambda\mathbb{M}.\mathbb{M} = (\mathbb{H}, \mathbb{D}) \wedge \mathtt{m}\ \mathbb{H} \wedge \mathbb{D} = \mathsf{dom}(\mathbb{H}) \\
\mathtt{v}_1 * \mathtt{v}_2 & \overset{\text{def}}{=} & \lambda\mathbb{M}.\exists\mathbb{H}_1, \mathbb{H}_2, \mathbb{D}_1, \mathbb{D}_2.(\mathbb{H}_1, \mathbb{D}_1) \uplus (\mathbb{H}_2, \mathbb{D}_2) \\
 & & = \mathbb{M} \wedge \mathtt{v}_1\ (\mathbb{H}_1, \mathbb{D}_1) \wedge \mathtt{v}_2\ (\mathbb{H}_2, \mathbb{D}_2) \\
\mathtt{v}_1 \circledast \mathtt{v}_2 & \overset{\text{def}}{=} & \lambda\mathbb{M}.\exists\mathbb{H}_1, \mathbb{H}_2, \mathbb{D}_1, \mathbb{D}_2.(\mathbb{H}_1, \mathbb{D}_1) \oplus (\mathbb{H}_2, \mathbb{D}_2) \\
 & & = \mathbb{M} \wedge \mathtt{v}_1\ (\mathbb{H}_1, \mathbb{D}_1) \wedge \mathtt{v}_2\ (\mathbb{H}_2, \mathbb{D}_2) \\
\mathtt{v}_1 \wedge \mathtt{v}_2 & \overset{\text{def}}{=} & \lambda\mathbb{M}.\mathtt{v}_1\ \mathbb{M} \wedge \mathtt{v}_2\ \mathbb{M} \\
\mathtt{v}_1 \vee \mathtt{v}_2 & \overset{\text{def}}{=} & \lambda\mathbb{M}.\mathtt{v}_1\ \mathbb{M} \vee \mathtt{v}_2\ \mathbb{M} \\
\exists\, x.\ \mathtt{v} & \overset{\text{def}}{=} & \lambda\mathbb{M}.\exists x.\mathtt{v}\ \mathbb{M} \\
\forall\, x.\ \mathtt{v} & \overset{\text{def}}{=} & \lambda\mathbb{M}.\forall x.\mathtt{v}\ \mathbb{M} \\
[\mathtt{v}] & \overset{\text{def}}{=} & \lambda\mathbb{X}.\ \mathtt{v}\ \mathbb{X}.\mathbb{M} \\
r = v & \overset{\text{def}}{=} & \lambda\mathbb{X}.\ \mathbb{X}.\mathbb{R}(r) = v \\
\mathsf{own}_r(l, \mathtt{t}) & \overset{\text{def}}{=} & \lambda\mathbb{X}.\ \mathbb{X}.\mathbb{L}(l) = (\varepsilon, \mathbb{Q}) \wedge \mathtt{t} \in \mathbb{Q} \\
\mathsf{own}_w(l, \mathtt{t}) & \overset{\text{def}}{=} & \lambda\mathbb{X}.\ \mathbb{X}.\mathbb{L}(l) = (\mathtt{t}, \emptyset) \\
\mathtt{a}_1 \wedge \mathtt{a}_2 & \overset{\text{def}}{=} & \lambda\mathbb{X}.\mathtt{a}_1\ \mathbb{X} \wedge \mathtt{a}_2\ \mathbb{X} \\
\mathtt{a}_1 \vee \mathtt{a}_2 & \overset{\text{def}}{=} & \lambda\mathbb{X}.\mathtt{a}_1\ \mathbb{X} \vee \mathtt{a}_2\ \mathbb{X} \\
\exists\, x.\ \mathtt{a} & \overset{\text{def}}{=} & \lambda\mathbb{X}.\exists x.\mathtt{a}\ \mathbb{X} \\
\forall\, x.\ \mathtt{a} & \overset{\text{def}}{=} & \lambda\mathbb{X}.\forall x.\mathtt{a}\ \mathbb{X}
\end{array}$$

Fig. 8 The Assertion Language

$$\begin{array}{l}
\lfloor \mathtt{m}_1 * \mathtt{m}_2 \rfloor_r \Leftrightarrow \lfloor \mathtt{m}_1 \rfloor_r * \lfloor \mathtt{m}_2 \rfloor_r \\
\lfloor \mathtt{m}_1 * \mathtt{m}_2 \rfloor_w \Leftrightarrow \lfloor \mathtt{m}_1 \rfloor_w * \lfloor \mathtt{m}_2 \rfloor_w \\
\lfloor \mathsf{emp} \rfloor_r \Leftrightarrow \lfloor \mathsf{emp} \rfloor_w \\
\lfloor \mathsf{emp} \rfloor_{\_} \circledast \mathtt{v} \Leftrightarrow \mathtt{v} \\
\mathtt{v}_1 \circledast \mathtt{v}_2 \Leftrightarrow \mathtt{v}_2 \circledast \mathtt{v}_1 \\
(\mathtt{v}_1 \circledast \mathtt{v}_2) \circledast \mathtt{v}_3 \Leftrightarrow \mathtt{v}_1 \circledast (\mathtt{v}_2 \circledast \mathtt{v}_3) \\
(\mathtt{v}_1 \circledast \mathtt{v}_2) * \mathtt{v}_3 \Leftrightarrow \mathtt{v}_1 \circledast (\mathtt{v}_2 * \mathtt{v}_3) \\
(\mathtt{v}_1 \vee \mathtt{v}_2) \circledast \mathtt{v}_3 \Leftrightarrow (\mathtt{v}_1 \circledast \mathtt{v}_3) \vee (\mathtt{v}_2 \circledast \mathtt{v}_3) \\
(\mathtt{v}_1 \wedge \mathtt{v}_2) \circledast \mathtt{v}_3 \Leftrightarrow (\mathtt{v}_1 \circledast \mathtt{v}_3) \wedge (\mathtt{v}_2 \circledast \mathtt{v}_3)
\end{array}$$

Fig. 9 Axioms for Weak Separation Conjunction

$$\begin{array}{rrcl}
(\mathit{ProgSpec}) & \phi & ::= & ([\psi_1, \ldots, \psi_n], \Gamma) \\
(\mathit{CdHpSpec}) & \psi & ::= & \{\mathtt{f} \rightsquigarrow \mathtt{a}\}^* \\
(\mathit{ResourceINV}) & \Gamma & \in & \mathit{Locks} \rightharpoonup \mathit{HeapPred}
\end{array}$$

$$\begin{array}{rl}
\phi, [\mathtt{a}_1, \ldots, \mathtt{a}_n] \vdash \mathbb{W} & \text{(Well-formed program)} \\
\psi, \Gamma \vdash \{\mathtt{a}\}\,(\mathbb{M}, \mathbb{T}, \mathbb{L}) & \text{(Well-formed thread)} \\
\psi, \Gamma \vdash \mathbb{C} : \psi' & \text{(Well-formed code heap)} \\
\psi, \Gamma \vdash \{\mathtt{a}\}\,\mathbb{I} & \text{(Well-formed instruction sequences)} \\
\psi, \Gamma \vdash \{\mathtt{a}\}\,\iota\,\{\mathtt{a}'\} & \text{(Well-formed instructions)}
\end{array}$$

Fig. 10 Specification Constructs

$$\forall_* x \in S.\ P(x) \overset{\text{def}}{=} \begin{cases} \mathsf{emp} & \text{if } S = \emptyset \\ P(x_i) * \forall_* x \in S'.\ P(x) & \text{if } S = S' \uplus \{x_i\} \end{cases}$$

$$\begin{array}{lcl}
\mathsf{Precise}(\mathtt{m}) & \overset{\text{def}}{=} & \forall\mathbb{H}_1, \mathbb{H}_2, \mathbb{H}.\mathbb{H}_1 \subseteq \mathbb{H} \rightarrow \mathbb{H}_2 \subseteq \mathbb{H} \rightarrow \\
 & & \quad \mathtt{m}\ \mathbb{H}_1 \wedge \mathtt{m}\ \mathbb{H}_2 \rightarrow \mathbb{H}_1 = \mathbb{H}_2
\end{array}$$

$$\mathsf{Precise}(\Gamma) \overset{\text{def}}{=} \forall l \in \mathsf{dom}(\Gamma).\ \mathsf{Precise}(\Gamma(l))$$

$$\mathtt{a} \Rightarrow \mathtt{a}' \overset{\text{def}}{=} \forall\mathbb{X}.\ \mathtt{a}\ \mathbb{X} \rightarrow \mathtt{a}'\ \mathbb{X}$$

$$\begin{array}{l}
\mathtt{a} \triangleright \mathsf{Next}'_\iota \overset{\text{def}}{=} \forall(\mathbb{M}, (\mathbb{R}, k), \mathbb{L}).\ \exists\, \mathbb{M}', \mathbb{R}'.(\mathbb{M}', \mathbb{R}') = \\
\qquad \mathsf{Next}'_\iota\ (\mathbb{M}, \mathbb{R}) \wedge \mathtt{a}\ (\mathbb{M}', (\mathbb{R}', k), \mathbb{L})
\end{array}$$

Fig. 11 Definitions of Auxiliary Propositions

quire invariants specified in Γ to be precise, denoted as $\mathsf{Precise}(\Gamma)$. Each $\mathbb{M}_k$ is privately owned by thread $\mathbb{T}_k$ with access permission and every thread of the program is well-formed. Thus the verification of a multi-threaded program can be decompose into the verification of its component threads accessing only private heap. The reasoning is both thread-modular and data-modular in our framework.

The well-formedness of $\mathbb{T}_k$ is checked by applying the THRD rule. A thread is well-formed when the code heap is required to be well-formed and the precondition for the thread is satisfied. Since the precondition $\mathtt{a}$ only specifies the private resource, we use "filter" operator "$\mathbb{L}\rfloor_k$" to prevent $\mathtt{a}$ from having access to the ownership information of locks not owned by the current thread:

$$(\mathbb{L}\rfloor_k)(l) \overset{\text{def}}{=} \begin{cases} (k, \emptyset) & \text{if } \mathbb{L}(l) = (k, \emptyset) \\ (\varepsilon, \{k\}) & \text{if } \mathbb{L}(l) = (\varepsilon, \mathbb{Q}) \wedge k \in \mathbb{Q} \\ (\varepsilon, \emptyset) & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\mathsf{a} * \mathsf{v} &\overset{\text{def}}{=} \lambda(\mathbb{M}, (\mathbb{R}, \mathtt{t}), \mathbb{L}).\exists \mathbb{M}_1, \mathbb{M}_2.\mathbb{M}_1 \uplus \mathbb{M}_2 = \mathbb{M} \land \\
&\qquad \mathsf{a}\ (\mathbb{M}_1, (\mathbb{R}, \mathtt{t}), \mathbb{L}) \land \mathsf{v}\ \mathbb{M}_2 \\
\mathsf{rlk}\ l\ \mathsf{a} &\overset{\text{def}}{=} \lambda(\mathbb{M}, (\mathbb{R}, k), \mathbb{L}).\ \exists \mathbb{Q}.\mathbb{L}(l) = (\varepsilon, \mathbb{Q}) \land \\
&\qquad \mathsf{a}\ (\mathbb{M}, (\mathbb{R}, k), \mathbb{L}\{l \leadsto (\varepsilon, \mathbb{Q} \cup \{k\})\}) \\
\mathsf{unrlk}\ l\ \mathsf{a} &\overset{\text{def}}{=} \lambda(\mathbb{M}, (\mathbb{R}, k), \mathbb{L}).\ \exists \mathbb{Q}.\mathbb{L}(l) = (\varepsilon, \mathbb{Q}) \land k \in \mathbb{Q} \land \\
&\qquad \mathsf{a}\ (\mathbb{M}, (\mathbb{R}, k), \mathbb{L}\{l \leadsto (\varepsilon, \mathbb{Q} \setminus \{k\})\}) \\
\mathsf{wlk}\ l\ \mathsf{a} &\overset{\text{def}}{=} \lambda(\mathbb{M}, (\mathbb{R}, k), \mathbb{L}).\ \mathsf{a}\ (\mathbb{M}, (\mathbb{R}, k), \mathbb{L}\{l \leadsto (k, \emptyset)\}) \\
\mathsf{unwlk}\ l\ \mathsf{a} &\overset{\text{def}}{=} \lambda(\mathbb{M}, (\mathbb{R}, k), \mathbb{L}).\ \mathbb{L}(l) = (k, \emptyset) \land \\
&\qquad \mathsf{a}\ (\mathbb{M}, (\mathbb{R}, k), \mathbb{L}\{l \leadsto (\varepsilon, \emptyset)\})
\end{aligned}$$

Fig. 12 Definitions of Auxiliary Macros

And 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 SEQ, J and JR rules ensure that it is safe to execute the instruction sequence if the precondition is satisfied. If the instruction sequence starts with a normal sequential instruction ι, we need to come up with an assertion a' which serves both as the post-condition of ι and as the precondition of the remaining instruction sequence. If reaching the last jump instruction of the instruction sequence, both the J and JR rules require that the assertion assigned to the target address in ψ be satisfied after the jump.

Most inference rules for instructions are similar and grouped in the OTHER rule. It requires that the precondition a ensures the safe execution of the instruction; and the resulting state satisfies the post-condition a'. We use "$\mathsf{a} \Rightarrow \mathsf{a}'$" for logical implication lifted for state predicates and "$\mathsf{a}' \triangleright \mathsf{Next}_\iota$" to represent the weakest precondition of a'. They are formalized in Fig 11.

In the WLOCK rule, we use "$\mathsf{wlk}\ l\ \mathsf{a}'$" which is formalized in Fig. 12 to represent the weakest precondition of a'. If the execution of $\mathsf{wlock}\ l$ instruction successfully acquired the lock l in write mode, through our global invariant we know that the part of heap protected by l satisfies the invariant $\lfloor \Gamma(l) \rfloor_w$ allowing the write permission to be transferred. Therefore, we can carry both the knowledge $\Gamma(l)$ and the read-write access permission in the post-condition a'. Also carrying $\lfloor \Gamma(l) \rfloor_w$ in a' allows subsequent instructions to read or write the part of heap.

In the UNWLOCK rule, The weakest precondition of a' is "$\mathsf{unwlk}\ l\ \mathsf{a}'$" (see Fig. 12). When the lock l is released by executing $\mathsf{unwlock}\ l$, the heap protected by l must be well formed with respect to the invariant "$\lfloor \Gamma(l) \rfloor_w$". The strong separating conjunction "$*$" ensures that a' does not specify this part of heap. Therefore the subsequent instructions cannot access the part of heap unless the lock is acquired again. It is still correct to replace the strong separating conjunction "$*$" with the weak separating conjunction "$\circledast$" in the WLOCK and UNWLOCK rule, because we can infer the strong separation from the weak separation according to lemma 3.2.

The RLOCK rule is similar to the WLOCK rule, we can

$\boxed{\phi, [\mathsf{a}_1, \ldots, \mathsf{a}_n] \vdash \mathbb{W}}$ (***Well-formed program***)

$$\frac{\begin{array}{c}\phi = ([\psi_1, \ldots, \psi_n], \Gamma) \\ \mathbb{M}_s \oplus \mathbb{M}_1 \oplus \cdots \oplus \mathbb{M}_n = \mathbb{M} \\ \mathsf{a}_\Gamma\ (\mathbb{M}_s, \_, \mathbb{L}) \quad \mathsf{Precise}(\Gamma) \\ \psi_k, \Gamma \vdash \{\mathsf{a}_k\}\ (\mathbb{M}_k, \mathbb{T}_k, \mathbb{L}) \ \text{ for all } k\end{array}}{\phi, [\mathsf{a}_1, \ldots, \mathsf{a}_n] \vdash (\mathbb{M}, [\mathbb{T}_1, \ldots, \mathbb{T}_n], \mathbb{L})} \ (\text{PROG})$$

$\boxed{\psi, \Gamma \vdash \{\mathsf{a}\}\ (\mathbb{M}, \mathbb{T}, \mathbb{L})}$ (***Well-formed thread***)

$$\frac{\begin{array}{c}\mathsf{a}\ (\mathbb{M}, (\mathbb{R}, k), \mathbb{L}|_k) \\ \psi, \Gamma \vdash \mathbb{C} : \psi \qquad \psi, \Gamma \vdash \{\mathsf{a}\}\ \mathbb{I}\end{array}}{\psi, \Gamma \vdash \{\mathsf{a}\}\ (\mathbb{M}, (\mathbb{C}, \mathbb{R}, \mathbb{I}, k), \mathbb{L})} \ (\text{THRD})$$

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

$$\frac{\forall \mathtt{f} \in \mathsf{dom}(\psi') : \quad \psi, \Gamma \vdash \{\psi'(\mathtt{f})\}\ \mathbb{C}(\mathtt{f})}{\psi, \Gamma \vdash \mathbb{C} : \psi'} \ (\text{CDHP})$$

$\boxed{\psi, \Gamma \vdash \{\mathsf{a}\}\ \mathbb{I}}$ (***Well-formed instr. sequences***)

$$\frac{\psi, \Gamma \vdash \{\mathsf{a}\}\ \iota\ \{\mathsf{a}'\} \qquad \psi, \Gamma \vdash \{\mathsf{a}'\}\ \mathbb{I}}{\psi, \Gamma \vdash \{\mathsf{a}\}\ \iota; \mathbb{I}} \ (\text{SEQ})$$

$$\frac{\mathsf{a} \Rightarrow \psi(\mathtt{f})}{\psi, \Gamma \vdash \{\mathsf{a}\}\ \mathsf{j}\ \mathtt{f}} \ (\text{J}) \qquad \frac{\begin{array}{c}\forall \mathbb{X}.\mathbb{X} = (\mathbb{M}, (\mathbb{R}, k), \mathbb{L}) \to \\ \mathsf{a}\ \ \mathbb{X} \ \to \ \psi(\mathbb{R}(\mathtt{r}_s))\ \ \mathbb{X}\end{array}}{\psi, \Gamma \vdash \{\mathsf{a}\}\ \mathsf{jr}\ \mathtt{r}_s} \ (\text{JR})$$

$\boxed{\psi, \Gamma \vdash \{\mathsf{a}\}\ \iota\ \{\mathsf{a}'\}}$ (***Well-formed instructions***)

$$\frac{\mathsf{a} * \lfloor \mathtt{m} \rfloor_r \Rightarrow \mathsf{rlk}\ l\ \mathsf{a}'}{\psi, \Gamma\{l \leadsto \mathtt{m}\} \vdash \{\mathsf{a}\}\ \mathsf{rlock}\ l\ \{\mathsf{a}'\}} \ (\text{RLOCK})$$

$$\frac{\mathsf{a} \Rightarrow (\mathsf{unrlk}\ l\ \mathsf{a}') * \lfloor \mathtt{m} \rfloor_r}{\psi, \Gamma\{l \leadsto \mathtt{m}\} \vdash \{\mathsf{a}\}\ \mathsf{unrlock}\ l\ \{\mathsf{a}'\}} \ (\text{UNRLOCK})$$

$$\frac{\mathsf{a} * \lfloor \mathtt{m} \rfloor_w \Rightarrow \mathsf{wlk}\ l\ \mathsf{a}'}{\psi, \Gamma\{l \leadsto \mathtt{m}\} \vdash \{\mathsf{a}\}\ \mathsf{wlock}\ l\ \{\mathsf{a}'\}} \ (\text{WLOCK})$$

$$\frac{\mathsf{a} \Rightarrow (\mathsf{unwlk}\ l\ \mathsf{a}') * \lfloor \mathtt{m} \rfloor_w}{\psi, \Gamma\{l \leadsto \mathtt{m}\} \vdash \{\mathsf{a}\}\ \mathsf{unwlock}\ l\ \{\mathsf{a}'\}} \ (\text{UNWLOCK})$$

$$\frac{\begin{array}{c}\iota \in \{\mathsf{add}, \mathsf{addi}, \mathsf{ld}, \mathsf{sub}, \mathsf{st}\} \\ \mathsf{a} \Rightarrow \mathsf{a}' \triangleright \mathsf{Next'}_\iota\end{array}}{\psi, \Gamma \vdash \{\mathsf{a}\}\ \iota\ \{\mathsf{a}'\}} \ (\text{OTHERS})$$

Fig. 13 Inference Rules

just carry the knowledge $\Gamma(l)$ with read-only access permission in the post-condition a' which allows the subsequent instructions only to read the part of heap.

Similarly, the UNRLOCK rule also ensures the lock-protected heap must be well formed when the lock is released via $\mathsf{unrlock}\ l$ instruction.

3.7 Soundness

The soundness of these inference rules with respect to the operational semantics of the abstract machine is proved following the syntactic approach [8]. From the "progress" and "preservation" lemmas, we can guarantee that given a well-formed program under compatible preconditions, the current instruction sequence will be able to execute without getting "stuck". The soundness of our framework is formally stated as Theorem 3.5.

Lemma 3.3 (Progress) $\phi = ([\psi_1, \ldots, \psi_n], \Gamma)$. *If there exist* $\mathsf{a}_1, \ldots, \mathsf{a}_n$*, such that* $\phi, [\mathsf{a}_1, \ldots, \mathsf{a}_n] \vdash \mathbb{W}$*, then there exists an extended program state* $\mathbb{W}'$ *such that* $\mathbb{W} \overset{lg}{\longmapsto} \mathbb{W}'$.

Lemma 3.4 (Preservation) $\phi = ([\psi_1, \ldots, \psi_n], \Gamma)$. *If* $\phi, [\mathsf{a}_1, \ldots, \mathsf{a}_n] \vdash \mathbb{W}$ *and* $\mathbb{W} \overset{lg}{\longmapsto} \mathbb{W}'$*, then there exist* $\mathsf{a}'_1, \ldots, \mathsf{a}'_n$ *such that* $\phi, [\mathsf{a}'_1, \ldots, \mathsf{a}'_n] \vdash \mathbb{W}'$.

Theorem 3.5 (Soundness) $\phi = ([\psi_1, \ldots, \psi_n], \Gamma)$. *If there exist* $\mathsf{a}_1, \ldots, \mathsf{a}_n$*, such that* $\phi, [\mathsf{a}_1, \ldots, \mathsf{a}_n] \vdash \mathbb{W}$*, then for any* $n \geq 0$*, there exist an extended program state* $\mathbb{W}'$ *and* $\mathsf{a}'_1, \ldots, \mathsf{a}'_n$ *such that* $\mathbb{W} \overset{lg}{\longmapsto}{}^{n} \mathbb{W}'$ *and* $\phi, [\mathsf{a}'_1, \ldots, \mathsf{a}'_n] \vdash \mathbb{W}'$.

Proof. By induction over n. The base case is trivial. In the inductive case, if $n = k + 1$, from the induction hypothesis we can know that there exist an intermediate extended program state $\mathbb{W}''$ and $\mathsf{a}''_1 \ldots \mathsf{a}''_n$, such that $\mathbb{W} \overset{lg}{\longmapsto}{}^{k} \mathbb{W}''$ and $\phi, [\mathsf{a}''_1, \ldots, \mathsf{a}''_n] \vdash \mathbb{W}''$. Using lemma 3.3 and 3.4 and the induction hypothesis we can conclude the proof.
Qed. □

4 An Example

In this section, we use an simple example to demonstrate the effectiveness of our reasoning system, and illustrate an example in high-level program and their assembly counterpart. Fig. 14 shows a simple program, in which the shared heap location m and n are protected by the read-write lock l_1 and the location x is protected by the read-write lock l_2. The values stored in the location m and n are initialized with value 0. $\texttt{Thread}_1$ tries to acquire l_1 for reading and writing shared heap location m and n and acquires l_2 for only reading the shared heap location x. At the time successfully acquiring the locks, l_1 and l_2 are respectively held in write and read mode. After the execution of $\texttt{Thread}_1$, both heap locations m and n are added with the same value stored in x. $\texttt{Thread}_2$ is trivial, l_2 is acquired for both reading and writing the shared heap location x. The corresponding assembly code is given in Fig. 15.

```
      Initially : [m] = [n] = 0 ;
Thread_1:                       Thread_2:
   wlock  l_1;                     wlock l_2;
   rlock  l_2;         ||          [x] := [x] - 1;
   [m] := [m] + [x];               unwlock l_2;
   [n] := [n] + [x];
   unrlock  l_2;
   unwlock  l_1;
```

Fig. 14 Read-Write Lock Example

We verify the code under our framework. Following the MIPS convention, we assume the register $\mathtt{r}_0$ always contains 0. Assertions are shown as annotations enclosed in "-{}", the shared heap protected by l_1 and l_2 are specified by the invariants inv_1 and inv_2 respectively. We explain how to verify $\texttt{Thread}_1$ with the inference rules in our program logic as follows:

- At the initial program point of $\texttt{Thread}_1$, the thread does not own any resource and precondition is $[\lfloor \mathsf{emp} \rfloor_*]$. If the first $\mathsf{wlock}\ l_1$ instruction acquired the read-write lock l_1 successfully, the shared heap block protected by lock l_1 is transferred from shared heap to the thread's private part with the lifted invariant $\lfloor inv_1 \rfloor_w$ which allows the following code to read and write the heap specified by inv_1 and the lock l_1 is held by $\texttt{Thread}_1$ with the unique identifier 1 in write mode. The post-condition $[\lfloor inv_1 \rfloor_w] \wedge \mathsf{own}_w(l_1, 1)$ is implied by the above requirements, so the first instruction is a well-formed instruction by applying the rule WLOCK.
- The reasoning of the second instruction $\mathsf{rlock}\ l_2$ is similar to the first instruction, it applies the rule RLOCK to prove the well-form instruction $\mathsf{rlock}\ l_2$. The shared heap specified by inv_2 is lifted with read-only permission and added a copy of that into the thread's private part. The top rule tell us that the private heaps owned by each thread satisfy weak separation, and the heap copy does not prevent other threads to acquire the shared heap specified by inv_2 with read-only permission.
- From the third instruction to the ninth instruction, they only make effects on the private heap or register files. It is easy to reason about these instructions by applying the inference rule OTHERS, and the domain of the private heap is unchanged during the execution of this subsequence.
- We can infer that the current state satisfies the in-

variant inv_2 from the pre-condition of instruction $\mathsf{unrlock}\ l_2$, this means that the invariants inv_2 is reestablished. Hence we can apply the rule UNRLOCK to reason about this instruction. The thread removes the private heap satisfying inv_2 from its private domain.

- The reasoning of the last instruction is similar to that of the instruction $\mathsf{unrlock}\ l_2$, we can apply the rule UNWLOCK to reason about it, it is obvious that the precondition implies the invariant inv_1, so we can safely remove the heap specified by inv_1 from thread's private domain and return it to shared part.

After reasoning about the code sequence with the specification annotations, we can conclude that the above assembly code satisfies the given specification, and the code fragment is well-formed under our framework. Our reasoning supports modularity, because we never need to consider the behavior of $\mathtt{Thread}_2$ while reasoning about $\mathtt{Thread}_1$. $\mathtt{Thread}_2$ is simpler than $\mathtt{Thread}_1$, we can use the same method to reason about it, and the detail is omitted here. It is also sound to replace the "*" with "$\circledast$" in the pre- and post- conditions in this example, and we can use lemma 3.2 to ensure the soundness of the conversion.

$inv_1 \overset{\text{def}}{=} \exists v.\mathtt{m} \mapsto v * \mathtt{n} \mapsto v$

$inv_2 \overset{\text{def}}{=} \exists v.\mathtt{x} \mapsto v$

$\Gamma \overset{\text{def}}{=} \{l_1 \rightsquigarrow inv_1, l_2 \rightsquigarrow inv_2\}$

`Thread_1:` -{$[\lfloor \mathsf{emp} \rfloor_*]$}
`(1)          wlock   l_1`
-{$[\lfloor inv_1 \rfloor_w] \wedge \mathsf{own}_w(l_1, 1)$}
`(2)          rlock   l_2`
-{$[\lfloor inv_1 \rfloor_w * \lfloor inv_2 \rfloor_r] \wedge \mathsf{own}_w(l_1, 1) \wedge \mathsf{own}_r(l_2, 1)$}
`(3)          ld      r1, x(r0)`
-{$\exists v1.[\lfloor inv_1 \rfloor_w * \lfloor \mathtt{x} \mapsto v1 \rfloor_r] \wedge r1 = v1 \wedge \mathsf{own}_w(l_1, 1) \wedge \mathsf{own}_r(l_2, 1)$}
`(4)          ld      r2, m(r0)`
-{$\exists v, v1.(r2 = v \wedge [\lfloor \mathtt{m} \mapsto v \rfloor_w * \lfloor \mathtt{n} \mapsto v \rfloor_w * \lfloor \mathtt{x} \mapsto v1 \rfloor_r] \wedge r1 = v1) \wedge \mathsf{own}_w(l_1, 1) \wedge \mathsf{own}_r(l_2, 1)$}
`(5)          add     r2, r2, r1`
-{$\exists v, v1.(r2 = v + v1 \wedge r1 = v1 \wedge [\lfloor \mathtt{m} \mapsto v \rfloor_w * \lfloor \mathtt{n} \mapsto v \rfloor_w * \lfloor \mathtt{x} \mapsto v1 \rfloor_r]) \wedge \mathsf{own}_w(l_1, 1) \wedge \mathsf{own}_r(l_2, 1)$}
`(6)          st      r2, m(r0)`
-{$\exists v, v1.(r2 = v + v1 \wedge r1 = v1 \wedge [\lfloor \mathtt{m} \mapsto v + v1 \rfloor_w * \lfloor \mathtt{n} \mapsto v \rfloor_w * \lfloor \mathtt{x} \mapsto v1 \rfloor_r]) \wedge \mathsf{own}_w(l_1, 1) \wedge \mathsf{own}_r(l_2, 1)$}
`(7)          ld      r2, n(r0)`
-{$\exists v, v1.(r2 = v \wedge r1 = v1 \wedge [\lfloor \mathtt{m} \mapsto v + v1 \rfloor_w * \lfloor \mathtt{n} \mapsto v \rfloor_w * \lfloor \mathtt{x} \mapsto v1 \rfloor_r]) \wedge \mathsf{own}_w(l_1, 1) \wedge \mathsf{own}_r(l_2, 1)$}
`(8)          add     r2, r2, r1`
-{$\exists v, v1.(r2 = v + v1 \wedge r1 = v1 \wedge [\lfloor \mathtt{m} \mapsto v + v1 \rfloor_w * \lfloor \mathtt{n} \mapsto v \rfloor_w * \lfloor \mathtt{x} \mapsto v1 \rfloor_r]) \wedge \mathsf{own}_w(l_1, 1) \wedge \mathsf{own}_r(l_2, 1)$}
`(9)          st      r2, n(r0)`
-{$\exists v, v1.(r2 = v + v1 \wedge r1 = v1 \wedge [\lfloor \mathtt{m} \mapsto v + v1 \rfloor_w * \lfloor \mathtt{n} \mapsto v + v1 \rfloor_w * \lfloor \mathtt{x} \mapsto v1 \rfloor_r]) \wedge \mathsf{own}_w(l_1, 1) \wedge \mathsf{own}_r(l_2, 1)$} -{$[\lfloor inv_1 \rfloor_w * \lfloor inv_2 \rfloor_r] \wedge \mathsf{own}_w(l_1, 1) \wedge \mathsf{own}_r(l_2, 1)$}
`(10)         unrlock  l_2`
-{$[\lfloor inv_1 \rfloor_w] \wedge \mathsf{own}_w(l_1, 1)$}
`(11)         unwlock  l_1`
-{$[\lfloor \mathsf{emp} \rfloor_*]$}

`Thread_2:` -{$[\lfloor \mathsf{emp} \rfloor_*]$}
`(1)          wlock l_2`
-{$\lfloor inv_2 \rfloor_w \wedge \mathsf{own}_w(l_2, 2)$}
`(2)          ld r1, x(r0)`
-{$\exists v.r1 = v \wedge [\lfloor \mathtt{x} \mapsto v \rfloor_w] \wedge \mathsf{own}_w(l_2, 2)$}
`(3)          subi r1, r1, 1`
-{$\exists v.r1 = v \wedge [\lfloor \mathtt{x} \mapsto v \rfloor_w] \wedge \mathsf{own}_w(l_2, 2)$}
`(4)          st r1, x(r0)`
-{$\exists v.r1 = v \wedge [\lfloor \mathtt{x} \mapsto v \rfloor_w] \wedge \mathsf{own}_w(l_2, 2)$} -{$\lfloor inv_2 \rfloor_w \wedge \mathsf{own}_w(l_2, 2)$}
`(5)          unwlock l_2`
-{$[\lfloor \mathsf{emp} \rfloor_*]$}

Fig. 15 Read-Write Lock in Our Framework

5 Implementation

We have mechanized our verification in the Coq proof assistant [9], an interactive theorem prover which uses CiC as basic logic. Using Coq we can construct the specification and proofs as types and terms in CiC respectively. Proof checking in Coq functions as type checking of terms in CiC, which is easier to implement and more trustworthy. We build the abstract machine model and sound program logic using this tool.

In Fig. 16 we give a breakdown of the size of our proofs for our framework. For each component we give the number of non-empty lines of Coq proof scripts. It took us several man-months (by programmer who are familiar with the Coq system) to complete. Interested readers can obtain the Coq implementation from [10].

6 Related Work

Owicki & Gries [11] introduced the concept of non-interference between the proofs of concurrent threads. The method is not compositional. To address this problem, C. Jones [12] introduced the compositional rely-guarantee method [6,13,14] to describe the state changes performanced by the environment and by the program respectively. The rely-guarantee method supports thread modular verification in the sense that each thread is verified with regard to its own specification. It is general and does not require language constructs for synchronizations. However, each individual step of the verification, we need to prove that state transition satisfies the guarantee, it makes proofs more complicated with rely-

Lines	Component
897	Basic properties and tactics for Map
248	Abstract machine encoding and lemmas
127	Inference rules encoding and lemmas
46	Assertions construct definitions
913	Soundness proof for the framework
1458	Auxiliary definitions and lemmas

Fig. 16 Proof Script Size

guarantee method than our proposed method based on CSL. Also, the relies and guarantees are usually complicated and hard to define, because memory modularity is not supported in rely-guarantee method. Our extension of CSL for verifying the properties of the concurrent programs with read-write locks supports not only thread modularity but also memory modularity.

Peter O'Hearn [1, 7] proposed CSL for a high level parallel language. The language construct for synchronization in this high level language is in the form of "**with** r **when** b **do** c", which is used to mark the conditional critical region. The semantics of the conditional critical region is that only if the resource r has not been acquired by others and the boolean expression b is true then the statement c can be executed; otherwise the thread will get blocked. The similar high level language constructs can be designed for read-only/read-write critical region, which can be implemented using our rlock/unrlock/wlock/unwlock primitives. Each lock in our language corresponds to a resource name at the high level. Atomic instructions in our assembly language are very similar to actions in Brookes Semantics [15] , where semantic functions are defined for statements and expressions. These semantic functions can be viewed as a translation from the high-level language to a low-level language similar to ours. Thus the method and formulation proposed in this paper can be applied on high level parallel languages with refined synchronization construct for read-only and read-write critical regions. CSL applies the local-reasoning idea from separation logic [5, 16] to verify shared-state concurrent programs with memory pointers. Separation logic assertions are used to capture ownerships of resources. Separating conjunction enforces the partition of resources. Verification of sequential threads in CSL is no different from verification of sequential programs. Memory modularity is supported by using separating conjunction and frame rules. However, following Owicki and Gries [11], CSL works only for *well-synchronized programs* with mutual exclusive locks in the sense that transfer of resource ownerships including total access permissions can only occur at entry and exit points of critical regions. Our work goes further, we extend CSL for *well-synchronized programs* with read-write locks which is widely applied in fine-grained concurrent programs. We believe that our extension of CSL is fit for verifying some other fine-grained concurrent programs, such as concurrent programs using transactional memory *etc*, this will leave for our future work.

Feng *et.al* [14] proposed a combination of rely-guarantee and CSL, SAGL. which improves the modularity of rely-guarantee reasoning method and make the definition of relies and guarantees easier. Vafeiadis [17] also proposed another approach to combining rely/guarantee and CSL, which we refer to here as RGSep. Both RGSep and SAGL partition memory into shared and private parts. Our work only consider private memory with access permissions which are acquired via the read-write lock primitives, since our program logic is simpler than SAGL and RGSep.

Our work is similar with Bornat *et al.* [18]'s work which proposed a refinement of CSL with fine-grained resource accounting. Our work is another novel and lightweight logical approach to extend CSL for verifying concurrent programs with read-write locks, the essential difference between [18] and our paper are: we focus on verifying concurrent assembly code with read-write locks and develop an extension to PCC framework; instead of using hand-writing proof, we provide machine-checkable proof for our framework.

7 Conclusion

In this paper we have presented a framework for verifying concurrent programs synchronized with read-write locks. We modeled an assembly-level machine with built-in read-write lock primitives. We extended concurrent separation logic and applied it to our framework. We also used an example to demonstrate the effectiveness of our extended CSL.

Acknowledgements 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. 60673126 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. Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007.
2. C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, editors, *Operating Systems Techniques*, pages 61–71. Academic Press, 1972.

3. The Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.0, October 2004.
4. George Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Prog. Lang.*, pages 106–119. ACM Press, January 1997.
5. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS'02*, pages 55–74, July 2002.
6. Dachuan Yu and Zhong 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.
7. Peter W. O'Hearn. Resources, concurrency and local reasoning. In *Proc. 15th Int'l Conf. on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 49–67, 2004.
8. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
9. The Coq Development Team. The Coq proof assistant reference manual. The Coq release v7.1, October 2001.
10. Ming Fu, Yu Zhang, and Yong Li. Formal verification of concurrent programs with read-write locks(coq). `http://ssg.ustcsz.edu.cn/vsync/papers/ccprwl/`, July 2008.
11. Susan Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.
12. Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. on Programming Languages and Systems*, 5(4):596–619, 1983.
13. Xinyu Feng and Zhong Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proc. ICFP'05*, pages 254–267, 2005.
14. Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP '07*, pages 173–188, 2007.
15. Stephen Brookes. A semantics for concurrent separation logic. In *Proc. 15th International Conference on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 16–34, 2004.
16. Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM Symp. on Principles of Prog. Lang.*, pages 14–26, 2001.
17. Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.
18. Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proc. 32nd ACM Symp. on Principles of Prog. Lang.*, pages 259–270, 2005.