On the Relationship between Concurrent Separation Logic and Assume-Guarantee Reasoning*

Xinyu Feng Rodrigo Ferreira Zhong Shao

Department of Computer Science, Yale University
New Haven, CT 06520-8285, U.S.A.
{feng, rodrigo, shao}@cs.yale.edu

Abstract. We study the relationship between Concurrent Separation Logic (CSL) and the assume-guarantee (A-G) method (a.k.a. rely-guarantee method). We show in three steps that CSL can be treated as a specialization of the A-G method for well-synchronized concurrent programs. First, we present an A-G based program logic for a low-level language with built-in locking primitives. Then we extend the program logic with explicit separation of "private data" and "shared data", which provides better memory modularity. Finally, we show that CSL (adapted for the low-level language) can be viewed as a specialization of the extended A-G logic by enforcing the invariant that "*shared resources are well-formed outside of critical regions*". This work can also be viewed as a different approach (from Brookes') to proving the soundness of CSL: our CSL inference rules are proved as lemmas in the A-G based logic, whose soundness is established following the syntactic approach to proving soundness of type systems.

1 Introduction

It is hard to prove non-interference and correctness of shared-state concurrent programs because of the exponential state space. Memory aliasing makes concurrency verification even harder. Therefore a program logic supporting both thread modularity and memory modularity is the key to practical concurrency verification.

Peter O'Hearn [11, 10] proposed concurrent separation logic (CSL), which applies the local-reasoning idea from separation logic [7, 14] 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 [12], CSL works only for *well-synchronized programs* in the sense that transfer of resource ownerships can only occur at entry and exit points of critical regions. It is unclear how to apply CSL to support general concurrent programs with ad-hoc synchronizations.

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

Another approach to modular verification of shared-state concurrent programs is the assume-guarantee method (a.k.a. rely-guarantee method) [8]. In this approach, invariants of state transitions are specified using assumptions and guarantees. Each thread ensures that its atomic transitions satisfy its guarantee to the environment (*i.e.*, the collection of all other threads) as long as its assumption is satisfied by the environment. Non-interference is guaranteed as long as threads have compatible specifications, *i.e.*, the guarantee of each thread satisfies the assumptions of all other threads. The A-G method supports thread modular verification in the sense that each thread is verified with regard to its own specifications, and without looking into code of other threads. It is very general and does not require language constructs for synchronizations. However, in each individual step of the verification, we need to prove that the state transition satisfies the guarantee. This makes proofs more complicated in A-G reasoning than in CSL. Also, assumptions and guarantees are usually complicated and hard to define, because they specify global invariants for all shared resources during the program execution.

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

We then show that CSL can be viewed as a specialization of SAGL with the invariant that *shared resources are well-formed outside of critical regions*. The specialization is pinned down by formalizing the CSL invariant as a specific assumption and guarantee in SAGL. Our formulation can also be viewed as a novel approach to proving the soundness of CSL. Different from Brookes' proof based on an action-trace semantics [2], we prove that CSL inference rules are lemmas in SAGL with the specific assumption and guarantee. The soundness of SAGL is then proved following the syntactic approach to type soundness [18]. The proofs are formalized in the Coq proof assistant [16].

Our study is based on an assembly language with RISC-style instructions and built-in lock/unlock and memory allocation/free primitives. Instead of using the high-level parallel language proposed by Hoare [6], 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 [4] in Coq 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 [9]. CSL and the A-G method studied in this paper are all adapted to this low-level language. The relationship between the low-level CSL and the original logic by O'Hearn [11, 10] is discussed in Sect. 7.

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

Fig. 1. The Abstract Machine

In the rest of this paper, we first present our low-level language in Sect. 2. We then present an A-G based logic (AGL) for this language in Sect. 3. We extend AGL with local reasoning and propose SAGL in Sect. 4. In Sect. 5, we adapt the original CSL to the low-level language and formalize the relationship between CSL and SAGL. We use two examples to illustrate the use of SAGL in Sect. 6. Finally, we discuss related work and conclude in Sect. 7.

2 The Language

Figure 1 defines the model of an abstract machine and the syntax of the assembly language. The whole program state $\mathbb{P}$ contains a shared memory $\mathbb{M}$, a lock mapping $\mathbb{L}$ which maps a lock to the id of its owner thread, and n threads $[\mathbb{T}_1, \ldots, \mathbb{T}_n]$. The memory is modeled as a finite partial mapping from memory locations $\mathtt{l}$ (natural numbers) to word values (natural numbers). Each thread $\mathbb{T}_i$ contains its own code heap $\mathbb{C}$, register file $\mathbb{R}$, the instruction sequence $\mathbb{I}$ that is currently being executed, and its thread id i.

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. We also use lock/unlock primitives to do synchronization, and use alloc/free to do dynamic memory allocation and free.

The step relation ($\longmapsto$) of program states ($\mathbb{P}$) is defined in Fig. 2. We use the auxiliary relation $(\mathbb{M}, \mathbb{T}, \mathbb{L}) \overset{\mathrm{t}}{\longmapsto} (\mathbb{M}', \mathbb{T}', \mathbb{L}')$ 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*. In Fig. 2 we show operational semantics of representative instructions, which are mostly standard. Note that we do not support reentrant-locks. If the lock l has been acquired, execution of the "lock l" instruction will be blocked even if the lock is owned by the current thread. The relation Next_ι defines the effects of the sequential instruction ι over memory and register files.

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

where

$(\mathbb{M},(\mathbb{C},\mathbb{R},\mathbb{I},k),\mathbb{L}) \overset{\mathsf{t}}{\longmapsto} (\mathbb{M}',\mathbb{T}',\mathbb{L}')$		
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 $\mathbb{I}' = \mathbb{C}(\mathbb{R}(\mathtt{r}_s))$
beq $\mathtt{r}_s,\mathtt{r}_t,\mathtt{f};\mathbb{I}'$	$(\mathbb{M},(\mathbb{C},\mathbb{R},\mathbb{I}',k),\mathbb{L})$ $(\mathbb{M},(\mathbb{C},\mathbb{R},\mathbb{I}'',k),\mathbb{L})$	if $\mathbb{R}(\mathtt{r}_s) \neq \mathbb{R}(\mathtt{r}_t)$ if $\mathbb{R}(\mathtt{r}_s) = \mathbb{R}(\mathtt{r}_t)$ and $\mathbb{I}'' = \mathbb{C}(\mathtt{f})$
lock $l;\mathbb{I}'$	$(\mathbb{M},(\mathbb{C},\mathbb{R},\mathbb{I}',k),\mathbb{L}\{l\rightsquigarrow k\})$ $(\mathbb{M},(\mathbb{C},\mathbb{R},\mathbb{I},k),\mathbb{L})$	if $l \notin dom(\mathbb{L})$ if $l \in dom(\mathbb{L})$
unlock $l;\mathbb{I}'$	$(\mathbb{M},(\mathbb{C},\mathbb{R},\mathbb{I}',k),\mathbb{L}\setminus\{l\})$	if $\mathbb{L}(l) = k$
$\iota;\mathbb{I}'$ 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 $\mathsf{Next}_\iota\ (\mathbb{M},\mathbb{R}) =$	
addi $\mathtt{r}_d,\mathtt{r}_s,i$	$(\mathbb{M},\mathbb{R}\{\mathtt{r}_d\rightsquigarrow\mathbb{R}(\mathtt{r}_s)+i\})$	
ld $\mathtt{r}_t,i(\mathtt{r}_s)$	$(\mathbb{M},\mathbb{R}\{\mathtt{r}_t\rightsquigarrow\mathbb{M}(\mathbb{R}(\mathtt{r}_s)+i)\})$	when $\mathbb{R}(\mathtt{r}_s)+i \in dom(\mathbb{M})$
st $\mathtt{r}_t,i(\mathtt{r}_s)$	$(\mathbb{M}\{\mathbb{R}(\mathtt{r}_s)+i\rightsquigarrow\mathbb{R}(\mathtt{r}_t)\},\mathbb{R})$	when $\mathbb{R}(\mathtt{r}_s)+i \in dom(\mathbb{M})$
alloc $\mathtt{r}_d,\mathtt{r}_s$	$(\mathbb{M}\{\mathtt{l},\ldots,\mathtt{l}+\mathbb{R}(\mathtt{r}_s)-1\rightsquigarrow\_\},\mathbb{R}\{\mathtt{r}_d\rightsquigarrow\mathtt{l}\})$ where $\mathtt{l},\ldots,\mathtt{l}+\mathbb{R}(\mathtt{r}_s)-1 \notin dom(\mathbb{M})$	
free $\mathtt{r}_s$	$(\mathbb{M}\setminus\{\mathbb{R}(\mathtt{r}_s)\},\mathbb{R})$	when $\mathbb{R}(\mathtt{r}_s) \in dom(\mathbb{M})$

Fig. 2. Operational Semantics of the Machine

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

3 AGL: an A-G Based Program Logic

In this section we present an A-G based program logic (AGL) for our assembly language. AGL is a variation of the CCAP logic [19] which applies the A-G method for assembly code verification. Different from CCAP, AGL works for the preemptive thread model instead of the non-preemptive model.

Figure 3 shows the specification constructs for AGL. For each thread in the program, its specification contains three parts: the specification Ψ for the code heap, the assumption A and the guarantee G. The specification Φ of the whole program just groups specifications for each thread. We use CiC, our *meta-logic* mechanized by Coq [16], as the assertion language for assertions and program specifications. CiC corresponds to the higher-order predicate logic with inductive definitions via Curry-Howard isomorphism.

Assumptions and guarantees are meta-logic predicates over a pair of extended thread states $\mathbb{X}$, which contains the shared memory $\mathbb{M}$, the thread's register file $\mathbb{R}$ and id k, and

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

Fig. 3. Specification Constructs for AGL

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

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

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

$$
\frac{\mathtt{a}\,(\mathbb{M},(\mathbb{R},k),\mathbb{L}) \quad \Psi,\mathsf{A},\mathsf{G} \vdash \mathbb{C}:\Psi \quad \Psi,\mathsf{A},\mathsf{G} \vdash \{\mathtt{a}\}\,\mathbb{I}}{\Psi,\mathsf{A},\mathsf{G} \vdash \{\mathtt{a}\}\,(\mathbb{M},(\mathbb{C},\mathbb{R},\mathbb{I},k),\mathbb{L})}\ (\textsc{thrd})
$$

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

$$
\frac{\forall \mathtt{f} \in \mathit{dom}(\Psi') : \quad \Psi,\mathsf{A},\mathsf{G} \vdash \{\Psi'(\mathtt{f})\}\,\mathbb{C}(\mathtt{f})}{\Psi,\mathsf{A},\mathsf{G} \vdash \mathbb{C}:\Psi'}\ (\textsc{cdhp})
$$

Fig. 4. AGL Inference Rules

---

the global lock mapping $\mathbb{L}$. The assumption A for a thread specifies the expected invariant of state transitions made by the environment. The arguments it takes are states before and after a transition, respectively. The guarantee G of a thread specifies the invariant of state transitions made by the thread.

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

Inference rules. Inference rules of AGL are presented in Figs. 4 and 5. The PROG rule defines the well-formedness of the program $\mathbb{P}$ with respect to the program specification Φ and the set of preconditions $([\mathtt{a}_1,\ldots,\mathtt{a}_n])$ for the instruction sequences that are currently executed by all the threads. Checking the well-formedness of $\mathbb{P}$ involves two steps. First we check the compatibility of assumptions and guarantees for all the threads. The predicate NI is defined as follows:

$$
\begin{array}{l}
\mathsf{NI}([(\mathsf{A}_1,\mathsf{G}_1),\ldots,(\mathsf{A}_n,\mathsf{G}_n)]) \stackrel{\text{def}}{=} \forall i,j,\mathbb{M},\mathbb{M}',\mathbb{R}_i,\mathbb{R}_i',\mathbb{R}_j,\mathbb{L},\mathbb{L}'. \\
\qquad i \neq j \to \mathsf{G}_i\,(\mathbb{M},(\mathbb{R}_i,i),\mathbb{L})\,(\mathbb{M}',(\mathbb{R}_i',i),\mathbb{L}') \to \mathsf{A}_j\,(\mathbb{M},(\mathbb{R}_j,j),\mathbb{L})\,(\mathbb{M}',(\mathbb{R}_j,j),\mathbb{L}'),
\end{array} \tag{1}
$$

which simply says that the guarantee of each thread should satisfy assumptions of all

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

$$\frac{\Psi, \mathsf{A}, \mathsf{G} \vdash \{\mathtt{a}\}\,\mathtt{i}\,\{\mathtt{a}'\} \quad \Psi, \mathsf{A}, \mathsf{G} \vdash \{\mathtt{a}'\}\mathbb{I} \quad (\mathtt{a} \circ \mathsf{A}) \Rightarrow \mathtt{a}}{\Psi, \mathsf{A}, \mathsf{G} \vdash \{\mathtt{a}\}\,\mathtt{i};\mathbb{I}}\ (\textsc{seq})$$

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

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

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

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

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

Fig. 5. AGL Inference Rules (cont'd)

other threads. Then we apply the THRD rule to check that implementation of each thread actually satisfies the specification. Each thread $\mathbb{T}_i$ is verified separately. therefore thread modularity is supported.

In the THRD rule, we require that the precondition $\mathtt{a}$ be satisfied by the current extended thread state $(\mathbb{M}, (\mathbb{R}, k), \mathbb{L})$; that the thread code heap satisfy its specification Ψ, A and G; and that it be safe to execute the current instruction sequence $\mathbb{I}$ under the precondition $\mathtt{a}$ and the thread specification.

The CDHP rule checks the well-formedness of thread code heaps. It requires that each instruction sequence specified in Ψ' be well-formed with respect to the imported interfaces specified in Ψ, the assumption A and the guarantee G.

The SEQ rule and the JR rule ensure that it is safe to execute the instruction sequence if the precondition is satisfied. If the instruction sequence starts with a normal sequential instruction $\mathtt{i}$, we need to come up with an assertion $\mathtt{a}'$ which serves both as the postcondition of $\mathtt{i}$ and as the precondition of the remaining instruction sequence. Also we need to ensure that, if the current thread is preempted at a state satisfying $\mathtt{a}$, $\mathtt{a}$ must be preserved by any state transitions (by other threads) satisfying the assumption A. This is enforced by $(\mathtt{a} \circ \mathsf{A}) \Rightarrow \mathtt{a}$:

$$(\mathtt{a} \circ \mathsf{A}) \Rightarrow \mathtt{a} \ \stackrel{\text{def}}{=}\ \forall \mathbb{X}, \mathbb{X}'.\ \mathtt{a}\,\mathbb{X} \wedge \mathsf{A}\,\mathbb{X}\,\mathbb{X}' \rightarrow \mathtt{a}\,\mathbb{X}'.$$

If we reach the last jump instruction of the instruction sequence, the JR rule requires that the assertion assigned to the target address in Ψ be satisfied after the jump. It also

$$(\textit{CdHpSpec}) \quad \Psi \quad ::= \quad \{\mathtt{f} \rightsquigarrow (\mathtt{a}, \mathtt{v})\}^*$$
$$(\textit{Assertion}) \quad \mathtt{a}, \mathtt{v} \quad \in \quad \textit{XState} \to \textit{Prop}$$

Fig. 6. Extension of AGL Specification Constructs in SAGL

requires that a be preserved by state transitions satisfying A. Here we use the syntactic sugar $\forall X@(x_1,\ldots,x_n).\, P(X,x_1,\ldots,x_n)$ to mean that, for all tuple X containing elements $x_1,\ldots,x_n$, the predicate P holds. It is formally defined as:

$$\forall X, x_1,\ldots,x_n.(X = (x_1,\ldots,x_n)) \to P(X,x_1,\ldots,x_n)\,.$$

The notation $\lambda X@(x_1,\ldots,x_n).\, f(X,x_1,\ldots,x_n)$ that we use later is defined similarly. The rule for direct jumps (j f) is similar to the JR rule and is not presented here.

Instruction rules require that the precondition ensure the safe execution of the instruction; and that the resulting state satisfy the postcondition. Also, if shared states ($\mathbb{M}$ and $\mathbb{L}$) are updated by the instruction, we need to ensure that the update satisfies the guarantee G. For the lock instruction, if the control falls through, we know that the lock is not held by any thread. This extra knowledge can be used together with the precondition a to show the postcondition is satisfied by the resulting state. The rest of instruction rules are straightforward and will not be explained here. Interested readers can refer to the companion technical report [4] for a complete presentation of instruction rules.

The soundness of AGL is also formalized in the technical report [4], which is similar to the soundness theorem of SAGL presented in Sect. 4.

4 SAGL: Separated A-G Logic

AGL is a general program logic supporting thread modular verification of concurrent code. However, because it treats all memory as shared resources, it does not have good memory modularity, and assumptions and guarantees are hard to define and use. During program verification, we have to prove for each individual instruction that the guarantee is not broken, even if there is no memory sharing. Moreover, if each thread dynamically allocates memory and uses allocated memory as private resources (see the example in Sect. 6), the domain of memory becomes dynamic and nondeterministic, which makes it very hard to specify the assumption and guarantee.

In this section, we extend AGL with explicit partition of private resources and shared resources. The extended logic, which we call Separated A-G Logic (SAGL), has much better support of memory modularity than AGL without sacrificing any expressiveness. Borrowing the local-reasoning idea in separation logic, private resources of one thread are not visible to other threads, therefore will not be touched by others. Assumptions and guarantees in SAGL only specify shared resources. The dynamic domain of private memory caused by memory allocation is no longer a challenge to define assumptions and guarantees because private memory does not have to be specified.

Figure 6 shows our extensions of AGL specifications for SAGL. In the specification Ψ of each thread code heap, the precondition assigned to each code label now becomes a pair of assertions $(\mathtt{a}, \mathtt{v})$. The assertion a plays the same role as in AGL. It specifies the shared resources (all memory are treated as shared in AGL). The assertion v specifies the private resources of the thread. Other threads' private resources are not specified.

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

$$\frac{\begin{array}{l}\Phi = ([\Psi_1, \ldots, \Psi_n], [(\mathsf{A}_1, \mathsf{G}_1), \ldots, (\mathsf{A}_n, \mathsf{G}_n)]) \quad \mathsf{NI}([(\mathsf{A}_1, \mathsf{G}_1), \ldots, (\mathsf{A}_n, \mathsf{G}_n)]) \\ \mathbb{M}_s \uplus \mathbb{M}_1 \uplus \cdots \uplus \mathbb{M}_n = \mathbb{M} \quad \Psi_k, \mathsf{A}_k, \mathsf{G}_k \vdash \{(\mathsf{a}_k, \mathsf{v}_k)\}\,(\mathbb{M}_s, \mathbb{M}_k, \mathbb{T}_k, \mathbb{L}) \;\text{ for all } k\end{array}}{\Phi, [(\mathsf{a}_1, \mathsf{v}_1), \ldots, (\mathsf{a}_n, \mathsf{v}_n)] \vdash (\mathbb{M}, [\mathbb{T}_1, \ldots, \mathbb{T}_n], \mathbb{L})}\ (\textsc{prog})$$

$\boxed{\Psi, \mathsf{A}, \mathsf{G} \vdash \{(\mathsf{a}, \mathsf{v})\}\,(\mathbb{M}_s, \mathbb{M}_v, \mathbb{T}, \mathbb{L})}$ **(*Well-formed thread*)**

$$\frac{\mathsf{a}\,(\mathbb{M}_s, (\mathbb{R}, k), \mathbb{L}) \quad \mathsf{v}\,(\mathbb{M}_v, (\mathbb{R}, k), \mathbb{L}|_k) \quad \Psi, \mathsf{A}, \mathsf{G} \vdash \mathbb{C} : \Psi \quad \Psi, \mathsf{A}, \mathsf{G} \vdash \{(\mathsf{a}, \mathsf{v})\}\,\mathbb{I}}{\Psi, \mathsf{A}, \mathsf{G} \vdash \{(\mathsf{a}, \mathsf{v})\}\,(\mathbb{M}_s, \mathbb{M}_v, (\mathbb{C}, \mathbb{R}, \mathbb{I}, k), \mathbb{L})}\ (\textsc{thrd})$$

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

$$\frac{\forall \mathtt{f} \in \mathit{dom}(\Psi') : \quad \Psi, \mathsf{A}, \mathsf{G} \vdash \{\Psi'(\mathtt{f})\}\,\mathbb{C}(\mathtt{f})}{\Psi, \mathsf{A}, \mathsf{G} \vdash \mathbb{C} : \Psi'}\ (\textsc{cdhp})$$

Fig. 7. SAGL Inference Rules

Inference rules. The inference rules of SAGL are shown in Figs. 7 and 8. They look very similar to AGL rules. In the PROG rule, as in AGL, we check the compatibility of assumptions and guarantees, and check the well-formedness of each thread. However, here we require that there be a partition of memory into $n+1$ parts: one part $\mathbb{M}_s$ is shared and other parts $\mathbb{M}_1, \ldots, \mathbb{M}_n$ are privately owned by the threads $\mathbb{T}_1, \ldots, \mathbb{T}_n$, respectively. When we check the well-formedness of thread $\mathbb{T}_k$, the memory in the extended thread state is not the global memory. It just contains $\mathbb{M}_s$ and $\mathbb{M}_k$.

The THRD rule in SAGL is similar to the one in AGL, except that the memory visible by each thread is separated into two parts: the shared $\mathbb{M}_s$ and the private $\mathbb{M}_v$. We require that assertions a and v hold over $\mathbb{M}_s$ and $\mathbb{M}_v$ respectively. Since v only specifies the private resource, we use the "filter" operator $\mathbb{L}|_k$ to prevent v from having access to the ownership information of locks not owned by the current thread:

$$(\mathbb{L}|_k)(l) \stackrel{\text{def}}{=} \begin{cases} k & \mathbb{L}(l) = k \\ \mathit{undefined} & \text{otherwise} \end{cases} \tag{2}$$

i.e., $\mathbb{L}|_k$ is a subset of $\mathbb{L}$ which maps locks to k.

Instruction rules are shown in Fig. 8. In the SEQ rule, we use (a, v) as the precondition. However, to ensure that the precondition is preserved by state transitions satisfying A, we only check a (*i.e.,* we check $(\mathsf{a} \circ \mathsf{A}) \Rightarrow \mathsf{a}$) because A only specifies shared resources. We know that the private resources will not be touched by the environment. We require a to be precise to enforce the unique boundary between shared and private resources. Following the definition in CSL [11], an assertion a is precise if and only if for any memory $\mathbb{M}$, there is at most one subset $\mathbb{M}'$ that satisfies a, *i.e.,*

$$\begin{array}{l}\mathsf{Precise}(\mathsf{a}) \stackrel{\text{def}}{=} \forall \mathbb{M}, \mathbb{R}, k, \mathbb{L}, \mathbb{M}_1, \mathbb{M}_2.\ (\mathbb{M}_1 \subseteq \mathbb{M}) \wedge (\mathbb{M}_2 \subseteq \mathbb{M}) \wedge \\ \qquad\qquad \mathsf{a}\,(\mathbb{M}_1, (\mathbb{R}, k), \mathbb{L}) \wedge \mathsf{a}\,(\mathbb{M}_2, (\mathbb{R}, k), \mathbb{L}) \rightarrow \mathbb{M}_1 = \mathbb{M}_2\,.\end{array} \tag{3}$$

The JR rule requires a be precise and it be preserved by state transitions satisfying the assumption. Also, the specification assigned to the target address needs to be satisfied by the resulting state of the jump, and the identity state transition made by the jump

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

$$\frac{\Psi, \mathsf{A}, \mathsf{G} \vdash \{(\mathsf{a}, \nu)\}\,\imath\,\{(\mathsf{a}', \nu')\} \quad \Psi, \mathsf{A}, \mathsf{G} \vdash \{(\mathsf{a}', \nu')\}\,\mathbb{I} \quad (\mathsf{a} \circ \mathsf{A}) \Rightarrow \mathsf{a} \quad \mathsf{Precise}(\mathsf{a})}{\Psi, \mathsf{A}, \mathsf{G} \vdash \{(\mathsf{a}, \nu)\}\,\imath;\mathbb{I}} \text{ (SEQ)}$$

$$\frac{\mathsf{Precise}(\mathsf{a}) \quad (\mathsf{a} \circ \mathsf{A}) \Rightarrow \mathsf{a} \quad \forall \mathbb{X}@(\mathbb{M}, (\mathbb{R}, k), \mathbb{L}).\ (\mathsf{a} * \nu)\ \mathbb{X} \rightarrow (\mathsf{a}' * \nu')\ \mathbb{X} \wedge (\lfloor \mathsf{G} \rfloor_{(\mathsf{a}, \mathsf{a}')}\ \mathbb{X}\ \mathbb{X}) \text{ where } (\mathsf{a}', \nu') = \Psi(\mathbb{R}(\mathtt{r}_s))}{\Psi, \mathsf{A}, \mathsf{G} \vdash \{(\mathsf{a}, \nu)\}\,\mathsf{jr}\ \mathtt{r}_s} \text{ (JR)}$$

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

$$\frac{\forall \mathbb{X}@(\mathbb{M}, (\mathbb{R}, k), \mathbb{L}).\ (\mathsf{a} * \nu)\ \mathbb{X} \wedge l \notin \mathit{dom}(\mathbb{L}) \rightarrow (\mathsf{a}' * \nu')\ \mathbb{X}' \wedge (\lfloor \mathsf{G} \rfloor_{(\mathsf{a}, \mathsf{a}')}\ \mathbb{X}\ \mathbb{X}') \text{ where } \mathbb{X}' = (\mathbb{M}, (\mathbb{R}, k), \mathbb{L}\{l \rightsquigarrow k\})}{\Psi, \mathsf{A}, \mathsf{G} \vdash \{(\mathsf{a}, \nu)\}\,\mathsf{lock}\ l\,\{(\mathsf{a}', \nu')\}} \text{ (LOCK)}$$

$$\frac{\forall \mathbb{X}@(\mathbb{M}, (\mathbb{R}, k), \mathbb{L}).\ (\mathsf{a} * \nu)\ \mathbb{X} \rightarrow \mathbb{L}(l) = k \wedge (\mathsf{a}' * \nu')\ \mathbb{X}' \wedge (\lfloor \mathsf{G} \rfloor_{(\mathsf{a}, \mathsf{a}')}\ \mathbb{X}\ \mathbb{X}') \text{ where } \mathbb{X}' = (\mathbb{M}, (\mathbb{R}, k), \mathbb{L} \setminus \{l\})}{\Psi, \mathsf{A}, \mathsf{G} \vdash \{(\mathsf{a}, \nu)\}\,\mathsf{unlock}\ l\,\{(\mathsf{a}', \nu')\}} \text{ (UNLOCK)}$$

Fig. 8. SAGL Inference Rules (cont'd)

satisfies the guarantee G. We use the separating conjunction of the shared and private predicates as the pre- and post-condition. We define $\mathsf{a} * \nu$ as:

$$\mathsf{a} * \nu \stackrel{\text{def}}{=} \lambda(\mathbb{M}, (\mathbb{R}, k), \mathbb{L}). \exists \mathbb{M}_1, \mathbb{M}_2.(\mathbb{M}_1 \uplus \mathbb{M}_2 = \mathbb{M}) \wedge \mathsf{a}\ (\mathbb{M}_1, (\mathbb{R}, k), \mathbb{L}) \wedge \nu\ (\mathbb{M}_2, (\mathbb{R}, k), \mathbb{L}|_k)\,. \tag{4}$$

Again, the use of $\mathbb{L}|_k$ prevents ν from having access to the ownership information of locks not owned by the current thread. We use $f_1 \uplus f_2$ to represent the union of finite partial mappings with disjoint domains.

To ensure G is satisfied over shared resources, we lift G to $\lfloor \mathsf{G} \rfloor_{(\mathsf{a}, \mathsf{a}')}$:

$$\begin{aligned} \lfloor \mathsf{G} \rfloor_{(\mathsf{a}, \mathsf{a}')} \stackrel{\text{def}}{=}\ & \lambda \mathbb{X}@(\mathbb{M}, (\mathbb{R}, k), \mathbb{L}), \mathbb{X}'@(\mathbb{M}', (\mathbb{R}', k'), \mathbb{L}'). \\ & \exists \mathbb{M}_1, \mathbb{M}_2, \mathbb{M}'_1, \mathbb{M}'_2.\ (\mathbb{M}_1 \uplus \mathbb{M}_2 = \mathbb{M}) \wedge (\mathbb{M}'_1 \uplus \mathbb{M}'_2 = \mathbb{M}') \\ & \qquad \wedge \mathsf{a}\ (\mathbb{M}_1, (\mathbb{R}, k), \mathbb{L}) \wedge \mathsf{a}'\ (\mathbb{M}'_1, (\mathbb{R}', k'), \mathbb{L}') \\ & \qquad \wedge \mathsf{G}\ (\mathbb{M}_1, (\mathbb{R}, k), \mathbb{L})\ (\mathbb{M}'_1, (\mathbb{R}', k'), \mathbb{L}')\,, \end{aligned} \tag{5}$$

Here we use precise predicates a and a' to enforce the unique boundary between shared and private resources.

As expected, the SAGL rule for each individual instruction is almost the same as its counterpart in AGL, except that we always use the separating conjunction of predicates for shared and private resources. Each instruction rule requires that memory in states before and after the transition can be partitioned to private and shared; private parts satisfy private predicates and shared parts satisfy shared predicates and G.

It is important that we always combine shared predicates with private predicates instead of checking separately the relationship between a and a' and between ν and ν'. This gives us the ability to support *dynamic redistribution* of private and shared

$$
\begin{array}{rrcl}
(\textit{ProgSpec}) & \phi & ::= & ([\psi_1,\ldots,\psi_n],\Gamma) \\
(\textit{CdHpSpec}) & \psi & ::= & \{\mathtt{f} \rightsquigarrow \nu\}^* \\
(\textit{ResourceINV}) & \Gamma & \in & \textit{Locks} \rightharpoonup \textit{MemPred} \\
(\textit{MemPred}) & \mathtt{m} & \in & \textit{Memory} \to \textit{Prop}
\end{array}
$$

Fig. 9. Specification Constructs for CSL

memory. Instead of enforcing static partition, we allow that part of private memory becomes shared under certain conditions and vice versa. As we will show in the next section, this ability makes our SAGL very expressive and is the enabling feature that makes the embedding of CSL into SAGL possible.

AGL can be viewed as a specialized version of SAGL where all the ν's are set to emp (emp is an assertion which can only be satisfied by memory with empty domain).

Soundness. The soundness of SAGL is formulated in Theorem 1. In addition to the safety of well-formed programs, it also characterizes partial correctness: assertions assigned to labels in Ψ will hold whenever the labels are reached. Theorem 1 is proved following the syntactic approach to type soundness [18]. Here we only present the main theorem. The proof is given in our technical report and is formalized in Coq [4].

Theorem 1 (SAGL-Soundness). *For any program $\mathbb{P}$ with specification $\Phi = ([\Psi_1,\ldots,\Psi_n],[(\mathsf{A}_1,\mathsf{G}_1),\ldots,(\mathsf{A}_n,\mathsf{G}_n)])$, if $\Phi,[(a_1,\nu_1)\ldots,(a_n,\nu_n)] \vdash \mathbb{P}$, then,*

- *for any natural number m, there exists $\mathbb{P}'$ such that $(\mathbb{P} \longmapsto^m \mathbb{P}')$;*
- *for any m and $\mathbb{P}' = (\mathbb{M}',[\mathbb{T}'_1,\ldots,\mathbb{T}'_n],\mathbb{L}')$, if $(\mathbb{P} \longmapsto^m \mathbb{P}')$, then,*
  - *$\Phi,[(a'_1,\nu'_1),\ldots,(a'_n,\nu'_n)] \vdash \mathbb{P}'$ for some $a'_1,\ldots,a'_n$ and $\nu'_1,\ldots,\nu'_n$;*
  - *for any k, there exist $\mathbb{M}''$, $\mathbb{T}''_k$ and $\mathbb{L}''$ such that $(\mathbb{M}',\mathbb{T}'_k,\mathbb{L}') \xmapsto{\mathsf{t}} (\mathbb{M}'',\mathbb{T}''_k,\mathbb{L}'')$;*
  - *for any k, if $\mathbb{T}'_k = (\mathbb{C}_k,\mathbb{R}'_k,\mathsf{jr}\ r_s,k)$, then $(a''_k * \nu''_k)\ (\mathbb{M}',(\mathbb{R}'_k,k),\mathbb{L}')$ holds, where $(a''_k,\nu''_k) = \Psi_k(\mathbb{R}'_k(r_s))$;*
  - *for any k, if $\mathbb{T}'_k = (\mathbb{C}_k,\mathbb{R}'_k,\mathsf{bgt}\ r_s,r_t,f;\mathbb{I},k)$ and $\mathbb{R}'_k(r_s) > \mathbb{R}'_k(r_t)$, then $(a''_k * \nu''_k)\ (\mathbb{M}',(\mathbb{R}'_k,k),\mathbb{L}')$ holds, where $(a''_k,\nu''_k) = \Psi_k(f)$;*

5 Concurrent Separation Logic (CSL)

Both AGL and SAGL treat lock/unlock primitives as normal instructions. They do not require that shared memory be protected by locks. This shows the generality of the A-G method, which makes no assumption about language constructs for synchronizations. Any ad-hoc synchronizations can be verified using the A-G method.

If we focus on a special class of programs following Hoare [6] where accesses of shared resources are protected by critical regions (implemented by locks in our language), we can further simplify our SAGL logic and derive a variation of CSL (CSL adapted to our assembly language).

5.1 CSL Specifications and Rules

In CSL, shared memory is partitioned and each part is protected by a unique lock. For each part of the partition, an invariant is assigned to specify its well-formedness.

$$\mathtt{m} * \mathtt{m}' \stackrel{\text{def}}{=} \lambda\mathbb{M}.\exists\mathbb{M}_1,\mathbb{M}_2.\ (\mathbb{M}_1 \uplus \mathbb{M}_2 = \mathbb{M}) \wedge \mathtt{m}\ \mathbb{M}_1 \wedge \mathtt{m}'\ \mathbb{M}_2$$

$$\mathtt{v} * \mathtt{m} \stackrel{\text{def}}{=} \lambda\mathbb{X}@(\mathbb{M},(\mathbb{R},k),\mathbb{L}).\exists\mathbb{M}_1,\mathbb{M}_2.\ (\mathbb{M}_1 \uplus \mathbb{M}_2 = \mathbb{M}) \wedge \mathtt{v}\ (\mathbb{M}_1,(\mathbb{R},k),\mathbb{L}) \wedge \mathtt{m}\ \mathbb{M}_2$$

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

$$\mathsf{acq}\ l\ \mathtt{v} \stackrel{\text{def}}{=} \lambda(\mathbb{M},(\mathbb{R},k),\mathbb{L}).\ \mathtt{v}\ (\mathbb{M},(\mathbb{R},k),\mathbb{L}\{l \rightsquigarrow k\})$$

$$\mathsf{rel}\ l\ \mathtt{v} \stackrel{\text{def}}{=} \lambda(\mathbb{M},(\mathbb{R},k),\mathbb{L}).\ \mathbb{L}(l) = k \wedge \mathtt{v}\ (\mathbb{M},(\mathbb{R},k),\mathbb{L} \setminus \{l\})$$

Fig. 10. Definitions of Notations in CSL

A thread cannot access shared memory unless it has acquired the corresponding lock. After the lock is acquired, the thread takes advantage of mutual-exclusion provided by locks and treats the part of memory as private. When the thread releases the lock, it must ensure that the part of memory is well-formed with regard to the corresponding invariant. In this way the following global invariant is enforced:

Shared resources are well-formed outside critical regions.

Figure 9 shows the specification constructs for CSL. The program specification ϕ contains a collection of code heap specifications for each thread and the specification Γ for lock-protected memory. Code heap specification ψ maps a code label to an assertion $\mathtt{v}$ as the precondition of the corresponding instruction sequence. Here $\mathtt{v}$ plays similar role of the private predicate in SAGL. Since each thread privately owns the lock protected memory if it owns the lock, all memory accessible by a thread is viewed as private memory. Therefore we do not need an assertion $\mathtt{a}$ to specify the shared memory as we did in SAGL. This also explains why we do not need assumptions and guarantees in CSL. The specification Γ of lock-protected memory maps a lock to an invariant $\mathtt{m}$, which specifies the corresponding part of memory. The invariant $\mathtt{m}$ is simply a predicate over memory because the register file is private to each thread.

Inference rules. The inference rules for CSL are presented in Fig. 11. The PROG rule requires that there be a partition of the global memory into $n+1$ parts. Each $\mathbb{M}_k$ is privately owned by thread $\mathbb{T}_k$. The well-formedness of $\mathbb{T}_k$ is checked by applying the THRD rule. $\mathbb{M}_s$ is the part of memory protected by free locks (locks not owned by any threads). It must satisfy the invariants specified in Γ. Here $\mathtt{a}_\Gamma$ is the separating conjunction of invariants assigned to free locks in Γ, which is defined as:

$$\mathtt{a}_\Gamma \stackrel{\text{def}}{=} \lambda(\mathbb{M},(\mathbb{R},k),\mathbb{L}).\ (\forall_* l \in (dom(\Gamma) - dom(\mathbb{L})).\ \Gamma(l))\ \mathbb{M}, \tag{6}$$

that is, *shared resources are well-formed outside of critical regions*. Here $\forall_*$ is an indexed, finitely iterated separating conjunction, which is formalized in Fig. 10. Separating conjunctions with memory predicates ($\mathtt{v} * \mathtt{m}$ and $\mathtt{m} * \mathtt{m}'$) are also defined in Fig. 10. As in O'Hearn's original work on CSL [11], we also require all invariants specified in Γ to be precise, *i.e.*, $\mathsf{Precise}(\Gamma)$.

The THRD rule checks the well-formedness of threads. It requires that the current extended thread state satisfies the precondition $\mathtt{v}$. Since $\mathtt{v}$ only cares about the resource privately owned by the thread, it takes $\mathbb{L}|_k$ instead of complete $\mathbb{L}$ as argument. Recall

$\boxed{\phi, [\nu_1, \ldots, \nu_n] \vdash \mathbb{P}}$ **(*Well-formed program*)**

$$\frac{\begin{array}{l}\phi = ([\psi_1, \ldots, \psi_n], \Gamma) \qquad \mathbb{M}_s \uplus \mathbb{M}_1 \uplus \cdots \uplus \mathbb{M}_n = \mathbb{M} \\ \mathtt{a}_\Gamma\,(\mathbb{M}_s, \_, \mathbb{L}) \quad \mathsf{Precise}(\Gamma) \quad \psi_k, \Gamma \vdash \{\nu_k\}\,(\mathbb{M}_k, \mathbb{T}_k, \mathbb{L}) \ \text{ for all } k\end{array}}{\phi, [\nu_1, \ldots, \nu_n] \vdash (\mathbb{M}, [\mathbb{T}_1, \ldots, \mathbb{T}_n], \mathbb{L})}\ (\textsc{prog})$$

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

$$\frac{\nu\,(\mathbb{M}, (\mathbb{R}, k), \mathbb{L}|_k) \quad \psi, \Gamma \vdash \mathbb{C} : \psi \quad \psi, \Gamma \vdash \{\nu\}\,\mathbb{I}}{\psi, \Gamma \vdash \{\nu\}\,(\mathbb{M}, (\mathbb{C}, \mathbb{R}, \mathbb{I}, k), \mathbb{L})}\ (\textsc{thrd})$$

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

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

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

$$\frac{\psi, \Gamma \vdash \{\nu\}\,\iota\,\{\nu'\} \quad \psi, \Gamma \vdash \{\nu'\}\,\mathbb{I}}{\psi, \Gamma \vdash \{\nu\}\,\iota; \mathbb{I}}\ (\textsc{seq}) \qquad \frac{\forall \mathbb{X} @ (\mathbb{M}, (\mathbb{R}, k), \mathbb{L}).\ \nu\,\mathbb{X} \to \psi(\mathbb{R}(\mathtt{r}_s))\,\mathbb{X}}{\psi, \Gamma \vdash \{\nu\}\,\mathsf{jr}\ \mathtt{r}_s}\ (\textsc{jr})$$

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

$$\frac{\nu * \mathtt{m} \Rightarrow \mathsf{acq}\ l\ \nu'}{\psi, \Gamma\{l \rightsquigarrow \mathtt{m}\} \vdash \{\nu\}\,\mathsf{lock}\ l\,\{\nu'\}}\ (\textsc{lock}) \qquad \frac{\nu \Rightarrow (\mathsf{rel}\ l\ \nu') * \mathtt{m}}{\psi, \Gamma\{l \rightsquigarrow \mathtt{m}\} \vdash \{\nu\}\,\mathsf{unlock}\ l\,\{\nu'\}}\ (\textsc{unlock})$$

Fig. 11. CSL Inference Rules

that $\mathbb{L}|_k$ is defined in (2) in Section 4 to represent the subset of $\mathbb{L}$ which maps locks to k. The CDHP rule and rules for instruction sequences are similar to their counterparts in AGL and SAGL and require no more explanation.

In the LOCK rule, we use "$\mathsf{acq}\ l\ \nu'$" to represent the weakest precondition of ν'; and "$\nu \Rightarrow \nu'$" for logical implication lifted for state predicates. They are formalized in Fig. 10. If the lock l instruction successfully acquires the lock l, we know by our global invariant that the part of memory protected by l satisfies the invariant $\Gamma(l)$ (*i.e.*, $\mathtt{m}$), because l is a free lock before lock l is executed. Therefore, we can carry the knowledge $\mathtt{m}$ in the postcondition ν'. Also, carrying $\mathtt{m}$ in ν' allows subsequent instructions to access that part of memory, since separation logic predicates capture ownerships of memory.

In the UNLOCK rule, "$\mathsf{rel}\ l\ \nu'$" is the weakest precondition for ν' (see Fig. 10). At the time the lock l is released, the memory protected by l must be well formed with respect to $\mathtt{m} = \Gamma(l)$. The separating conjunction here ensures that ν' does not specify this part of memory. Therefore the following instructions cannot use the part of memory unless the lock is acquired again.

The complete set of rules are presented in the technical report [4]. The frame rule, conjunction rule and consequence rule are admissible in our CSL. These rules and the proof of their admissibility can be found in the report [4] too.

5.2 Interpretation of CSL in SAGL

We prove the soundness of CSL by giving it an interpretation in SAGL, and proving CSL rules as derivable lemmas. This interpretation also formalizes the specialization made for CSL to achieve the simplicity.

From SAGL's point of view, each thread has two parts of memory: the private and the shared. In CSL, the private memory of a thread includes the memory protected by locks held by the thread and the memory that will never be shared. The shared memory are the parts protected by free locks. Therefore, we can use the following interpretation to translate a CSL specification to a SAGL specification:

$$[\![\nu]\!]_\Gamma \stackrel{\text{def}}{=} (\mathtt{a}_\Gamma, \nu) \tag{7}$$

$$[\![\psi]\!]_\Gamma \stackrel{\text{def}}{=} \lambda \mathtt{f}.[\![\psi(\mathtt{f})]\!]_\Gamma \text{ if } \mathtt{f} \in dom(\psi), \tag{8}$$

where $\mathtt{a}_\Gamma$ formalizes the CSL invariant and is defined by (6). We just reuse CSL specification ν as the specification of private memory, and use the separating conjunction $\mathtt{a}_\Gamma$ of invariants assigned to free locks as the specification for shared memory.

Since the assumption and guarantee in SAGL only specifies shared memory, we can define A_Γ and G_Γ for CSL threads:

$$\mathsf{A}_\Gamma \stackrel{\text{def}}{=} \lambda \mathbb{X}@(\mathbb{M},(\mathbb{R},k),\mathbb{L}),\mathbb{X}'@(\mathbb{M}',(\mathbb{R}',k'),\mathbb{L}').\mathbb{R}=\mathbb{R}'\wedge k=k'\wedge(\mathtt{a}_\Gamma\,\mathbb{X}\rightarrow\mathtt{a}_\Gamma\,\mathbb{X}') \tag{9}$$

$$\mathsf{G}_\Gamma \stackrel{\text{def}}{=} \lambda \mathbb{X}@(\mathbb{M},(\mathbb{R},k),\mathbb{L}),\mathbb{X}'@(\mathbb{M}',(\mathbb{R}',k'),\mathbb{L}').\,k=k'\wedge\mathtt{a}_\Gamma\,\mathbb{X}\wedge\mathtt{a}_\Gamma\,\mathbb{X}' \tag{10}$$

which enforces the invariant $\mathtt{a}_\Gamma$ of shared memory.

With above interpretations, we can prove the following soundness theorem.

Theorem 2 (CSL-Soundness).

1. *If* $\psi,\Gamma\vdash\{\nu\}\,\mathfrak{c}\,\{\nu'\}$ *in CSL, then* $[\![\psi]\!]_\Gamma,\mathsf{A}_\Gamma,\mathsf{G}_\Gamma\vdash\{[\![\nu]\!]_\Gamma\}\,\mathfrak{c}\,\{[\![\nu']\!]_\Gamma\}$ *in SAGL;*
2. *If* $\psi,\Gamma\vdash\{\nu\}\,\mathbb{I}$ *in CSL and* $\mathsf{Precise}(\Gamma)$*, then* $[\![\psi]\!]_\Gamma,\mathsf{A}_\Gamma,\mathsf{G}_\Gamma\vdash\{[\![\nu]\!]_\Gamma\}\,\mathbb{I}$ *in SAGL;*
3. *If* $\psi,\Gamma\vdash\mathbb{C}:\psi'$ *in CSL and* $\mathsf{Precise}(\Gamma)$*, then* $[\![\psi]\!]_\Gamma,\mathsf{A}_\Gamma,\mathsf{G}_\Gamma\vdash\mathbb{C}:[\![\psi']\!]_\Gamma$ *in SAGL;*
4. *If* $\psi,\Gamma\vdash\{\nu\}\,(\mathbb{M}_k,\mathbb{T}_k,\mathbb{L})$ *in CSL,* $\mathsf{Precise}(\Gamma)$*, and* $a_\Gamma\,(\mathbb{M}_s,\_,\mathbb{L})$*, then* $[\![\psi]\!]_\Gamma,\mathsf{A}_\Gamma,\mathsf{G}_\Gamma\vdash\{[\![\nu]\!]_\Gamma\}\,(\mathbb{M}_s,\mathbb{M}_k,\mathbb{T}_k,\mathbb{L})$ *in SAGL;*
5. *If* $([\psi_1,\ldots,\psi_n],\Gamma),[\nu_1,\ldots,\nu_n]\vdash\mathbb{P}$ *in CSL, then* $\Phi,[[\![\nu_1]\!]_\Gamma,\ldots,[\![\nu_n]\!]_\Gamma]\vdash\mathbb{P}$ *in SAGL, where* $\Phi=([[\![\psi_1]\!]_\Gamma,\ldots,[\![\psi_n]\!]_\Gamma],[(\mathsf{A}_\Gamma,\mathsf{G}_\Gamma),\ldots,(\mathsf{A}_\Gamma,\mathsf{G}_\Gamma)])$.

6 SAGL Examples

We use two complementary examples to demonstrate how SAGL combines merits of AGL and CSL. Figure 12 shows a simple program, which allocates a fresh memory cell and then writes into and reads from it. Following the MIPS convention, we assume the register $\mathtt{r}_0$ always contains 0. The corresponding high-level pseudo code is given as comments (followed by "; ;"). It is obvious that two threads executing the same code (but may use different m) will never interfere with each other, therefore the test in line (7) is always True and the program never reaches the unsafe branch.

It is trivial to certify the code in CSL since there is no memory-sharing at all. However, due to the nondeterministic operation of the alloc instruction, it is challenging to

```
(1)    start:  -{(emp, emp)}
(2)            addi  r1, r0, 1                  ;; local int x, y;
(3)            alloc r2, r1                     ;; x := alloc(1);
               -{(emp, r2 ↦ _)}
(4)            addi  r1, r0, m
(5)            st    r1, 0(r2)                  ;; [x] := m;
               -{(emp, (r2 ↦ m) ∧ r1 = m)}
(6)            ld    r3, 0(r2)                  ;; y := [x];
               -{(emp, (r2 ↦ m) ∧ r1 = m ∧ r3 = m)}
(7)            beq   r1, r3, safe               ;; while(y == m){}
(8)    unsafe: -{(emp, False)}
(9)            free  r0                         ;; free(0);  (* unsafe! *)
(10)   safe:   -{(emp, r2 ↦ _)}
(11)           j     safe
```

Fig. 12. Example 1: Memory Allocation

certify the code in AGL because the specification of A and G requires global knowledge of memory. We certify the code in SAGL. Assertions are shown as annotations enclosed in "-{}". Recall that in SAGL the first assertion in the pair specifies shared resources and the second one specifies private resources. We treat all the resources as private, therefore the shared predicate is simply emp. The corresponding A and G are trivial. The whole verification is as simple as in CSL.

Our second example is adapted from Yu and Shao [19], which computes the greatest common divisor (GCD) of α and β, stored at locations m and n initially. The high-level pseudo code is shown in Fig. 13. Each thread's local variables are allocated in its private registers in the assembly code, which is similar to the high-level code and is shown in the technical report [4].

In this example, synchronization is achieved without using locks. To certify the code in CSL, we have to rewrite it by wrapping each memory-access command using lock and unlock commands and by introducing auxiliary variables. This time we use the "AGL part" of SAGL to certify the code: private predicates are simply emp. Assertions for the first thread are shown as annotations. In A_1 and G_1, we use primed values (*e.g.*, $[m]'$ and $[n]'$) to represent memory values in the resulting state of each action.

We give more examples in the technical report [4], which illustrate the support of dynamic redistribution of shared and private memory in SAGL.

7 Related Work and Conclusion

O'Hearn [11] proposed CSL for a high-level parallel language following Hoare [6]. Synchronization in the language is achieved by the conditional critical region (CCR) in the form of "with *r* when *b* do *c*". Semantics of CCRs is as follows: the statement *c* can be executed only if the resource *r* has not been acquired by others and the Boolean expression *b* is true; otherwise the thread will be blocked. We adapt CSL to an assembly language. The CCR can be implemented using our lock/unlock primitives. Each lock in our language corresponds to a resource name at the high-level. Atomic instructions in

$$\mathsf{a}_1 \stackrel{\text{def}}{=} \exists p,q.\ (\mathtt{m} \mapsto p) * (\mathtt{n} \mapsto q) \wedge gcd(p,q) = gcd(\alpha,\beta)$$
$$\mathsf{a}_2 \stackrel{\text{def}}{=} \exists p,q.\ (\mathtt{m} \mapsto p) * (\mathtt{n} \mapsto q) \wedge gcd(p,q) = gcd(\alpha,\beta) \wedge \mathtt{x} = p \wedge \mathtt{y} \geq q \wedge (p \geq q \rightarrow \mathtt{y} = q)$$
$$\mathsf{a}_3 \stackrel{\text{def}}{=} \exists p,q.\ (\mathtt{m} \mapsto p) * (\mathtt{n} \mapsto q) \wedge gcd(p,q) = gcd(\alpha,\beta) \wedge \mathtt{x} = p \wedge \mathtt{y} = q \wedge p > q$$
$$\mathsf{a}_4 \stackrel{\text{def}}{=} \exists p.\ (\mathtt{m} \mapsto p) * (\mathtt{n} \mapsto p) \wedge p = gcd(\alpha,\beta)$$
$$\mathsf{A}_1 \stackrel{\text{def}}{=} ([\mathtt{m}] = [\mathtt{m}]') \wedge ([\mathtt{n}] \geq [\mathtt{n}]') \wedge ([\mathtt{m}] \geq [\mathtt{n}] \rightarrow [\mathtt{n}] = [\mathtt{n}]') \wedge (gcd([\mathtt{m}],[\mathtt{n}]) = gcd([\mathtt{m}]',[\mathtt{n}]'))$$
$$\mathsf{G}_1 \stackrel{\text{def}}{=} ([\mathtt{n}] = [\mathtt{n}]') \wedge ([\mathtt{m}] \geq [\mathtt{m}]') \wedge ([\mathtt{n}] \geq [\mathtt{m}] \rightarrow [\mathtt{m}] = [\mathtt{m}]') \wedge (gcd([\mathtt{m}],[\mathtt{n}]) = gcd([\mathtt{m}]',[\mathtt{n}]'))$$

```
local int x, y;                          local int x, y;
while(true){                             while(true){
  -{(a1, emp)}
  x := [m];                                x := [n];
  y := [n];                                y := [m];
  -{(a2, emp)}
  if(x > y)                        ||      if(x > y)
    -{(a3, emp)}
    [m] := x-y;                              [n] := x-y;
  if(x == y) { break;}                     if(x == y) { break;}
}                                        }
-{(a4, emp)}
```

Fig. 13. Example 2: Parallel GCD

our assembly language are very similar to actions in Brookes Semantics [2], where semantic functions are defined for statements and expressions. These semantic functions can be viewed as a translation from the high-level language to a low-level language similar to ours. Recently, Reynolds [15] and Brookes [3] have studied grainless semantics for concurrency. Brookes also gives a grainless semantics to CSL [3].

The PROG rule of our CSL corresponds to O'Hearn's parallel composition rule [11]. The number of threads in our machine is fixed, therefore the nested parallel composition statement supported by Brookes [2] is not supported in our language. We studied verification of assembly code with dynamic thread creation in an earlier paper [5].

CSL is still evolving. Bornat *et al.* [1] proposed a refinement of CSL with fine-grained resource accounting. Parkinson *et al.* [13] applied CSL to verify a non-blocking implementation of stacks. As in the original CSL, these works also assume language constructs for synchronizations. We suspect that there exist reductions from these variations to SAGL-like logics. We leave this as our future work.

Concurrently with our work on SAGL, Vafeiadis and Parkinson [17] proposed another approach to combining rely/guarantee and separation logic, which we refer to here as RGSep. Both RGSep and SAGL partition memory into shared and private parts. However, shared memory cannot be accessed directly in RGSep. It has to be converted into private first to be accessed. Conversions can only occur at boundaries of critical regions, which is a built-in language construct required by RGSep to achieve atomicity. RGSep, in principle, does not assume smallest granularity of transitions. In SAGL, shared memory can be accessed directly, or be converted into private first and then accessed. Conversions can be made dynamically at any program point, instead of being coupled with critical regions. However, like A-G reasoning, SAGL assumes small-

est granularity. We suspect that RGSep can be compiled into a specialized version of SAGL, following the way we translate CSL. On the other hand, if our instructions are wrapped using critical regions, SAGL might be derived from RGSep too.

We also use SAGL as the basis to formalize the relationship between CSL and A-G reasoning. We encode the CSL invariant as an assumption and guarantee in SAGL, and prove that CSL rules are derivable from corresponding SAGL rules with the specific assumption and guarantee. Soundness of SAGL is proved following the syntactic approach to type soundness. Our work has been formalized in Coq [4].

References

[1] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Proc. 32nd ACM Symp. on Principles of Prog. Lang.*, pages 259–270, 2005.

[2] S. Brookes. A semantics for concurrent separation logic. In *Proc. 15th International Conference on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 16–34, 2004.

[3] S. Brookes. A grainless semantics for parallel programs with shared mutable data. In *Proc. MFPS XXI*, volume 155 of *Electr. Notes Theor. Comput. Sci.*, pages 277–307, 2006.

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

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

[6] 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.

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

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

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

[10] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science (to appear)*. Journal version of [11].

[11] P. W. O'Hearn. Resources, concurrency and local reasoning. In *Proc. 15th Int'l Conf. on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 49–67, 2004.

[12] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[13] M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. In *Proc. 34th ACM Symp. on Principles of Prog. Lang.*, page to appear. ACM Press, Jan. 2007.

[14] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS'02*, pages 55–74, July 2002.

[15] J. C. Reynolds. Toward a grainless semantics for shared-variable concurrency. In *Proc. FSTTCS'04*, volume 3328 of *LNCS*, pages 35–48, 2004.

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

[17] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. Available at `http://www.cl.cam.ac.uk/~mjp41/RGSep.pdf`, 2007.

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

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