Noname manuscript No.
(will be inserted by the editor)

GPS+ : Reasoning about Fences and Relaxed Atomics

Mengda He[1] · **Viktor Vafeiadis**[2] · **Shengchao Qin**[1] · **João F. Ferreira**[1,3]

the date of receipt and acceptance should be inserted later

Abstract In order to support efficient compilation to modern architectures, mainstream programming languages, such as C/C++ and Java, have adopted *weak* (or *relaxed*) memory models. According to these weak memory models, multithreaded programs are allowed to exhibit behaviours that would have been inconsistent under the traditional strong (i.e., sequentially consistent) memory model. This makes the task of reasoning about concurrent programs even more challenging. The GPS framework, developed by Turon et al. [23], has made a step forward towards tackling this challenge for the release-acquire fragment of the C11 memory model. By integrating ghost states, per-location protocols and separation logic, GPS can successfully verify programs with release-acquire atomics.

In this paper, we introduced GPS+ to support a larger class of C11 programs, that is, programs with release-acquire atomics, relaxed atomics and release-acquire fences. Key elements of our proposed logic include two new types of assertions, a more expressive resource model and a set of new verification rules.

1 Introduction

In concurrent programming, memory consistency models define how different threads may interact with each other using shared memory. Most work on concurrent program verification assumes the *sequentially consistency* (SC) memory model [14]. Under SC, there is a single global memory and threads take turns to access it. Within each thread instructions are executed in order, and each update to memory becomes visible to all threads at the same time and as soon as it occurs.

Although the SC model is intuitive and simplifies reasoning about concurrent programs, for efficiency reasons current hardware does not follow the SC model. In fact,

[1]School of Computing, Teesside University
[2]Max Planck Institute for Software Systems (MPI-SWS)
[3]HASLab / INESC TEC, Universidade do Minho, Portugal
E-mail: {m.he, s.qin,jff}@tees.ac.uk, viktor@mpi-sws.org

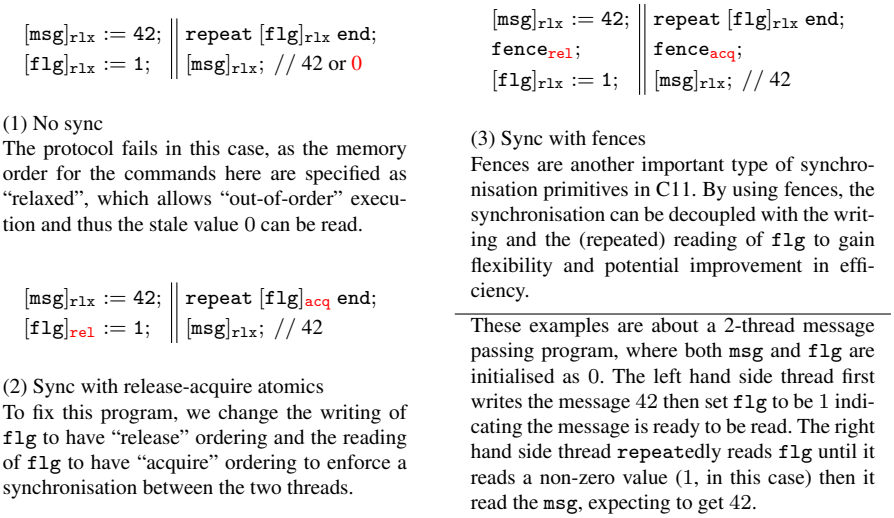


Fig. 1: Message passing examples

implementing SC over current hardware is expensive because costly synchronisation instructions (e.g., hardware fences) are required to keep memory operations properly synchronised, thereby also preventing many compiler and runtime optimisations.

As a result, modern hardware architectures and programming languages have adopted what is known as *relaxed* (or *weak*) *memory models*. In these models, different threads may observe that shared memory operations happen in different orders. For instance, x86 follows the *total-store-order (TSO)* model, in which all the stores are totally ordered but threads may observe their own stores early. ARM and PowerPC architectures have even weaker memory models. Similarly, to allow programmers to write more efficient concurrent code, programming languages like C/C++ and Java follow a weak memory model [1, 17].

In this paper, we focus on the C/C++ memory model (henceforth, C11). As a weak memory model, it allows different threads to have different observations of the memory, unless some synchronisations are enforced between them. In C11, synchronisations are formed by using release and acquire operations. Intuitively, a release operation shares its knowledge about the lastest memory updates, which is picked up by the acquire operations from other threads. We demonstrate this idea by using the examples shown in Fig. 1[1].

In order to be able to reason about concurrent programs, there is therefore a demand for *relaxed programming logics*—that is, logics for reasoning about concurrent programs under weak memory models. For the C11 memory model, there are notable program logics such as *Relaxed Separation Logic* (RSL) [24] and *GPS* [23], both of

[1] Here we explain these examples in an intuitive manner, leaving the formal introductions of syntax and semantics to the next section.

$$
\begin{array}{lll}
\mathit{Val} & v & ::= x \mid V \quad \text{where } V \in \mathbb{N} \\
\mathit{Exp} & e & ::= v \mid v + v \mid v == v \mid v \,\texttt{mod}\, v \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{fork}\ e \\
 & & \quad \mid \texttt{repeat}\ e\ \texttt{end} \mid \texttt{alloc}(n) \mid [v]_O \mid [v]_O := v \mid \texttt{fence}_O \mid \texttt{CAS}(v, v, v) \mid \texttt{FAI}(v) \\
\mathit{MO} & O & ::= \texttt{rel} \mid \texttt{acq} \mid \texttt{rlx} \mid \texttt{na} \\
\mathit{EvalCtx} & K & ::= [\,] \mid \texttt{let}\ x = K\ \texttt{in}\ e
\end{array}
$$

Fig. 2: A language for C11 concurrency with relaxed atomics and fences

which are extensions of separation logic. RSL supports invariant-based reasoning and ownership transfer via C11 release/acquire accesses, whereas GPS supports more advanced protocol-style reasoning. The more advanced of these logics, GPS, has been successfully applied to verify an implementation of Linux's RCU synchronisation primitive [21].

Neither of these logics, however, supports the full range of features of the C11 memory model. In this paper, we concentrate on two key features that are not supported (or only rudimentally supported): memory *fences* and *relaxed atomic* accesses. Our work extends GPS to support these features, thereby making the extended GPS+ applicable to a much bigger class of C11 programs. There has been a very recent closely related line of work on extending RSL with support for these features [8], but given that GPS is substantially more advanced than RSL, the soundness argument for our extension is substantially more complex than that of [8].

To support relaxed atomics and fences, we introduce two new types of assertions, namely *shareable* assertions and *waiting-to-be-acquired* assertions. We design a set of new verification rules that can verify programs with release/acquire atomics, relaxed atomics and release/acquire fences. We have formulated the soundness proof of the proposed verification logic.

Our work is based on the C11 memory model [1], which we describe in §2. We then briefly introduce GPS in §3 and present our new program logic GPS+ in §4. The new rules are put into action in §5 with an illustrative example. §6 presents the resource model of our logic, while §7 formulates the soundness proof of our proposed program logic. We conclude in §8 with a discussion of related and future work.

2 The Language and the Memory Model

We first present the syntax and semantics for a language capturing the essential C11 features, an extension of the core language used in GPS [23]; we then introduce the (simplified) C11 memory model on which our work is based.

2.1 The Language

Our core language (Fig. 2) is an expression-oriented language with pointer arithmetic, `let`-bindings (which form the only evaluation context K), conditional statements, thread forking, `repeat` e commands (which repeatedly execute the body e until a non-zero value is returned), memory allocation, load, store and fence operations

$$
\begin{array}{lll}
\textit{Action} & \alpha ::= & \mathbb{S} \mid \mathbb{A}(\ell..\ell') \mid \mathbb{W}(\ell, V, O) \mid \mathbb{R}(\ell, V, O) \mid \mathbb{U}(\ell, V, V) \mid \mathbb{F}(O) \\
\textit{ActName} & a & \text{(from an infinite set)} \\
\textit{ActMap} & A & \in \textit{ActName} \xrightarrow{fin} \textit{Action} \\
\textit{Graph} & G & = (A, \mathsf{sb}, \mathsf{mo}, \mathsf{rf}) \text{ where } \mathsf{sb}, \mathsf{mo}, \mathsf{rf} \subseteq \mathsf{dom}(A) \times \mathsf{dom}(A) \\
\textit{ThreadMap} & T & \in \mathbb{N} \xrightarrow{fin} (\textit{ActName} \times \textit{Exp})
\end{array}
$$

Fig. 3: Syntax of event graph

annotated with a specific *memory order* (mo), and the atomic operations compare-and-swap and fetch-and-increment.

The memory order annotation can be `rel` (for release-atomic stores), `acq` (for acquire-atomic loads), `rlx` (for relaxed atomic accesses), and `na` (for non-atomic accesses). Fence commands can be annotated with `rel` or `acq`. For the atomic compare-and-swap and fence-and-increment commands, we assume they have both `rel` and `acq` effects in case the operation succeeds, and only `acq` in case the update does not take place (in the case of CAS).

2.2 The Graph Semantics

The C11 memory model allows different threads to have different observations of the memory. Therefore, its semantics is not expressed in terms of changing a single shared memory, but rather keeps track of the entire history of an execution. In the formal C11 model by Batty et al. [2], an execution is represented as a labelled event graph and there is a set of axioms judging whether the execution is consistent according to the memory model (e.g. whether an access to a certain location leads to a data-race, or if it is possible for a read action to return a certain value).

Fig. 3 gives the definition of an event graph, which is formed by an action map and three relations. The *sequenced-before* relation sb records the order of events as specified in the program. As in GPS and RSL, we make this relation *not* transitive. Thus it relates each node only to its immediate successor in program order. The *modification-order* mo is a strict, total order on all writing actions to the same location. The *reads-from* relation rf relates a writing action to the reading actions that read from it. Since it is functional in its codomain, we often write $\mathsf{rf}(r)$ for the write action w such that $\mathsf{rf}(w, r)$.

Following GPS, event graphs are generated from programs by a two-layered operational semantics, whose rules are shown in Fig. 4 and Fig. 5, where **C** is the word size. In the event layer, actions are generated from program expressions $e \xrightarrow{\alpha} e'$. Note that a load operation generates a read action $\mathbb{R}$ with an arbitrary value. The actual value read is constrained by the C11 memory model in the second layer of semantics. Note also that $\mathbb{S}$ stands for a skip action, $\mathbb{A}$ for a memory allocation, $\mathbb{W}$ for a write, $\mathbb{U}$ for an atomic update, and $\mathbb{F}$ for a fence action.

In the second layer of semantics, instead of transforming expressions, a machine step changes *machine configurations* $\langle T; G\rangle$. Here T is the pool of threads maintaining the identity of the last event produced by each thread and their corresponding continuation expressions, and G is the event graph built up so far. In the graph G, all the

$$
\begin{array}{lll}
\texttt{let}\ x = V\ \texttt{in}\ e & \xrightarrow{\mathbb{S}} & e[V/x] \\
\texttt{repeat}\ e\ \texttt{end} & \xrightarrow{\mathbb{S}} & \texttt{let}\ x = e\ \texttt{in}\ \texttt{if}\ x\ \texttt{then}\ x\ \texttt{else}\ \texttt{repeat}\ e\ \texttt{end} \\
\texttt{alloc}(n) & \xrightarrow{\mathbb{A}(\ell..\ell+n-1)} & \ell \\
[\ell]_O & \xrightarrow{\mathbb{R}(\ell,V,O)} & V \\
[\ell]_O := V & \xrightarrow{\mathbb{W}(\ell,V,O)} & 0 \\
\texttt{CAS}(\ell, V_o, V_n) & \xrightarrow{\mathbb{U}(\ell,V_o,V_n)} & 1 \\
\texttt{CAS}(\ell, V_o, V_n) & \xrightarrow{\mathbb{R}(\ell,V',\texttt{rlx})} & 0 \qquad V' \neq V_o \\
\texttt{FAI}(\ell) & \xrightarrow{\mathbb{U}(\ell,V,V')} & V \qquad V' = (V+1) \bmod \mathbf{C} \\
\texttt{fence}_O & \xrightarrow{\mathbb{F}(O)} & 0 \\
K[e] & \xrightarrow{\alpha} & K[e'] \quad e \xrightarrow{\alpha} e'
\end{array}
$$

Fig. 4: Some event-step semantic rules: $e \xrightarrow{\alpha} e'$

$$
\frac{\begin{array}{c} e \xrightarrow{\alpha} e' \qquad \texttt{consistentC11}(G') \\ G'.A = G.A \uplus [a' \mapsto \alpha] \qquad G'.\mathsf{sb} = G.\mathsf{sb} \uplus (a, a') \\ G.\mathsf{mo} \subseteq G'.\mathsf{mo} \qquad G.\mathsf{rf} \subseteq G'.\mathsf{rf} \subseteq G.\mathsf{rf} \uplus (b, a') \end{array}}{\langle T \uplus [i \mapsto (a, e)]; G\rangle \longrightarrow \langle T \uplus [i \mapsto (a', e')]; G'\rangle}
$$

$$
\langle T \uplus [i \mapsto (a, K[\texttt{fork}\ (e)])]; G\rangle \longrightarrow \langle T \uplus [i \mapsto (a, K[0]) \uplus [j \mapsto (a, e)]; G\rangle
$$

Fig. 5: Machine step semantics: $\langle T; G\rangle \longrightarrow \langle T'; G'\rangle$

events that have taken place are recorded in the action map A and are connected with three kinds of directed edges, namely sb, mo and rf.

From a machine configuration $\langle T; G\rangle$, a move from an arbitrary thread can transfer into a new machine configuration $\langle T'; G'\rangle$ if the newly constructed graph G' is legal under C11 memory model: $\mathsf{consistentC11}(G')$.

2.3 The Memory Model

2.3.1 Happens-Before Relation

We have so far introduced sb, mo and rf. Now we describe the essential part of the memory model: synchronisations. Different from GPS and RSL, now fences can also form synchronisations. Our memory model is still simplified when compared with the standard [1] (for example, the subtle *release-sequence* is omitted).

We first introduce a derived relation *synchronised-with* ($\mathsf{sw} \subseteq \mathsf{dom}(A) \times \mathsf{dom}(A)$).As illustrated in Fig. 6, a pair of release write and acquire read can synchronise. Relaxed atomics can also synchronise with the help of corresponding fences.

The idea of synchronisation in C11 is that when an event c is synchronised with another event b, i.e. $(b, c) \in \mathsf{sw}$, then b's observation about its preceding memory updates becomes visible to c (and its succeeding events) as well. Based on this, the heart of the C11 memory model, *happens-before* relation, can be defined as: $\mathsf{hb} \triangleq (\mathsf{sb} \cup \mathsf{sw})^+$.

For instance, Fig. 7 represents an execution of the program shown in the case (2) of Fig. 1. When the acquire load c reads from (rf) the release store b, a synchronised-

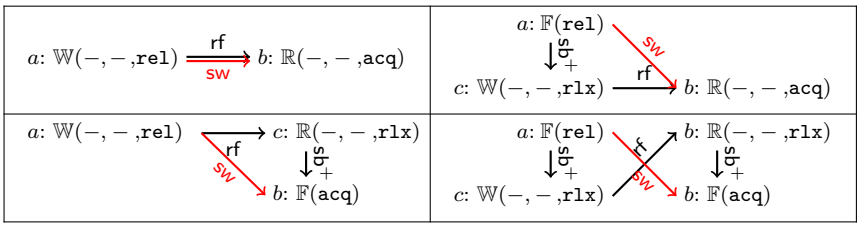


Fig. 6: Four ways to form synchronization

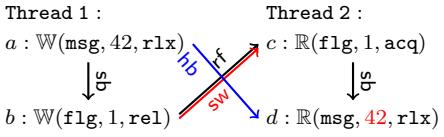


Fig. 7: Message passing using release write and acquire read

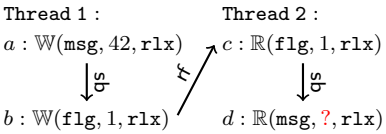


Fig. 8: Failed message passing

with (sw) relation is established between them. Consequently, information observed by the source store b is eligible to be shared with the reader c. In particular, this ensures d is aware that a has happened, thus it will not read the stale value 0.

On the other hand if either one or both of actions on `flg` are relaxed as shown in case (1) of Fig. 1 and Fig. 8, such sw relation fails to be established, which means the out-of-order executions allowed by the C11 standard may cause d to read the value 0 as well.

2.3.2 Data-Race and Memory Error

C11 provides various levels of memory consistency orders, from the most strict *sequentially-consistent* `sc` to the most relaxed *non-atomic* `na` (which does not even ensure atomicity), as a handy feature for users to flexibly balance the efficiency and safety of their programs. However, one must remember that when two events concurrently access a non-atomic location and at least one of them is a write event, it will lead to a *data race*. The C11 standard declares that if an execution is data-race free, the non-atomic actions will perform as they are sequentially-consistent; otherwise, the result of execution is undefined. Another situation that will lead to an undefined result is a *memory error*, which happens when an event accesses a location that has not been allocated.

2.3.3 Axioms

Following Batty et al. [2], the C11 memory model is formulated as a set of axioms (over an event graph G) in Fig. 9, denoted as $\mathsf{consistentC11}(G)$. In the machine step layer of semantics, the execution of our program is restricted by the C11 memory model via the checking with $\mathsf{consistentC11}(G)$, e.g. a load can not read from a write

$$
\begin{aligned}
&\mathsf{consistentC11}((A, \mathsf{sb}, \mathsf{mo}, \mathsf{rf})) \triangleq \\
&\quad \forall a, b.\ \mathsf{mo}(a,b) \Rightarrow \exists \ell.\ \mathsf{writes}(a,\ell,-), \mathsf{writes}(b,\ell,-) && \text{(ConsistentMO1)}\\
&\wedge\ \forall \ell.\ \mathsf{strictTotalOrder}(\{a \mid \mathsf{writes}(a,\ell,-)\}, \mathsf{mo}) && \text{(ConsistentMO2)}\\
&\wedge\ \forall b.\ \mathsf{rf}(b) \neq \bot \Leftrightarrow \exists \ell, a.\ \mathsf{writes}(a,\ell,-) \wedge \mathsf{reads}(b,\ell,-) \wedge \mathsf{hb}(a,b) && \text{(ConsistentRF1)}\\
&\wedge\ \forall a, b.\ \mathsf{rf}(b) = a \Rightarrow \exists \ell, V.\ \mathsf{writes}(a,\ell,V) \wedge \mathsf{reads}(b,\ell,V) \wedge \neg\mathsf{hb}(b,a) && \text{(ConsistentRF2)}\\
&\wedge\ \forall a, b.\ \mathsf{rf}(b) = a \wedge (\mathsf{isNonatomic}(a) \vee \mathsf{isNonatomic}(b)) \Rightarrow \mathsf{hb}(a,b) && \text{(ConsistentRFNA)}\\
&\wedge\ \forall a, b.\ \mathsf{hb}(a,b) \Rightarrow a \neq b \wedge \\
&\quad \neg\mathsf{mo}(\mathsf{rf}(b), \mathsf{rf}(a)) \wedge \neg\mathsf{mo}(\mathsf{rf}(b), a) \wedge \neg\mathsf{mo}(b, \mathsf{rf}(a)) \wedge \neg\mathsf{mo}(b,a) && \text{(Conherence)}\\
&\wedge\ \forall a, c.\ \mathsf{isUpd}(c) \wedge \mathsf{rf}(c) = a \Rightarrow \mathsf{mo}(a,c) \wedge\ \nexists b.\ \mathsf{mo}(a,b) \wedge \mathsf{mo}(b,c) && \text{(AtomicCAS)}\\
&\wedge\ \forall a \neq b, \vec{\ell}, \vec{\ell}'.\ A(a) = \mathbb{A}(\vec{\ell}) \wedge A(b) = \mathbb{A}(\vec{\ell}') \Rightarrow \vec{\ell} \cap \vec{\ell}' = \varnothing && \text{(ConsistentAlloc)}\\
&\wedge\ \mathsf{acyclic}(\mathsf{hb} \cup \mathsf{rf}) && \text{(Acyclic)}
\end{aligned}
$$

where $\mathsf{strictTotalOrder}(S, R) \triangleq (\nexists a.\ R(a,a))$
$\wedge\ (\forall a, b, c.\ R(a,b) \wedge R(b,c) \Rightarrow R(a,c)) \wedge (\forall a, b \in S.\ a \neq b \Rightarrow R(a,b) \vee R(b,a))$
$\mathsf{acyclic}(R) \triangleq \nexists x.\ R^+(x,x)$
$\mathsf{reads}(a,\ell,V) \triangleq A(a) \in \mathbb{R}(\ell,V,-), \mathbb{U}(\ell,V,-)$
$\mathsf{writes}(a,\ell,V) \triangleq A(a) \in \mathbb{W}(\ell,V,-), \mathbb{U}(\ell,-,V)$

Fig. 9: The C11 axioms

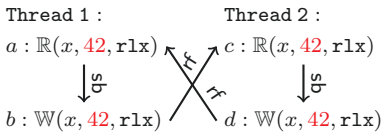
that "happens-after" (ConsistentRF2) and no location should be allocated more than once (ConsistentAlloc).

2.3.4 Thin-Air Read and the Strengthening of the Memory Model

Our core language includes relaxed atomic operations. However in the C11 memory model, relaxed atomics are known to have the *thin-air-read* issue [2], which refers to the problem that a program will allow a relaxed atomic read to return any value out of the thin air, without breaking the very few restrictions applied to relaxed atomics (Fig. 10). This problem makes it impossible to rigorously reason about a program with relaxed atomics. To rule out thin-air reads, we follow the same approach as RSL [24], i.e. we add an extra axiom, Acyclic, to the consistency check (Fig. 9).

$$
\begin{array}{l||l}
\texttt{let}\ r_1 = [x]_{\texttt{rlx}}; & \texttt{let}\ r_2 = [x]_{\texttt{rlx}}; \\
[x]_{\texttt{rlx}} := r_1; & [x]_{\texttt{rlx}} := r_2;
\end{array}
$$

(a) The program, where r_1, r_2 are local variables and x is shared location with initial value 0



(b) An arbitrary value 42 may pop up out of "thin-air"

Fig. 10: A program and its "thin-air-read" execution

3 The GPS Framework

Our proposed reasoning mechanism is built on top of the GPS framework [23,21], which combines three concepts advocated by state-of-the-art concurrent program logics (e.g. [25,9,7,3,15,6,22,18,20,4,13]), namely ghost states, protocols and separation logic, and adapts them in a novel way to support modular weak memory reasoning.

Compared with separation logic, GPS introduces *state assertions*, $\boxed{\ell : s \mid \tau}$, to keep track of the modifications to atomic locations; *knowledge*, $\Box P$, for the duplicable assertions, e.g. pure assertions, state assertions for atomics, etc.; *escrow* to wrap up ownership dependent assertions (e.g. assertions about non-atomic locations $x \hookrightarrow v$) before they can be spread to other threads; *ghost assertions*, $\boxed{\gamma : t \mid \mu}$, and *ghost move*, $\Rrightarrow$, for auxiliary states. To support these advanced features, *resources* are used to logically represent computation states. A resource $r \in \mathit{Resource}$ is a triple (Π, g, Σ) where the *physical location map* Π maps each location to either a value (for non-atomics) or a protocol and state (for atomics), the *ghost identity map* g keeps the ghost values, and the *known escrow set* Σ contains all escrows available. Resources form a *partially commutative monoid* (PCM) with composition $\oplus$. Semantics of assertions is provided by this resource model. For instance, the separation assertion $P_1 * P_2$ is interpreted as the current state r can be split into two *compatible* parts, $r = r_1 \oplus r_2$, where r_1 satisfies P_1 and r_2 satisfies P_2. GPS uses triple in the form of $\{P\}\ e\ \{x.Q\}$, where x is a placeholder for the return value in Q and can be omitted if Q does not describe the return value. It states if e terminates in a value v, the resources will satisfy $Q[v/x]$.

We shall first give a brief introduction about GPS, focusing on atomic writes/reads and escrows, which are essential for synchronisations.

3.1 Protocols for Atomic Locations

Following the C11 standard, atomic locations in GPS are meant to be read and written concurrently. Therefore it is difficult to make any stable assertions about the precise contents of an atomic location. GPS advocates per-location protocols to describe how the contents of each atomic location can evolve over time. A *state assertion* $\boxed{\ell : s \mid \tau}$ indicates that an atomic location l is governed by the protocol τ, and is at least at state s. All possible state transition relations have to be defined in τ as a partial order $\sqsubseteq_\tau$; and in τ, *state interpretation* $\tau(s, z)$ for each state s and its corresponding value z also has to be specified.

The state assertions about atomic locations belong to *knowledge* in GPS, which refers to assertions that do not depend on ownership. Intuitively, knowledge represents the statement of facts, and thus can be duplicated and spread to different threads. From the resource point of view, a knowledge refers to a piece of resource r that is compatible with itself, that is, $r \oplus r$ is defined. State assertions are ownership independent because according to the C11 standard, atomic locations are meant to be accessed concurrently (without `hb` ordering) in different threads. Correspondingly, GPS assertions about their states can be present in different threads at the same time. Conversely, the assertions about non-atomic locations (i.e. $x \hookrightarrow v$) are not knowledge

and must be owned by one thread at a time as concurrent access to them may raise data races. Knowledge is indicated by a modality $\Box$, and GPS has useful rules to reason about knowledge:

$$\boxed{\ell : s \mid \tau} \Rightarrow \Box \boxed{\ell : s \mid \tau},\ \Box P \Rightarrow P \text{ and } \Box P \Leftrightarrow \Box P * \Box P$$

The first rule says that a state assertion can be transformed into its knowledge form. The second says knowledge can always be turned back into its normal assertion. And the third shows that knowledge can be duplicated and thus be shared.

A state interpretation $\tau(s, z)$ for a protocol τ governing a location ℓ is an assertion specifying what must be true for a thread to be permitted to write z to ℓ and thus change it to state s. A read action which reads from this write may retrieve this assertion. This approach elegantly captures the idea of synchronisations in the C11 standard. Intuitively, the write action happens before that read (as a synchronised-with relation is formed between them), so it signifies that the effect of any preceding actions (those happened-before the write) can be transmitted to the reading thread.

The rule for atomic (i.e. *acquire*) read in GPS is given as:

$$\frac{[\textsc{gps}\text{-}\textsc{atomic}\text{-}\textsc{load}] \quad \forall s' \sqsupseteq_\tau s.\, \forall z.\, \tau(s', z) * P \Rightarrow \Box Q}{\left\{\boxed{\ell : s \mid \tau} * P\right\}\ [\ell]_{\mathtt{acq}}\ \left\{z.\, \exists s'.\, \boxed{\ell : s' \mid \tau} * P * \Box Q\right\}}$$

The possible writes that an atomic read can observe are quantified in the premise. Note that only assertions in knowledge form ($\Box Q$) can be gained, as it is possible for multiple threads to all read the location at the same state and thus gain the same assertion. Therefore if the assertion is not an ownership independent knowledge, data races may occur. The inclusion of the assertion P enables *rely-guarantee* reasoning through protocols [23].

The atomic (i.e. *release*) write rule in GPS is defined as:

$$\frac{[\textsc{gps}\text{-}\textsc{atomic}\text{-}\textsc{store}] \quad P \Rrightarrow \tau(s'', v) * Q \qquad \forall s' \sqsupseteq_\tau s.\, \tau(s', -) * P \Rightarrow s'' \sqsupseteq_\tau s'}{\left\{\boxed{\ell : s \mid \tau} * P\right\}\ [\ell]_{\mathtt{rel}} := v\ \left\{\boxed{\ell : s'' \mid \tau} * Q\right\}}$$

Note that from the precondition we only know the lower bound state for ℓ is state s (i.e. the location ℓ is at least at state s before the write takes place). Without knowing which exact state ℓ might have possibly been moved to by environment actions prior to this write, here the write moves it to state s'' that is reachable from any state s' such that $s' \sqsupseteq_\tau s$. In the first premise, P is transformed to the state interpretation $\tau(s'', v)$ with some frame Q via a *ghost move* $\Rrightarrow$. Ghost moves are another important concept in GPS: they represent moves that only change logical states without affecting the actual machine states. Ghost moves can take place any time that suits the logic user's needs. They can do useful things like creating ghost assertions, packing and unpacking escrows, which we are going to discuss next.

3.2 Escrows for Non-Atomic Locations

According to the rule [GPS–ATOMIC–LOAD], only knowledge can be transmitted in synchronisations. However, very often we need to transfer the ownership of non-

atomic locations. To do this, GPS allows them to be wrapped up into knowledge form and be retrieved at the right time, via the use of *escrows*.

An escrow of the form $\sigma : P \rightsquigarrow Q$ can be considered as a safe-box protecting Q, and the key to open it is P (which is not duplicable). Ghost moves are used to pack and unpack escrows:

$$\frac{[\textsc{gps-escrow-pack}]}{\sigma : P \rightsquigarrow Q \quad Q \Rrightarrow [\sigma]} \qquad \frac{[\textsc{gps-escrow-unpack}]}{\sigma : P \rightsquigarrow Q \quad P \wedge [\sigma] \Rrightarrow Q}$$

A packed escrow $[\sigma]$ is an ownership-independent assertion and can also be used in its knowledge form: $[\sigma] \Leftrightarrow \Box[\sigma]$.

The "key" P is consumed once it has been used to unpack an escrow. Therefore instead of using physical resources, ghost assertions are introduced to describe the permissions to unpack an escrow. A ghost assertion $\boxed{\gamma : t\ \mu}$ says there is a ghost variable γ, whose value is ghost permission t drawn from some *partial commutative monoid* (PCM) μ. New ghost t can appear out of thin air, with a fresh identity: $\mathsf{true} \Rrightarrow \exists \gamma.\ \boxed{\gamma : t\ \mu}$.

A special kind of permission is *token* Tok. Tok has only two kinds of permissions: ξ is the unit and represents empty permission; and $\diamond$ represents for full permission. They are usually written as $\boxed{\gamma : \xi}$ and $\boxed{\gamma : \diamond}$ for short.

4 GPS+ : Reasoning about Relaxed Atomics and Fences

We now present our key proposal: a program logic that supports the reasoning of a bigger class of C11 programs (than GPS), including relaxed atomics and release-acquire fences.

4.1 Two New Types of Assertions

We would like to handle relaxed atomic operations in a similar way as release and acquire atomics are treated in GPS, since they are also applied on atomic locations. Moreover, we would like to ensure that the idea of per-location protocols works for all of them. However, as defined in §2.3 and illustrated in Fig. 7 and Fig. 8, one challenge is that relaxed atomics form synchronised-with relations *differently* from release-acquire atomics: a sw relation is automatically set up when an acquire load operation reads from a release store operation; but for relaxed atomics the C11 standard states that the sw relation can only be established with the help of fences[2]. Fig. 11 shows that fences are needed to restore the sw and thus the hb relations for the example in Fig. 8.

We interpret these restrictions as (i) a relaxed atomic store operation can only transmit the information that has been marked as shareable by a preceding release fence; and (ii) a relaxed load should not put the knowledge gained from its loading source to the current state, instead it should mark the knowledge as not yet available

[2] Or via release sequences, which we do not consider in this paper.

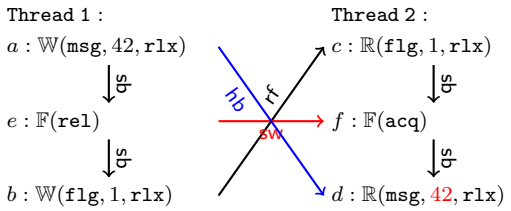


Fig. 11: Message passing using relaxed atomics with fences

and await a succeeding acquire fence to transform them to normal knowledge form. To cater for these new scenarios, we introduce two new types of assertions: *shareable assertions* $\langle P \rangle$, and *waiting-to-be-acquired assertions* $\boxtimes P$.

Intuitively $\langle P \rangle$ indicates that P is shareable. That is, it can be transmitted to others (even by a relaxed store operation). $\boxtimes P$ signifies that knowledge received by a relaxed load is not yet available according to the C11 standard. Reading, updating or re-transmitting $\boxtimes P$ is not permitted until an acquire fence transforms it into normal knowledge $\square P$. Intuitively, the two new types of assertions provided an additional dimension to evaluate assertions, based on if they (knowledge or not) can be used/-transmitted by the current thread.

The formal semantics for these new assertions and their properties will be presented later in §6. It is worth noting here that unlike $\square P \Rightarrow P$, the property $\boxtimes P \Rightarrow P$ does not hold, as according to the C11 standard, $\boxtimes$ can only be stripped off by using an acquire fence. Moreover, unlike the knowledge symbol $\square$ that can be nested, the nesting of shareable or waiting-to-be-acquired assertions is not allowed. As otherwise, if an assertion like $\boxtimes\langle P \rangle$ is permitted, after an acquire fence it immediately becomes a shareable assertion, which clearly violates the C11 standard.

It is also worth noting that, in order to prevent improper assertions (like $\boxtimes P$ or $\langle P \rangle$) from being included in state interpretations for atomic variables, we require that all state interpretations must be "normal" assertions, i.e. $\forall \tau, s, V.\ \mathsf{normal}(\tau(s, V))$, where $\mathsf{normal}(P) \triangleq P \Rightarrow \texttt{false} \vee \langle P \rangle \not\Rightarrow \texttt{false}$. A similar restriction is applied to the assertions used in escrows: for each escrow $\sigma : P \rightsquigarrow P'$, we require $\mathsf{normal}(P)$ and $\mathsf{normal}(P')$.

4.2 New Verification Rules

With the new forms of knowledge and assertions, we can now ensure that knowledge will be distributed in a controlled manner both from the starting point (a store operation) and at the finishing point (a load operation). We present a number of newly-designed verification rules in Fig. 12. The rules that are inherited from GPS without change and the rule for FAI are left for the technical report [10].

Being atomic store operations, both release and relaxed stores can transmit some extra information to their readers. But according to the standard and as pointed out in their instrumented semantics we discussed before, the scopes of information that are available for them to release are different. This difference is captured by our rules. Being a store using a weaker memory order, a relaxed store can only use the

$$\frac{P \Rrightarrow \tau(s'', v) * Q \qquad \forall s' \sqsupseteq_\tau s.\, \tau(s', -) * P \Rightarrow s'' \sqsupseteq_\tau s'}{\left\{\boxed{\ell : s \mid \tau} * P\right\}\, [\ell]_{\texttt{rel}} := v\, \left\{\boxed{\ell : s'' \mid \tau} * Q\right\}} \;[\textsc{release-store}]$$

$$\frac{P_2 \Rrightarrow \tau(s'', v) * Q \qquad \forall s' \sqsupseteq_\tau s.\, \tau(s', -) * P_1 * P_2 \Rightarrow s'' \sqsupseteq_\tau s'}{\left\{\boxed{\ell : s \mid \tau} * P_1 * \langle P_2 \rangle\right\}\, [\ell]_{\texttt{rlx}} := v\, \left\{\boxed{\ell : s'' \mid \tau} * P_1 * Q\right\}} \;[\textsc{relaxed-store}]$$

$$\frac{\langle P \rangle \not\Rightarrow \texttt{false}}{\{P\}\ \texttt{fence}_{\texttt{rel}}\ \{\langle P \rangle\}} \;[\textsc{release-fence}] \qquad \frac{\forall s' \sqsupseteq_\tau s.\, \forall z.\, \tau(s', z) * P \Rightarrow \Box Q}{\left\{\boxed{\ell : s \mid \tau} * P\right\}\ [\ell]_{\texttt{acq}}\ \left\{z.\, \exists s'.\, \boxed{\ell : s' \mid \tau} * P * \Box Q\right\}} \;[\textsc{acquire-load}]$$

$$\frac{\forall s' \sqsupseteq_\tau s.\, \forall z.\, \tau(s', z) * P \Rightarrow \Box Q}{\left\{\boxed{\ell : s \mid \tau} * P\right\}\ [\ell]_{\texttt{rlx}}\ \left\{z.\, \exists s'.\, \boxed{\ell : s' \mid \tau} * P * \boxtimes Q\right\}} \;[\textsc{relaxed-load}]$$

$$\{\boxtimes P\}\ \texttt{fence}_{\texttt{acq}}\ \{\Box P\} \;[\textsc{acquire-fence}]$$

$$\frac{\begin{array}{c}\forall s' \sqsupseteq_\tau s.\, \tau(s', v_o) * P_1 * P_2 \Rrightarrow \exists s'' \sqsupseteq_\tau s'.\, \tau(s'', v_n) * Q \\ \forall s'' \sqsupseteq_\tau s.\, \forall y \neq v_o.\, \tau(s'', y) * P_1 \Rightarrow \Box R\end{array}}{\left\{\boxed{l : s \mid \tau} * P_1 * \langle P_2 \rangle\right\}\, \texttt{CAS}(l, v_o, v_n)\, \begin{Bmatrix} z.\exists s''.\, \boxed{l : s'' \mid \tau} * ((z{=}1 * Q) \\ \vee (z{=}0 * P_1 * \langle P_2 \rangle * \Box R)) \end{Bmatrix}} \;[\textsc{cas}]$$

Fig. 12: New verification rules

assertion P_2 that is marked as shareable in its precondition to imply the interpretation of the state it is going to write, i.e. it can only transmit the things that are already marked as shareable. Meanwhile, a release store uses a general assertion P, which is not necessarily to be a shareable assertion, to ghostly imply the state interpretation it needs. Note that P can also contain shareable assertions, in which case the following [UNSHARE] ghost move becomes handy if the normal form of these assertions is needed to imply the state interpretation, $[\textsc{unshare}] : \langle P \rangle \Rrightarrow P$

This ghost move allows us to convert a shareable assertion back to its previous form (where resources were held in the local part instead of the shareable part). The assertion P_1 in the [RELAXED-STORE] rule is used to reduce the possible intermediate environment moves we need to consider.

A release fence marks resources that are ready to be shared. Our [RELEASE-FENCE] rule shows that an assertion P in its precondition is transformed into a shareable assertion after the fence (assuming it is possible to do so). The sanity check in the premise prevents `false` from being gained in the postcondition. Note that if the precondition P is already a shareable assertion or a waiting-to-be-acquired assertion (i.e. $\langle P \rangle \Rightarrow \texttt{false}$), the release-fence would act like the skip action, and the postcondition would remain as P (according to the frame rule in Separation Logic).

For atomic loads, the [ACQUIRE-LOAD] rule in GPS is compatible with our new setting. Note that the knowledge it retrieves from its load source is directly put in the postcondition. However as we have discussed, the knowledge gained by a relaxed

load should not be considered as immediately available to the current thread (for reading, updating or re-transmitting). Therefore, in our new [RELAXED–LOAD] rule, the knowledge $\square Q$ the load gains is marked as waiting-to-be-acquired knowledge $\boxtimes Q$ in its post condition. One can then use the [ACQUIRE–FENCE] rule to turn an acquirable knowledge into a normal one.

$\texttt{CAS}(\ell, v_o, v)$ (compare and swap) is an important synchronisation operation, which is widely used in various lock algorithms. It performs the following things in one atomic step: firstly it loads from ℓ, and compares the value it gets with the expected value v_o; if they are equal, it updates ℓ with a new value v and returns 1 indicating its success, otherwise returns 0. The CAS in our [CAS] rule is a release-acquire CAS, i.e. in the case of success (corresponding to the first premise) it behaves like a release store, and in the case of fail (corresponding to the second premise) it behaves like an acquire load that read some value other than v_o. Moreover, in the case of success, it can retrieve non-knowledge assertions from the interpretation of the state s'. As we require that all state interpretations must be normal assertions (or `false`), we do not need to be concerned that improper assertions, like shareable assertions that can be immediately re-transmitted by any following relaxed stores without a release fence, will be retrieved from $\tau(s', v_o)$ and left over in Q.

5 Illustrative Example

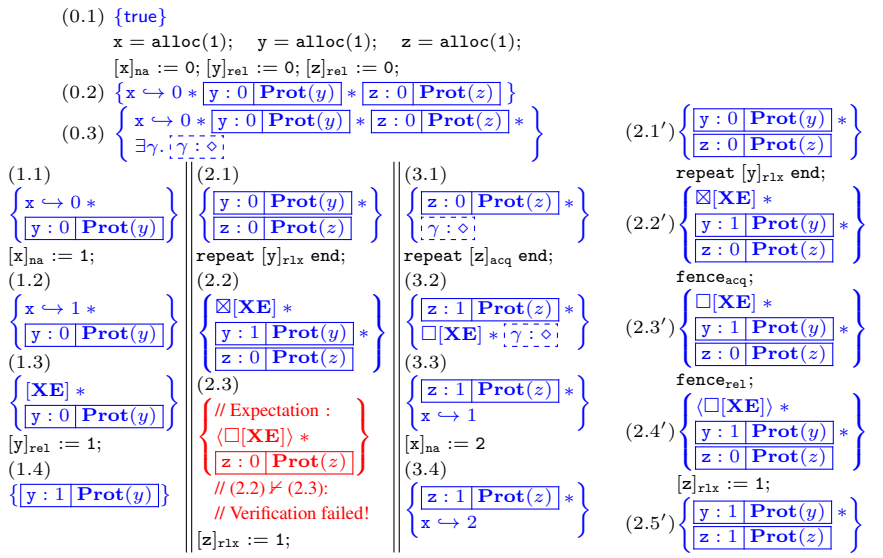
We illustrate our reasoning logic using the racy program shown in Fig 13. We first show how our logic can detect the data race and how it is unable to prove the program to be correct. We then show that after resolving the race by properly adding fences, our logic can prove it successfully.

Note that a message $\texttt{x} \hookrightarrow 1$ is created in thread 1, and is passed to thread 2 by the release store to `y`. Thread 2 performs a relaxed store to `z`, intending to retransmit this message to thread 3, where the ownership of `x` is demanded to perform the non-atomic write.

According to the C11 standard, this program contains a data race as it is not properly synchronised. Despite the fact that in thread 1 the store operation to `y` is release atomic, the load operation in thread 2 that reads from it is relaxed. Without a subsequent acquire fence, no synchronisation can be established between thread 1 and 2. Similarly, though the acquire load operation of `z` in thread 3 reads from the store operation in thread 2, the two threads are not synchronised as the store operation is relaxed and lacking a release fence before it. Therefore, the chain of happens be-

$$
\begin{array}{l}
\texttt{let x} = \texttt{alloc}(1)\ \texttt{in} \\
\texttt{let y} = \texttt{alloc}(1)\ \texttt{in} \\
\texttt{let z} = \texttt{alloc}(1)\ \texttt{in} \\
[\texttt{x}]_{\texttt{na}} := 0;\ [\texttt{y}]_{\texttt{rel}} := 0;\ [\texttt{z}]_{\texttt{rel}} := 0; \\
\begin{array}{l|l|l|l}
[\texttt{x}]_{\texttt{na}} := 1; & \texttt{repeat}\ [\texttt{y}]_{\texttt{rlx}}\ \texttt{end}; & \texttt{repeat}\ [\texttt{z}]_{\texttt{acq}}\ \texttt{end}; & \\
[\texttt{y}]_{\texttt{rel}} := 1; & [\texttt{z}]_{\texttt{rlx}} := 1; & [\texttt{x}]_{\texttt{na}} := 2 &
\end{array}
\end{array}
$$

Fig. 13: A program with a data race



(a) Failed verification of the racy program

(b) Verifying the fixed thread 2

Fig. 14: Verification of Relayed Message Passing

fore (hb) relation breaks between thread 1 and 3. Without having a happens before relation, the non-atomic writes to x in thread 1 and 3 produce a data race.

We show in Fig 14a that, with the help of the two new types of assertions, our logic can detect the failure of synchronisation, and will not prove the racy program to be correct. First, we define the escrow for x and protocols for y and z, where each of y and z has only two protocol states 0 and 1, and $0 \sqsubseteq_{\mathbf{Prot}(l)} 1$ for $l \in \{\mathtt{y}, \mathtt{z}\}$:

$$\mathbf{XE} : \boxed{\gamma : \diamond} \rightsquigarrow x \hookrightarrow 1, \qquad \mathbf{Prot}(\ell)(0, v) \triangleq v{=}0 \quad \mathbf{Prot}(\ell)(1, v) \triangleq v{=}1 \wedge \square[\mathbf{XE}] \quad \ell \in \{\mathtt{y}, \mathtt{z}\}$$

As shown in Fig. 14a, the verification could not be finished in thread 2. Even though in thread 1 the message about x is packed via ghost move from (1.2) to (1.3), and put into y's state interpretation as knowledge, the relaxed load operation of y in thread 2 can only extract the knowledge in a waiting-to-be-acquired form ⊠[XE] according to [RELAXED–LOAD]. Without subsequent acquire and release fences, this waiting-to-be-acquired knowledge is kept in this form and cannot be used to entail the required precondition for the next command [z]rlx := 1, in which the packed escrow is expected to be in the shareable form ⟨□[XE]⟩ according to the rule [RELAXED–STORE].

To resolve the data race in this program, as shown in Fig 14b, an acquire fence and a release fence are needed to be inserted between the relaxed load operation of y and the release operation to z in thread 2, which will change the waiting-to-be-acquired knowledge into normal knowledge and then shareable knowledge before the relaxed store operation to z transfers it to thread 3.

It is worth noting that our logic supports modular reasoning. The verification of thread 1 and 3 can be conducted separately despite the error in the original thread 2.

We have also applied our reasoning logic to a number of more challenging programs as documented in the appendix.

6 Resource Model

Under weak memory models, it is difficult to assume a sequentially-consistent global heap. Instead, resources are used in GPS to logically represent the computation states. In this section we shall first briefly introduce the GPS resource model and then present our new resource model which is built on the GPS one.

6.1 GPS Resources

Instead of assuming a sequentially-consistent global heap, GPS makes use of resources to logically represent computation states. A resource $r \in \mathit{Resource}$ is a triple (Π, g, Σ) where the *physical location map* Π maps each location to either a value (for non-atomics) or a protocol and state (for atomics), the *ghost identity map* g keeps the ghost values, and the *known escrow set* Σ contains all escrows available. Resources form a PCM with composition $\oplus$. Some useful definitions are:

$$\mathtt{emp} \triangleq ((\lambda n.\ \bot), (\lambda \mu.\ \lambda n.\ \epsilon_\mu), \varnothing) \qquad r \leq r' \triangleq \exists r''.\ r \oplus r'' = r' \qquad r \# r' \triangleq r \oplus r' \text{defined}$$

Each proposition P in GPS is interpreted as a set of resources, i.e. $[\![P]\!]^\rho \subseteq \mathit{Resource}$, where ρ is a term interpretation we assumed for state and PCM terms. Moreover, the interpretation satisfies the following property: $\forall r \in [\![P]\!]^\rho.\ \forall r' \# r.\ r \oplus r' \in [\![P]\!]^\rho$

GPS also introduces a rely-guarantee-styled instrumented semantics for all actions. Let us take the release store operation as an example. Given a resource r_{pre} that meets the pre-condition of the write, and assuming resource r is the actual resource used by the write (note r can be different from r_{pre} as the environment may also make changes prior to the write), the effect of this atomic write can be illustrated by its guarantee definition as shown below, where r_{sb} is the resource that will be passed down to its sb successor in the execution graph and r_{rf} is the resource to be transmitted to its reader:

$$\begin{array}{l}(r_{\mathsf{sb}}, r_{\mathsf{rf}}) \in \mathsf{guar}(r_{\mathsf{pre}}, r, \mathbb{W}(l, V, \mathtt{rel})) \text{ if} \\ \quad \exists \tau, s, S. r_{\mathsf{rf}} \in \mathsf{interp}(\tau)(s, V) \wedge r_{\mathsf{rf}} \oplus r_{\mathsf{sb}} = r[\ell := \mathtt{at}(\tau, S \cup \{s\})] \wedge r_{\mathsf{sb}}[\ell] = r_{\mathsf{rf}}[\ell] \\ \qquad \wedge (r[\ell] = \mathsf{uninit} \wedge S = \varnothing \vee r[\ell] = \mathtt{at}(\tau, S) \wedge \forall s_0 \in S.\ s_0 \sqsubseteq_\tau s) \\ \qquad \wedge \forall r_E.\ \left(\begin{array}{l} \exists \tau, s', V'.\ r_E \in \mathsf{interp}(\tau)(s', V') \wedge r_{\mathsf{pre}} \# r_E \\ \wedge\ r_{\mathsf{pre}}[\ell] \sqsubseteq_{\mathtt{at}} \mathcal{R}_E[\ell] \equiv_{\mathtt{at}} \mathtt{at}(\tau, S \cup \{s'\}) \end{array} \right) \Rightarrow r_E[\ell] \sqsubseteq_{\mathtt{at}} r_{\mathsf{rf}}[\ell]\end{array}$$

Note that $\mathsf{interp}(\tau)(s, V)$ denotes the semantics of the state interpretation under the new state s, namely $[\![\tau(s, V)]\!]^\rho$, which carries the information we intend to transmit through this atomic write. The notation $r[\ell]$ is short for $r.\Pi(\ell)$, which is the value of the physical location ℓ. For an atomic location, this is an atomic protocol value in the form of $\mathtt{at}(\tau, S)$, where τ is the protocol type governing that location and S is a trace

of states the location has gone through. Some relations between these protocol values are defined as:

$$\mathtt{at}(\tau, S) \sqsubseteq_\tau \mathtt{at}(\tau, S') \triangleq \forall s \in S.\, \exists s' \in S'.\, s \sqsubseteq_\tau s' \qquad \pi \equiv_{\mathtt{at}} \pi' \triangleq \pi \sqsubseteq_\tau \pi' \wedge \pi' \sqsubseteq_\tau \pi$$

The assertion-level ghost move is defined in terms of resource-level ghost moves:

$$\rho \models P \Rrightarrow Q \triangleq \forall r \in [\![P]\!]^\rho.\, r \Rrightarrow [\![Q]\!]^\rho$$

For instance, the escrow packing rule is validated by the following resource-level ghost move:

$$\frac{\mathsf{interp}(\sigma) = ([\![P]\!]^\rho, [\![P']\!]^\rho) \qquad r' \in [\![P']\!]^\rho}{(\Pi, g, \Sigma) \oplus r' \Rrightarrow \lfloor (\Pi, g, \Sigma \cup \{\sigma\}) \rfloor}$$

Note that the escrow's interpretation $\mathsf{interp}(\sigma) = ([\![P]\!]^\rho, [\![P']\!]^\rho)$. Note also that $\lfloor r \rfloor$ is defined as $\{r \oplus r' \mid r' \in \mathit{Resource}\}$.

6.2 The New Resource Model

To deal with the two new types of assertions, we extend the GPS resource model to a more expressive one by lifting resources to resource triples:

$$\mathit{ResTriple} \triangleq \{(r_1, r_2, r_3) \mid r_1, r_2, r_3 \in \mathit{Resource} \wedge r_1 \oplus r_2 \oplus r_3 \ \mathsf{defined}\}$$

For each resource triple $\mathcal{R} = (r_1, r_2, r_3)$ we use $\mathcal{R}[\mathsf{L}]$ to denote r_1,$\mathcal{R}[\mathsf{S}]$ for r_2, and $\mathcal{R}[\mathsf{A}]$ for r_3, representing resp. its *local*, *shareable*, and *waiting-to-be-acquired* component.

Like resources, *ResTriple* also forms a PCM. The composition operation $\oplus$ is defined point-wise; the compatibility can be defined as: $\mathcal{R}\#\mathcal{R}' \triangleq \mathcal{R} \oplus \mathcal{R}'$ defined. $\mathtt{EMP}$ is defined as a resource triple comprising only empty resources: $\mathtt{EMP} \triangleq (\mathtt{emp}, \mathtt{emp}, \mathtt{emp})$.

The semantics for propositions is lifted to the *ResTriple* model as well. The interpretation $[\![P]\!]^\rho$ of an assertion P is a set of resource triples satisfying the property:

$$\forall \mathcal{R} \in [\![P]\!]^\rho.\, \forall \mathcal{R}' \# \mathcal{R}.\, \mathcal{R} \oplus \mathcal{R}' \in [\![P]\!]^\rho$$

For any basic assertion P and resource triple $\mathcal{R}$, only the local part of $\mathcal{R}$ is needed when checking $\mathcal{R} \in [\![P]\!]^\rho$. For example, $\mathcal{R} \in [\![\boxed{\ell : s \mid \tau}]\!]^\rho \Leftrightarrow \exists S.\, \mathcal{R}[\mathsf{L}].\Pi(\ell) = \mathtt{at}(\tau, S) \wedge s \in S$

Composed assertions like separating conjunction are directly lifted up to use resource triples: $\mathcal{R} \in [\![P_1 * P_2]\!]^\rho \Leftrightarrow \exists \mathcal{R}_1, \mathcal{R}_2.\, \mathcal{R} = \mathcal{R}_1 \oplus \mathcal{R}_2 \wedge \mathcal{R}_1 \in [\![P_1]\!]^\rho \wedge \mathcal{R}_2 \in [\![P_2]\!]^\rho$.

The semantics for synchronisation related assertions, namely knowledge, shareable assertion and waiting-to-be-acquired assertions are defined as:

$$\begin{aligned}
&\mathcal{R} \in [\![\Box P]\!]^\rho \Leftrightarrow |(\mathcal{R}[\mathsf{L}], \mathtt{emp}, \mathtt{emp})| \in [\![P]\!]^\rho \\
&\mathcal{R} \in [\![\langle P \rangle]\!]^\rho \Leftrightarrow (\mathcal{R}[\mathsf{S}], \mathtt{emp}, \mathtt{emp}) \in [\![P]\!]^\rho \\
&\mathcal{R} \in [\![\boxtimes P]\!]^\rho \Leftrightarrow (\mathcal{R}[\mathsf{A}], \mathtt{emp}, \mathtt{emp}) \in [\![\Box P]\!]^\rho
\end{aligned}$$

Note the stripping $|\mathcal{R}|$ is a lifted version of the GPS stripping, i.e. $|(r_1, r_2, r_3)| \triangleq (|r_1|, |r_2|, |r_3|)$. [3]

[3] In GPS, $|r|$ represents the duplicable part of r: $r = r \oplus |r|$. For duplicable items in r, like atomic values and the known escrow set, stripping keeps them unchanged. That is, we have $|r|.\Sigma = r.\Sigma$, and if

Under the new resource model, the following properties hold. Note properties for knowledge that hold in GPS are all preserved in the new model but are omitted here.

$$\begin{array}{lll} \Box\langle P\rangle \;\Rightarrow\; \texttt{false if EMP}\notin[\![P]\!]^\rho & \boxtimes\langle P\rangle \;\Rightarrow\; \texttt{false if EMP}\notin[\![P]\!]^\rho & \langle P\rangle * \langle Q\rangle \Leftrightarrow \langle P * Q\rangle \\ \langle\boxtimes P\rangle \;\Rightarrow\; \texttt{false if EMP}\notin[\![P]\!]^\rho & \Box\boxtimes P \Rightarrow \texttt{false if EMP}\notin[\![P]\!]^\rho & \boxtimes P \Leftrightarrow \boxtimes P * \boxtimes P \\ \boxtimes\boxtimes P \Rightarrow \texttt{false if EMP}\notin[\![P]\!]^\rho & \langle\langle P\rangle\rangle \;\Rightarrow\; \texttt{false if EMP}\notin[\![P]\!]^\rho & \end{array}$$

6.2.1 Ghost Moves

As in GPS, assertion-level ghost moves are defined in terms of resource-level ghost moves: $\rho \models P \Rrightarrow Q \triangleq \forall \mathcal{R} \in [\![P]\!]^\rho.\, \mathcal{R} \Rrightarrow [\![Q]\!]^\rho$. The only difference is that resource triples are now used in the resource level. For instance, the resource level escrow packing ghost move is changed to:

$$\frac{\mathsf{interp}(\sigma) = ([\![P]\!]^\rho, [\![P']\!]^\rho) \quad \mathcal{R}' \in [\![P']\!]^\rho \quad \mathcal{R}[\mathsf{L}] = (\Pi, g, \Sigma)}{\mathcal{R} \oplus \mathcal{R}' \Rrightarrow \lfloor((\Pi, g, \Sigma \cup \{\sigma\}), \mathcal{R}[\mathsf{S}], \mathcal{R}[\mathsf{A}])\rfloor}$$

Based on this definition, we can obtain the same escrow packing rule as that in GPS.

In addition to all ghost moves inherited from GPS, we also propose a new one:

$$\frac{\mathcal{R}'[\mathsf{L}] = \mathcal{R}[\mathsf{L}] \oplus r \quad \mathcal{R}'[\mathsf{S}] \oplus r = \mathcal{R}[\mathsf{S}] \quad \mathcal{R}'[\mathsf{A}] = \mathcal{R}[\mathsf{A}]}{\mathcal{R} \Rrightarrow \lfloor\mathcal{R}'\rfloor}$$

This resource-level ghost move gives us the assertion-level ghost move rule [UNSHARE] (shown in §4).

6.2.2 Rely/Guarantee Definitions

Following the GPS approach, we define the instrumented semantics for all actions in the rely/guarantee style. But instead of manipulating resources, our actions work on resource triples, which is more expressive and allows us to describe the subtle difference among various kinds of actions. As an example, the guarantee definitions for release and relaxed writes are illustrated in Fig. 15.

Note that a release write can move a resource (r_2) from the triple's local part $\mathcal{R}[\mathsf{L}]$ to the shareable part $\mathcal{R}[\mathsf{S}]$ and transmit it to the readers, while the relaxed write can only transmit the resource already in the shareable component.

The full rely and guarantee conditions for GPS+'s extended set of actions in Fig. 16 and Fig. 17 with the following shorthand definitions:

$$\begin{array}{rl} \mathcal{R}_{\mathsf{rf}} \in \mathsf{envMove}(\mathcal{R}, l, V) \triangleq & \exists \tau, s.\, (\mathcal{R}_{\mathsf{rf}}[\mathsf{S}], \mathtt{emp}, \mathtt{emp}) \in \mathsf{interp}(\tau)(s, V) \\ & \wedge\, \mathcal{R}\#\mathcal{R}_{\mathsf{rf}} \wedge \mathcal{R}[\mathsf{L}][\ell] \sqsubseteq_\tau \mathcal{R}_{\mathsf{rf}}[\mathsf{S}][\ell] \equiv_{\mathtt{at}} \mathtt{at}(\tau, \{s\}) \end{array}$$

$\ell_{\mathtt{at}}$ is an atomic location in r we have $|r|.\Pi(\ell_{\mathtt{at}}) = r.\Pi(\ell_{\mathtt{at}})$. For non-duplicable items, like non-atomic values, stripping removes them. For example, if $\ell_{\mathtt{na}}$ is a non-atomic location in r we have $|r|.\Pi(\ell_{\mathtt{na}}) = \bot$. The value $\diamond$ of ghost type Tok is also not duplicable, and all ghost locations of type Tok will be set as empty after stripping: $|r|.g(\mathsf{Tok})(-) = \xi$.

$(\mathcal{R}_{\mathsf{sb}}, \mathcal{R}_{\mathsf{rf}}) \in \mathsf{guar}(\mathcal{R}_{pre}, \mathcal{R}, \mathbb{W}(\ell, V, \mathtt{rel}))$ if
$\exists \tau, s, S, \mathcal{R}', r_{\mathsf{rf}}.$
$\left(\begin{array}{l} \exists r_1, r_2.\ \mathcal{R}'[\mathsf{A}] = \mathcal{R}[\mathsf{A}] \\ \wedge\ \mathcal{R}[\mathsf{L}] = r_1 \oplus r_2 \wedge r_2 \leq r_{\mathsf{rf}} \\ \wedge \mathcal{R}'[\mathsf{L}] = r_1[\ell := \mathtt{at}(\tau, S \cup \{s\})] \\ \wedge\ \mathcal{R}'[\mathsf{S}] = \mathcal{R}[\mathsf{S}] \oplus r_2[\ell := \mathtt{at}(\tau, S \cup \{s\})] \end{array}\right)$
$\wedge\ (r_{\mathsf{rf}}, \mathtt{emp}, \mathtt{emp}) \in \mathsf{interp}(\tau)(s, V)$
$\wedge \mathcal{R}_{\mathsf{rf}} = (\mathtt{emp}, r_{\mathsf{rf}}, \mathtt{emp})$
$\wedge\ \mathcal{R}_{\mathsf{rf}} \oplus \mathcal{R}_{\mathsf{sb}} = \mathcal{R}'$
$\ldots$

(a) New guarantee condition for release write

$(\mathcal{R}_{\mathsf{sb}}, \mathcal{R}_{\mathsf{rf}}) \in \mathsf{guar}(\mathcal{R}_{pre}, \mathcal{R}, \mathbb{W}(\ell, V, \mathtt{rlx}))$ if
$\exists \tau, s, S, \mathcal{R}', r_{\mathsf{rf}}.$
$\left(\begin{array}{l} \mathcal{R}'[\mathsf{A}] = \mathcal{R}[\mathsf{A}] \\ \\ \wedge\ \mathcal{R}'[\mathsf{L}] = \mathcal{R}[\mathsf{L}][\ell := \mathtt{at}(\tau, S \cup \{s\})] \\ \wedge\ \mathcal{R}'[\mathsf{S}] = \mathcal{R}[\mathsf{S}][\ell := \mathtt{at}(\tau, S \cup \{s\})] \end{array}\right)$
$\wedge\ (r_{\mathsf{rf}}, \mathtt{emp}, \mathtt{emp}) \in \mathsf{interp}(\tau)(s, V)$
$\wedge \mathcal{R}_{\mathsf{rf}} = (\mathtt{emp}, r_{\mathsf{rf}}, \mathtt{emp})$
$\wedge\ \mathcal{R}_{\mathsf{rf}} \oplus \mathcal{R}_{\mathsf{sb}} = \mathcal{R}'$
$\ldots$

(b) Guarantee condition for relaxed write

Fig. 15: Guarantee conditions for release write vs relaxed write

These conditions describe the effect of actions. A rely condition $\mathsf{rely}(\mathcal{R}, \alpha)$ for action α, denoting a set of resource triples, signifies what the action expects from its incoming resource triples, where $\mathcal{R}$ represents α's precondition, and $\mathcal{R}_{\mathsf{rely}}$ represents the resource triples after taking possible environment moves under consideration. A guarantee condition $\mathsf{guar}(\mathcal{R}_{\mathsf{pre}}, \mathcal{R}, \alpha)$ signifies that α guarantees to produce pairs of resource triples $(\mathcal{R}_{\mathsf{sb}}, \mathcal{R}_{\mathsf{rf}})$, in which $\mathcal{R}_{\mathsf{sb}}$ is left for its sb successors and $\mathcal{R}_{\mathsf{rf}}$ is for its potential readers.

α	$\mathcal{R}_{\mathsf{rely}}$	$\mathcal{R}_{\mathsf{rely}} \in \mathsf{rely}(\mathcal{R}, \alpha)$ if
$\mathbb{R}(l, V, \mathtt{na})$	$\mathcal{R}$	$\mathcal{R}[\mathsf{L}][\ell] = \mathtt{na}(V') \Rightarrow V = V'$
$\mathbb{R}(l, V, \mathtt{rlx})$	$\mathcal{R}'$	$\exists \mathcal{R}_{\mathsf{rf}}.\ \mathcal{R}[\mathsf{L}][\ell] = \mathtt{at}(-) \Rightarrow \mathcal{R}_{\mathsf{rf}} \in \mathsf{envMove}(\mathcal{R}, \ell, V)$ $\wedge\ \mathcal{R}'[\mathsf{S}] = \mathcal{R}[\mathsf{S}] \wedge \mathcal{R}'[\mathsf{A}] = \mathcal{R}[\mathsf{A}] \oplus \lvert\mathcal{R}_{\mathsf{rf}}[\mathsf{S}]\rvert$ $\wedge\ \mathcal{R}'[\mathsf{L}] = \mathcal{R}[\mathsf{L}][\ \ell := \mathcal{R}_{\mathsf{rf}}[\mathsf{S}][\ell]\ ]$
$\mathbb{R}(l, V, \mathtt{acq})$	$\mathcal{R}'$	$\exists \mathcal{R}_{\mathsf{rf}}.\ \mathcal{R}[\mathsf{L}][\ell] = \mathtt{at}(-) \Rightarrow \mathcal{R}_{\mathsf{rf}} \in \mathsf{envMove}(\mathcal{R}, \ell, V)$ $\wedge\ \forall n \neq \mathsf{L}.\ \mathcal{R}'(n) = \mathcal{R}(n) \wedge \mathcal{R}'[\mathsf{L}] = \mathcal{R}[\mathsf{L}] \oplus \lvert\mathcal{R}_{\mathsf{rf}}[\mathsf{S}]\rvert$
$\mathbb{W}(l, V, \mathtt{at})$	$\mathcal{R}$	$\mathcal{R}[\mathsf{L}][\ell] = \mathtt{at}(-) \Rightarrow \exists V'.\ \mathsf{envMove}(\mathcal{R}, \ell, V') \neq \varnothing$
$\mathbb{U}(l, V, V')$	$\mathcal{R}'$	$\exists \mathcal{R}_{\mathsf{rf}}.\ \mathcal{R}[\mathsf{L}][\ell] = \mathtt{at}(-) \Rightarrow \mathcal{R}_{\mathsf{rf}} \in \mathsf{envMove}(\mathcal{R}, \ell, V)$ $\wedge\ \forall n \neq \mathsf{L}.\ \mathcal{R}'(n) = \mathcal{R}(n) \wedge \mathcal{R}'[\mathsf{L}] = \mathcal{R}[\mathsf{L}] \oplus \mathcal{R}_{\mathsf{rf}}[\mathsf{S}]$
otherwise	$\mathcal{R}$	always

Fig. 16: Rely conditions for actions

Note that the possible states for a physical location are uninit for uninitialised, $\mathtt{na}(V)$ for a non-atomic location holding value V, $\mathtt{at}(\tau, S)$ for an atomic location following protocol τ with a trace of state changes recorded in S, and $\perp$ for empty.

7 Soundness

We formulate the soundness of our proposed program logic in this section. As in GPS, our reasoning is compositional, i.e. triples about each program are proved separately

α	$(\mathcal{R}_{\mathsf{sb}}, \mathcal{R}_{\mathsf{rf}}) \in \mathsf{guar}(\mathcal{R}_{\mathsf{pre}}, \mathcal{R}, \alpha)$ if
$\mathbb{S}$	$\mathcal{R}_{\mathsf{rf}} = \mathtt{EMP} \wedge \mathcal{R}_{\mathsf{sb}} = \mathcal{R}$
$\mathbb{A}(\ell..\ell')$	$\mathcal{R}_{\mathsf{rf}} = \mathtt{EMP} \wedge \forall n \neq \mathsf{L}.\ \mathcal{R}_{\mathsf{sb}}(n) = \mathcal{R}(n) \wedge \mathcal{R}[\mathsf{L}] = \mathcal{R}[\mathsf{L}][\ell..\ell' := \mathsf{uninit}]$
$\mathbb{R}(\ell, V, \mathtt{na})$	$\mathcal{R}_{\mathsf{rf}} = \mathtt{EMP} \wedge \mathcal{R}_{\mathsf{sb}} = \mathcal{R} \wedge \mathcal{R}[\mathsf{L}] = \mathtt{na}(-)$
$\mathbb{R}(\ell, V, \mathtt{at})$	$\mathcal{R}_{\mathsf{rf}} = \mathtt{EMP} \wedge \mathcal{R}_{\mathsf{sb}} = \mathcal{R} \wedge \mathcal{R}[\mathsf{L}] = \mathtt{at}(-)$
$\mathbb{W}(\ell, V, \mathtt{na})$	$\mathcal{R}_{\mathsf{rf}} = \mathtt{EMP} \wedge \mathcal{R}[\mathsf{L}][\ell] \in \{\mathsf{uninit}, \mathtt{na}(-)\} \wedge$ $\forall n \neq \mathsf{L}.\ \mathcal{R}_{\mathsf{sb}}(n) = \mathcal{R}(n) \wedge \mathcal{R}_{\mathsf{sb}}[\mathsf{L}] = \mathcal{R}[\mathsf{L}][\ \ell := \mathtt{na}(V)\ ]$
$\mathbb{W}(\ell, V, \mathtt{rlx})$	$\exists \tau, s, S, \mathcal{R}', r_{\mathsf{rf}}.$ $\begin{pmatrix} \mathcal{R}'[\mathsf{A}] = \mathcal{R}[\mathsf{A}] \wedge \mathcal{R}'[\mathsf{L}] = \mathcal{R}[\mathsf{L}][\ell := \mathtt{at}(\tau, S \cup \{s\})] \\ \wedge\ \mathcal{R}'[\mathsf{S}] = \mathcal{R}[\mathsf{S}][\ell := \mathtt{at}(\tau, S \cup \{s\})] \end{pmatrix}$ $\wedge\ (r_{\mathsf{rf}}, \mathtt{emp}, \mathtt{emp}) \in \mathsf{interp}(\tau)(s, V) \wedge \mathcal{R}_{\mathsf{rf}} = (\mathtt{emp}, r_{\mathsf{rf}}, \mathtt{emp})$ $\wedge\ \mathcal{R}_{\mathsf{rf}} \oplus \mathcal{R}_{\mathsf{sb}} = \mathcal{R}' \wedge \mathcal{R}_{\mathsf{pre}}[\mathsf{L}][\ell] \neq \bot$ $\wedge\ (\mathcal{R}[\mathsf{L}][\ell] = \mathsf{uninit} \wedge S{=}\varnothing \vee \mathcal{R}[\mathsf{L}][\ell] = \mathtt{at}(\tau, S) \wedge \forall s_0 \in S.\ s_0 \sqsubseteq_\tau s)$ $\wedge\ \forall \mathcal{R}_E \in \mathsf{envMove}(\mathcal{R}_{\mathsf{pre}}, \ell, -).\ \mathcal{R}_E[\mathsf{S}][\ell] \sqsubseteq_{\mathtt{at}} \mathcal{R}_{\mathsf{rf}}[\mathsf{S}][\ell]$
$\mathbb{W}(\ell, V, \mathtt{rel})$	$\exists \tau, s, S, \mathcal{R}', r_{\mathsf{rf}}.$ $\begin{pmatrix} \exists r_1, r_2.\ \mathcal{R}'[\mathsf{A}] = \mathcal{R}[\mathsf{A}] \wedge \mathcal{R}[\mathsf{L}] = r_1 \oplus r_2 \wedge r_2 \leq r_{\mathsf{rf}} \\ \wedge\ \mathcal{R}'[\mathsf{L}] = r_1[\ell := \mathtt{at}(\tau, S \cup \{s\})] \\ \wedge\ \mathcal{R}'[\mathsf{S}] = \mathcal{R}[\mathsf{S}] \oplus r_2[\ell := \mathtt{at}(\tau, S \cup \{s\})] \end{pmatrix}$ $\wedge\ (r_{\mathsf{rf}}, \mathtt{emp}, \mathtt{emp}) \in \mathsf{interp}(\tau)(s, V) \wedge \mathcal{R}_{\mathsf{rf}} = (\mathtt{emp}, r_{\mathsf{rf}}, \mathtt{emp})$ $\wedge\ \mathcal{R}_{\mathsf{rf}} \oplus \mathcal{R}_{\mathsf{sb}} = \mathcal{R}' \wedge \mathcal{R}_{\mathsf{pre}}[\mathsf{L}][\ell] \neq \bot$ $\wedge\ (\mathcal{R}[\mathsf{L}][\ell] = \mathsf{uninit} \wedge S{=}\varnothing \vee \mathcal{R}[\mathsf{L}][\ell] = \mathtt{at}(\tau, S) \wedge \forall s_0 \in S.\ s_0 \sqsubseteq_\tau s)$ $\wedge\ \forall \mathcal{R}_E \in \mathsf{envMove}(\mathcal{R}_{\mathsf{pre}}, \ell, -).\ \mathcal{R}_E[\mathsf{S}][\ell] \sqsubseteq_{\mathtt{at}} \mathcal{R}_{\mathsf{rf}}[\mathsf{S}][\ell]$
$\mathbb{U}(\ell, V, V')$	$\exists \tau, s, S, \mathcal{R}', r_{\mathsf{rf}}.$ $\begin{pmatrix} \exists r_1, r_2.\ \mathcal{R}'[\mathsf{A}] = \mathcal{R}[\mathsf{A}] \wedge \mathcal{R}[\mathsf{L}] = r_1 \oplus r_2 \wedge r_2 \leq r_{\mathsf{rf}} \\ \wedge\ \mathcal{R}'[\mathsf{L}] = r_1[\ell := \mathtt{at}(\tau, S \cup \{s\})] \\ \wedge\ \mathcal{R}'[\mathsf{S}] = \mathcal{R}[\mathsf{S}] \oplus r_2[\ell := \mathtt{at}(\tau, S \cup \{s\})] \end{pmatrix}$ $\wedge\ (r_{\mathsf{rf}}, \mathtt{emp}, \mathtt{emp}) \in \mathsf{interp}(\tau)(s, V) \wedge \mathcal{R}_{\mathsf{rf}} = (\mathtt{emp}, r_{\mathsf{rf}}, \mathtt{emp})$ $\wedge\ \mathcal{R}_{\mathsf{rf}} \oplus \mathcal{R}_{\mathsf{sb}} = \mathcal{R}' \wedge \mathcal{R}_{\mathsf{pre}}[\mathsf{L}][\ell] \neq \bot$ $\wedge\ (\mathcal{R}[\mathsf{L}][\ell] = \mathsf{uninit} \wedge S{=}\varnothing \vee \mathcal{R}[\mathsf{L}][\ell] = \mathtt{at}(\tau, S) \wedge \forall s_0 \in S.\ s_0 \sqsubseteq_\tau s)$
$\mathbb{F}(\mathtt{rel})$	$\mathcal{R}_{\mathsf{rf}} = \mathtt{EMP} \wedge \mathcal{R}_{\mathsf{sb}}[\mathsf{A}] = \mathcal{R}[\mathsf{A}] \wedge \mathcal{R}_{\mathsf{sb}}[\mathsf{L}] \oplus \mathcal{R}_{\mathsf{sb}}[\mathsf{S}] = \mathcal{R}[\mathsf{L}] \oplus \mathcal{R}[\mathsf{S}]$
$\mathbb{F}(\mathtt{acq})$	$\mathcal{R}_{\mathsf{rf}} = \mathtt{EMP} \wedge \exists r, r'.\ \mathcal{R}[\mathsf{A}] = r \oplus r'$ $\wedge\ \mathcal{R}_{\mathsf{sb}}[\mathsf{A}] = r \wedge \mathcal{R}_{\mathsf{sb}}[\mathsf{L}] = r' \oplus \mathcal{R}[\mathsf{L}] \wedge \mathcal{R}_{\mathsf{sb}}[\mathsf{S}] = \mathcal{R}[\mathsf{S}]$

Fig. 17: Guarantee conditions for actions

and then linked together using the `let` and `fork` rules. To bridge the gap between such local reasoning and the underlying global semantics, similar to GPS, we formulate the notion of *local safety* and *global safety*, so as to demonstrate the soundness of the proposed reasoning system.

7.1 Local Safety

Based on rely-guarantee reasoning [12, 25], the *local safety* for a thread says that the actions the thread controls confirm to their guarantees, assuming actions the environment controls respect their rely conditions, as similarly shown in GPS. However our

proof is different from GPS's as our rely/guarantee definitions are based on resource triples to cover more actions and capture the subtle differences between them.

Using the rely and guarantee definitions, $\mathsf{LSafe}_n(e, \Phi)$ is defined as the set of resource triples on which command e can safely execute for n steps and end up with postcondition Φ, which is a mapping from value to proposition interpretation, being satisfied:

$\mathcal{R} \in \mathsf{LSafe}_0(e, \Phi) \triangleq$ always
$\mathcal{R} \in \mathsf{LSafe}_{n+1}(e, \Phi) \triangleq$
  If $e \in \mathit{Val}$ then $\mathcal{R} \in [\![\Phi(e)]\!]^\rho$
  If $e = K[\mathtt{fork}\ e']$ then $\mathcal{R} \in \mathsf{LSafe}_n(K[0], \Phi) * \mathsf{LSafe}_n(e', \mathsf{true})$
  If $e \xrightarrow{\alpha} e'$ then $\forall \mathcal{R}_F \# \mathcal{R}. \quad \forall \mathcal{R}_{pre} \in \mathsf{rely}(\mathcal{R} \oplus \mathcal{R}_F, \alpha).$
    $\exists P'.\ \mathcal{R}_{pre} \in [\![P']\!]^\rho \wedge \forall \mathcal{R}' \in [\![P']\!]^\rho.\ (\mathcal{R}_{pre}, \mathcal{R}') \in \mathsf{wpe}(\alpha)$
      $\Longrightarrow \exists \mathcal{R}_{post}.\ (\mathcal{R}_{post} \oplus \mathcal{R}_F, -) \in \mathsf{guar}(\mathcal{R}_{pre}, \mathcal{R}', \alpha) \wedge \mathcal{R}_{post} \in \mathsf{LSafe}_n(e', \Phi)$

Note that the expression e is actually executed with the state $\mathcal{R}'$, taking into account the possible interference from environment as long as it respects the rely condition for α. Note also the wpe is a sanity check, ensuring some obvious faults will not occur like re-allocating some locations, which is defined as:

α	$(\mathcal{R}_{\mathsf{pre}}, \mathcal{R}') \in \mathsf{wpe}(\alpha)$ if
$\mathbb{A}(\ell_1..\ell_n)$	$\forall i.\ 1 \le i \le n \Rightarrow \mathcal{R}'(\ell_i) = \bot$
$\mathbb{W}(\ell, -, \mathtt{at})$	$\mathcal{R}_{\mathsf{pre}}[\mathsf{L}][\ell] = \mathtt{at}(-) \wedge \mathcal{R}'[\mathsf{L}][\ell] = \mathtt{at}(-)$ $\Rightarrow\ \exists \mathcal{R}_E \in \mathsf{envMove}(\mathcal{R}_{\mathsf{pre}}, \ell, -).\ \mathcal{R}_E[\mathsf{L}][\ell] = \mathcal{R}'[\mathsf{L}][\ell]$
$\mathbb{U}(\ell, -, -)$	$\mathcal{R}_{\mathsf{pre}}[\mathsf{L}][\ell] = \mathtt{at}(-) \Rightarrow \mathcal{R}'[\mathsf{L}][\ell] \equiv \mathcal{R}_{pre}[\mathsf{L}][\ell]$

As in GPS, we define local soundness (with respect to the semantics for a Hoare triple) as: $\rho \vDash \{P\}\ e\ \{x.Q\} \triangleq \forall n, \mathcal{R} \in [\![P]\!]^\rho.\ \mathcal{R} \in \mathsf{LSafe}_n(e, [\![x.Q]\!]^\rho)$.
Intuitively it says given any state $\mathcal{R}$ that satisfies the precondition P, the expression e is safe to execute as many steps as possible and we can expect when it terminates with return value v, $Q[v/x]$ holds, where x is a place holder for the return value in Q and can be omitted if Q does not describe the return value.

Theorem 1 (local soundness) . *Our verification logic (presented in sec 4) is locally sound. That is, if* $\{P\}\ e\ \{x.Q\}$ *is provable, then for all closing* ρ *we have* $\rho \vDash \{P\}\ e\ \{x.Q\}$.

Proof This theorem can be proved by using structural induction on e. Base cases, including all the new verification rules in Fig. 12, are proved by using the last entry of LSafe_1 and check it with their rely and guarantee definitions. Then programs are linked using let-binding (including fork). □

7.2 Global Safety and the Final Soundness Theorem

Same as GPS our logic is for concurrent programs, in its soundness proof, besides showing the inference rules are valid according to the semantics of triples, we also need to show if a triple $\{P\}e\{x.Q\}$ holds e does not contain any data races, memory errors, nor dangling reads. However GPS+ allows relaxed operations, which means unlike that in GPS, our rf relations do not ensure hb relations. That means to prove properties like data-race free becomes much trickier.

To formally discuss global soundness, we first give following definitions: In an execution graph, two events are *conflicting* if they both access a same location while

at least one of them is a write action, and at least one of them is non-atomic. A graph G contains *data race*, written as $\mathsf{dataRace}(G)$, if there is at least one pair of conflicting accesses a and b, and we have $\neg\mathsf{hb}(a,b) \wedge \neg\mathsf{hb}(b,a)$. We say a graph G contains *memory error*, written as $\mathsf{memErr}(G)$, if there exists an event a that accesses the location l, but there does not exist an event that happens before a and allocates the location l. We say in a graph G has *initialised reads* written as $\mathsf{initReads}(G)$, if for every read event in G, there is a write event accesses the same location and happens before it.

Then we define the execution and the result of a program e using the machine step semantics defined in §2.

$$\mathsf{execs}(e) \triangleq \{(e', G) \mid \langle [i \mapsto (\mathsf{start}, e)]; ([\mathsf{start} \mapsto \mathbb{S}], \varnothing, \varnothing, \varnothing)\rangle \longrightarrow^* \langle [i \mapsto (-, e') \uplus T]; G\rangle\}$$

$$[\![e]\!] \triangleq \begin{cases} \textbf{error} & \exists(-, G) \in \mathsf{execs}(e).\ \mathsf{dataRace}(G) \vee \mathsf{memErr}(G) \\ \{V \mid (V, -) \in \mathsf{execs}(e)\} & \text{otherwise} \end{cases}$$

With the semantics for a program e defined, we can formulate the global soundness as: *if* $\vdash \{\mathsf{true}\}\ e\ \{x.P\}$ *then* $[\![e]\!] \subseteq \{V \mid [\![P[V/x]]\!] \neq \varnothing\}$, which ensures that Hoare triples proved in the proposed system accurately predict the final result of a closed program, according to the C11 memory model. To prove this, we introduce the notion of *global safety* based on a global event graph G.

$$\begin{array}{l} \mathsf{GSafe}_n(\mathcal{T}, G, \mathcal{L}) \triangleq \\ \quad \mathsf{valid}(G, \mathcal{L}, N) = N \wedge \mathsf{compat}(G, \mathcal{L}) \wedge \mathsf{conform}(G, \mathcal{L}, N) \wedge \\ \quad \forall a \in N.\ \mathcal{L}(\mathsf{sb}, a, \bot) = \bigoplus\{\mathcal{R} \mid \exists i.\ \mathcal{T}(i) = (a, -, \mathcal{R}, -)\} \wedge \\ \quad \forall i.\ \mathcal{T}(i) = (a, e, \mathcal{R}, \Phi) \Longrightarrow \mathcal{R} \in \mathsf{LSafe}_n(e, \Phi) \\ \text{where } N \triangleq \mathsf{dom}(G.A) \end{array}$$

The $\mathcal{L}$ is a labeling map that associates the graph's edges with the resource triples passed through. The $\mathcal{T} \in \mathit{IThreadMap} \triangleq \{\mathbb{N} \to (a, e, \mathcal{R}, \Phi)\}$ is an instrumented thread pool. It maps each thread to a tuple $(a, e, \mathcal{R}, \Phi)$, where a is a thread's last event in the graph, e is the thread's continuation, $\mathcal{R}$ is the resource triple it currently owns, and its postcondition is described by Φ. A instrumented thread pool can be down casted to a machine thread pool T using $\mathsf{erase}(\mathcal{T})$, where $\forall i.\ \mathcal{T} = (a, e, \mathcal{R}, \Phi) \Rightarrow \mathsf{erase}(\mathcal{T})(i) = (a, e)$. The $\mathsf{valid}(G, \mathcal{L}, \mathsf{dom}(G.A))$ is the set of properly labeled nodes. Being equal with $\mathsf{dom}(G.A)$, it ensures that all nodes are labeled properly. The $\mathsf{compat}(\mathsf{G})$ asserts any set of hb-independent edges in G, the sum of the resource triples they carry is defined. We say a set of edges $\mathcal{T}$ hb-*independent* if for all edges $(a, a'), (b, b') \in \mathcal{T}$ we have $\neg\mathsf{hb}^{=}(a', b)$. The conform checks the state transitions for atomic writes respect the mo order[23].

Intuitively, the $\mathsf{GSafe}_n(\mathcal{T}, G, \mathcal{L})$ states from a graph G and with the resource triples given in $\mathcal{L}$, any thread from $\mathcal{T}$ is safe to execute under the global context up to n steps. We would like to ensure the GSafe property holds during the program execution and the growth of the graph, until there is nothing to be executed. Therefore we first introduce our way to do the labeling after a new node is added to the graph (an action is executed), and use five lemmas (whose proofs are left in report [10]) to demonstrate that the properties required for GSafe will be restored after labels are added for a new node.

When a new node b is added to the graph, we first label its sb incoming edge. Suppose b's sb predecessor is a (if b is the first event of the execution, its sb predeces-

sor is a $\mathbb{S}$ node with empty resource triples on the outgoing edges), a's sb outgoing resource triple initially goes into a sink edge $\mathsf{sb}(a, \bot)$. If b is the first event in a forked thread, we take $\mathcal{R}$ from the sink edge and label $\mathsf{sb}(a, b)$ with it, leaving $\mathcal{R}_{\text{rem}}$ for a's local thread; otherwise, we take everything to label $\mathsf{sb}(a, b)$ leaving $\mathcal{R}_{\text{rem}}$ empty. By labeling the new sb edge in this way, the following lemma can be proved.

Lemma 1 (Step Preparation) .

$$
\begin{array}{ll}
\textit{if}\ \mathsf{consistentC11}(G) & \textit{then}\ \exists \mathcal{L}'.\ \mathsf{dom}(G.A) = \mathsf{valid}(G', \mathcal{L}', \mathsf{dom}(G.A)) \\
\quad \wedge \mathsf{consistentC11}(G') & \quad \wedge \mathsf{compat}(G', \mathcal{L}') \\
\quad \wedge \mathsf{dom}(G', A) = \mathsf{dom}(G.A) \uplus b & \quad \wedge \mathsf{conform}(G', \mathcal{L}', \mathsf{dom}(G'.A)) \\
\quad \wedge \mathcal{L}(\mathsf{sb}, a, \bot) = \mathcal{R} \oplus \mathcal{R}_{rem} & \quad \wedge \mathcal{L}'(\mathsf{sb}, a, \bot) = \mathcal{R}_{rem} \\
\quad \wedge \mathsf{dom}(G.A) \subseteq \mathsf{valid}(G, \mathcal{L}, \mathsf{dom}(G.A)) & \quad \wedge \mathsf{in}(\mathcal{L}', b, \mathsf{sb}) = \mathcal{R} \\
\quad \wedge \mathsf{compat}(G, \mathcal{L}) \wedge \mathsf{conform}(G, \mathcal{L}, N) & \quad \wedge \mathsf{in}(\mathcal{L}', b, \mathsf{rf}) = \mathtt{EMP} \\
\quad \wedge \forall c \in \mathsf{dom}(G.A).\ G.A(c) = G'.A(c) & \quad \wedge \mathsf{in}(\mathcal{L}', b, \mathsf{esc}) = \mathtt{EMP} \\
\quad \wedge G'.\mathsf{sb} = G.\mathsf{sb} \uplus [a, b) & \quad \wedge \forall a' \neq a.\ \mathcal{L}'(\mathsf{sb}, a', b) = \mathtt{EMP} \\
\quad \wedge \forall c \in \mathsf{dom}(G.A).\ G.\mathsf{rf}(c) = G'.\mathsf{rf}(c) & \quad \wedge \mathsf{out}(\mathcal{L}', b, \mathsf{all}) = \mathtt{EMP} \\
\quad \wedge G'.\mathsf{mo} \supseteq G.\mathsf{mo} & \quad \wedge \forall a' \neq a.\ \mathcal{L}'(\mathsf{sb}, a', \bot) = \mathcal{L}(\mathsf{sb}, a', \bot)
\end{array}
$$

Note that the $\mathsf{in}(\mathcal{L}, a, \mathtt{t})$ and $\mathsf{out}(\mathcal{L}, a, \mathtt{t})$ are used to represent the sum of resource triples on a's incoming or outgoing edges with type $\mathtt{t}$.

Then we label the rf incoming edge. Note that this labeling is for atomic read actions only (including $\mathtt{CAS}$). A non-atomic load reads from local resource triple carried in by sb incoming edge. But an atomic load can read from any writer to the same location as long as the $\mathsf{consistentC11}$ holds. A writer's outgoing rf resource triple, in the form of $(\mathtt{emp}, r, \mathtt{emp})$, initially goes to its rf sink edge. If the newly added read event reads from it, we label the rf edge between them in three different manners according to the reader's type. If the reader is relaxed (relaxed atomic read or failed $\mathtt{CAS}$), we label its rf incoming edge using a resource triple $(\mathtt{emp}, \mathtt{emp}, |r|)$; if it is an acquire read, we label its rf incoming edge using $(|r|, \mathtt{emp}, \mathtt{emp})$; and if it is an update read (successful $\mathtt{CAS}$), we shall use $(r, \mathtt{emp}, \mathtt{emp})$ and change the label of writer's rf sink edge to $(\mathtt{emp}, |r|, \mathtt{emp})$. This process gives us the following lemma.

Lemma 2 (Rely Step) .

$$
\begin{array}{ll}
\textit{if}\ G.A(a) = \alpha & \textit{then}\ \exists \mathcal{L}'.\ N \subseteq \mathsf{valid}(G, \mathcal{L}', N) \\
\quad \wedge \mathsf{dom}(G.A) = N \uplus a & \quad \wedge \mathsf{compat}(G, \mathcal{L}) \\
\quad \wedge N \in \mathsf{prefix}(G) \wedge N \subseteq \mathsf{valid}(G, \mathcal{L}, N) & \quad \wedge \mathsf{conform}(G, \mathcal{L}, N) \\
\quad \wedge \mathsf{in}(\mathcal{L}, a, \mathsf{all}) = \mathsf{out}(\mathcal{L}, a, \mathsf{all}) = \mathtt{EMP} & \quad \wedge \mathsf{in}(\mathcal{L}', a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}', a, \mathsf{rf}) \\
\quad \wedge \mathsf{compat}(G, \mathcal{L}) \wedge \mathsf{conform}(G, \mathcal{L}, N) & \qquad \in \mathsf{rely}(\mathsf{in}(\mathcal{L}', a, \mathsf{sb}), \alpha) \\
\quad \wedge \mathsf{consistentC11}(G) & \quad \wedge \mathsf{in}(\mathcal{L}', a, \mathsf{esc}) = \mathsf{out}(\mathcal{L}', a, \mathsf{all}) = \mathtt{EMP} \\
\quad \wedge \mathsf{in}(\mathcal{L}, a, \mathsf{rf}) = \mathtt{EMP} \wedge \mathsf{in}(\mathcal{L}, a, \mathsf{esc}) = \mathtt{EMP} & \quad \wedge \forall b, c.\ \mathcal{L}'(\mathsf{sb}, b, c) = \mathcal{L}(\mathsf{sb}, b, c) \\
\quad \wedge \mathsf{out}(\mathcal{L}, a, \mathsf{all}) = \mathtt{EMP} & \quad \wedge \forall b.\ \mathcal{L}'(\mathsf{sb}, b, \bot) = \mathcal{L}(\mathsf{sb}, b, \bot)
\end{array}
$$

Next we consider the resource transfer via ghost moves (essentially creating and redeeming of escrows). Given an escrow defined as $\sigma : P \rightsquigarrow P'$ and a node a owns resource triple $(r_{\text{rem}} \oplus r_{\text{esc}}, -, -)$ where $(r_{\text{esc}}, \mathtt{emp}, \mathtt{emp}) \in [\![P']\!]$. We can put $(r_{esc}, \mathtt{emp}, \mathtt{emp})$ under escrow, i.e. move it to a's esc sink edge and add σ to r_{rem}'s known escrow set. On the other hand, if a node b owns resource triple $(r \oplus r_{\text{cond}}, -, -)$, where $\sigma \in r.\Pi \wedge (r_{\text{cond}}, \mathtt{emp}, \mathtt{emp}) \in [\![P]\!]$. It can consume the escrow condition r_{cond} by disposing it to b's cond sink edge and redeem the resource triple under escrow by moving it from a's esc sink edge to edge $\mathsf{esc}(a, b)$. In the following lemmas we check the ghost moves are correct and the labeling so far satisfies new action's wpe.

Lemma 3 (Ghost Step) .

$$
\begin{array}{ll}
\textit{if} & \mathsf{dom}(G.A) = N \uplus a \wedge N \in \mathsf{prefix}(G) \wedge N \subseteq \mathsf{valid}(G, \mathcal{L}, \mathsf{dom}(G.A)) \\
& \wedge \mathsf{compat}(G, \mathcal{L}[(\mathsf{esc}, -, a, \bot) := \mathcal{L}(\mathsf{esc}, -, a, \bot) \oplus \mathcal{R}]) \\
& \wedge \mathsf{conform}(G, \mathcal{L}, N) \wedge \mathsf{consistentC11}(G) \\
& \wedge \mathcal{R}_{\mathit{before}} \triangleq \mathsf{in}(\mathcal{L}, a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}, a, \mathsf{rf}) \oplus \mathsf{in}(\mathcal{L}, a, \mathsf{esc}) \\
& \wedge \mathcal{R}_{\mathit{after}} \triangleq \mathcal{R} \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{esc}) \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{cond}) \\
& \wedge \mathcal{R}_{\mathit{before}} \Rrightarrow_{\mathcal{I}} \mathcal{R}_{\mathit{after}} \wedge |\mathcal{R}_{\mathit{before}}| \leq \mathcal{R} \wedge \mathcal{R} \Rrightarrow \mathcal{P} \wedge \forall c.\ \mathcal{L}(\mathsf{esc}, -, a, e) = \mathtt{EMP} \\
& \wedge \forall (\sigma, \mathcal{R}_E) \in \mathcal{I}.\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}') \Rightarrow \mathcal{R}_E \in \mathcal{Q}' \\
& \wedge \mathcal{L}(\mathsf{esc}, a, \bot) = \bigoplus \left\{ \mathcal{R}_E \,\middle|\, \begin{array}{l} (\sigma, \mathcal{R}_E) \in \mathcal{I}, \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}'), \mathcal{R}_E \in \mathcal{Q}', \\ (\nexists b.\ \mathsf{hb}^{=}(a, b) \wedge \mathcal{L}(\mathsf{cond}, b, \bot) \in \mathcal{Q}) \end{array} \right\} \\
\textit{then} & \exists \mathcal{L}', \mathcal{I}', \mathcal{R}', \mathcal{R}'_{\mathit{before}}, \mathcal{R}'_{\mathit{after}} \in \mathcal{P}. \\
& N \subseteq \mathsf{valid}(G, \mathcal{L}', \mathsf{dom}(G.A)) \\
& \wedge \mathsf{compat}(G, \mathcal{L}'[(\mathsf{esc}, -, a, \bot) := \mathcal{L}'(\mathsf{esc}, -, a, \bot) \oplus \mathcal{R}']) \\
& \wedge \mathsf{conform}(G, \mathcal{L}', N) \\
& \wedge \mathcal{R}'_{\mathit{before}} \triangleq \mathsf{in}(\mathcal{L}', a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}', a, \mathsf{rf}) \oplus \mathsf{in}(\mathcal{L}', a, \mathsf{esc}) \\
& \wedge \mathcal{R}'_{\mathit{after}} \triangleq \mathcal{R}' \oplus \mathsf{out}(\mathcal{L}', a, \mathsf{esc}) \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{cond}) \wedge \mathcal{R}'_{\mathit{before}} \Rrightarrow_{\mathcal{I}'} \mathcal{I}' \mathcal{R}'_{\mathit{after}} \\
& \wedge \forall b.\ \mathcal{L}'(\mathsf{sb}, b, \bot) = \mathcal{L}(\mathsf{sb}, b, \bot) \wedge \forall b.\ \mathcal{L}'(\mathsf{rf}, b, \bot) = \mathcal{L}(\mathsf{rf}, b, \bot) \\
& \wedge \forall b, c.\ \mathcal{L}'(\mathsf{sb}, b, c) = \mathcal{L}(\mathsf{sb}, b, c) \wedge \forall b, c.\ \mathcal{L}'(\mathsf{rf}, b, c) = \mathcal{L}(\mathsf{rf}, b, c) \\
& \wedge \forall c.\ \mathcal{L}'(\mathsf{esc}, -, a, e) = \mathtt{EMP} \wedge \forall (\sigma, \mathcal{R}_E) \in \mathcal{I}'.\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}') \Rightarrow \mathcal{R}_E \in \mathcal{Q}' \\
& \wedge \mathcal{L}(\mathsf{esc}, a, \bot) = \bigoplus \left\{ \mathcal{R}_E \,\middle|\, \begin{array}{l} (\sigma, \mathcal{R}_E) \in \mathcal{I}', \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}'), \mathcal{R}_E \in \mathcal{Q}', \\ (\nexists b.\ \mathsf{hb}^{=}(a, b) \wedge \mathcal{L}(\mathsf{cond}, b, \bot) \in \mathcal{Q}) \end{array} \right\}
\end{array}
$$

Lemma 4 (Protocol Equivalence for Writes) .

$$
\begin{array}{ll}
\textit{if} & \mathsf{dom}(G.A) = N \uplus a \wedge N \in \mathsf{prefix}(G) \wedge N \subseteq \mathsf{valid}(G, \mathcal{L}, \mathsf{dom}(G.A)) \\
& \wedge \mathsf{compat}(G, \mathcal{L}[(\mathsf{esc}, -, a, \bot) := \mathcal{L}(\mathsf{esc}, -, a, \bot) \oplus \mathcal{R}]) \\
& \wedge \mathsf{conform}(G, \mathcal{L}, N) \wedge \mathsf{consistentC11}(G) \\
& \wedge \mathsf{in}(\mathcal{L}, a, \mathsf{all}) \Rrightarrow_{\mathcal{I}} \mathcal{R} \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{esc}) \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{cond}) \\
& \wedge |\mathsf{in}(\mathcal{L}, a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}, a, \mathsf{rf})| \leq \mathcal{R} \\
\textit{then} & (\mathsf{in}(\mathcal{L}, a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}, a, \mathsf{rf}), \mathsf{in}(\mathcal{L}, a, \mathsf{all})) \in \mathsf{wpe}(G.A(a))
\end{array}
$$

Finally in guarantee step, we label a node's outgoing sb and rf edges using the corresponding resource triples generated according to the action's guarantee definition. Initially these resource triples go to sink nodes, until their future sb or rf successors are added to the graph and take them to label the corresponding edges.

Lemma 5 (Guarantee Step) .

$$
\begin{array}{ll}
\textit{if} & G.A(a) = \alpha \wedge \mathsf{dom}(G.A) = N \uplus a \wedge N \in \mathsf{prefix}(G) \wedge N \subseteq \mathsf{valid}(G, \mathcal{L}, \mathsf{dom}(G.A)) \\
& \wedge \mathsf{compat}(G, \mathcal{L}[(\mathsf{esc}, -, a, \bot) := \mathcal{L}(\mathsf{esc}, -, a, \bot) \oplus \mathcal{R}]) \\
& \wedge \mathsf{conform}(G, \mathcal{L}, N) \wedge \mathsf{consistentC11}(G) \\
& \wedge \mathcal{R}_{\mathsf{pre}} = \mathsf{in}(\mathcal{L}, a, \mathsf{sb}) \oplus \mathsf{in}(\mathcal{L}, a, \mathsf{rf}) \\
& \wedge \mathsf{in}(\mathcal{L}, a, \mathsf{all}) \Rrightarrow_{\mathcal{I}} \mathcal{R} \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{esc}) \oplus \mathsf{out}(\mathcal{L}, a, \mathsf{cond}) \\
& \wedge \mathcal{R}_{\mathsf{pre}} \in \mathsf{rely}(-, \alpha) \wedge |\mathsf{in}(\mathcal{L}, a, \mathsf{all})| \leq \mathcal{R} \\
& \wedge \forall (\sigma, \mathcal{R}_E) \in \mathcal{I}.\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}') \Rightarrow \mathcal{R}_E \in \mathcal{Q}' \\
& \wedge \mathcal{L}(\mathsf{esc}, a, \bot) = \bigoplus \left\{ \mathcal{R}_E \,\middle|\, \begin{array}{l} (\sigma, \mathcal{R}_E) \in \mathcal{I}, \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}'), \mathcal{R}_E \in \mathcal{Q}', \\ (\nexists b.\ \mathsf{hb}^{=}(a, b) \wedge \mathcal{L}(\mathsf{cond}, b, \bot) \in \mathcal{Q}) \end{array} \right\} \\
& \wedge \mathsf{out}(\mathcal{L}, a, \mathsf{sb}) = \mathsf{out}(\mathcal{L}, a, \mathsf{rf}) = \mathtt{EMP} \\
& \wedge (\mathcal{R}_{\mathsf{sb}}, \mathcal{R}_{\mathsf{rf}}) \in \mathsf{guar}(\mathcal{R}_{\mathsf{pre}}, \mathcal{R}, \alpha) \wedge \mathsf{wpe}(\alpha, \mathcal{R}_{\mathsf{pre}}, \mathsf{in}(\mathcal{L}, a, \mathsf{all})) \\
\textit{then} & \exists \mathcal{L}'. \\
& \wedge \mathsf{dom}(G.A) = \mathsf{valid}(G, \mathcal{L}', \mathsf{dom}(G.A)) \\
& \wedge \mathsf{compat}(G, \mathcal{L}') \wedge \mathsf{conform}(G, \mathcal{L}', \mathsf{dom}(G.A)) \\
& \wedge \forall b \neq a.\ \mathcal{L}'(\mathsf{sb}, b, \bot) = \mathcal{L}(\mathsf{sb}, b, \bot) \wedge \mathcal{L}'(\mathsf{sb}, a, \bot) = \mathcal{R}_{\mathsf{sb}}
\end{array}
$$

The proofs for the above-mentioned five lemmas are left in report [10]. We now present the Theorem *Instrumented Execution*, which says that given a program and

graph configuration that is globally safe for $n+1$ steps, any legal machine step (scheduled arbitrarily) will transform it into a new configuration that is globally safe for n steps.

Theorem 2 (Instrumented Execution) .
If $\mathsf{GSafe}_{n+1}(\mathcal{T}, G, \mathcal{L}) \wedge \langle \mathsf{erase}(\mathcal{T}); G\rangle \longrightarrow \langle T'; G'\rangle$
then $\exists \mathcal{T}', \mathcal{L}'.\ \mathsf{erase}(\mathcal{T}') = T' \wedge \mathsf{GSafe}_n(\mathcal{T}', G', \mathcal{L}')$.

Proof Starting from $\mathsf{GSafe}_{n+1}(\mathcal{T}, G, \mathcal{L})$, a machine step will tranform the graph into G' and thread pool into T', leaving LSafe_n for the thread that takes the move. By applying the labeling lemmas, Lemma 1, 2, 3, 4 and 5 consecutively to the graph G' with the newly added node, we can find a new labeling $\mathcal{L}'$ and establish the valid, compat, and $\mathsf{conform}$ along with other properties for the new graph and labeling. Therefore, the new graph is $\mathsf{GSafe}_n(\mathcal{T}', G', \mathcal{L}')$. □

In what follows we present one more lemma, whose proof is left in report [10].

Lemma 6 (Error Free) . *If* $\mathsf{GSafe}_n(\mathcal{T}, G, \mathcal{L})$ *then we have* $\neg\mathsf{dataRace}(G), \neg\mathsf{memErr}(G)$, *and all reads are initialised.*

Finally we move on to the global soundness theorem *Adequacy*.

Theorem 3 (Adequacy) . *If* $\vdash \{\mathsf{true}\}\ e\ \{x.P\}$ *then* $[\![e]\!] \subseteq \{V \mid [\![P[V/x]]\!] \neq \varnothing\}$.

Proof Firstly, given the triple is syntactically sound, we know starting with its precondition true, it is LSafe for any number of steps. As we are only interested in partical correctness, we assume the program terminates within n steps, i.e. after n steps, e is transformed into a pure value V and all its children threads terminate too. We can assert $\mathsf{LSafe}_{n+1}(e, [\![x.P]\!])$, and construct

$$\mathsf{GSafe}_{n+1}\left(\begin{array}{l}[0 \mapsto (\mathsf{start}, e, \mathtt{emp}, [\![x.P]\!])], ([\mathsf{start} \mapsto \mathbb{S}], \varnothing, \varnothing, \varnothing),\\ [(\mathsf{sb}, \mathsf{start}, \bot) \mapsto \mathtt{EMP}] \uplus [(\mathsf{rf}, \mathsf{start}, \bot) \mapsto \mathtt{EMP}]\end{array}\right).$$

Then according to Theorem *Instrumented Execution*, when the program terminates after n steps, we have $\exists \mathcal{T}', \mathcal{R}.\ \mathsf{GSafe}_1(\mathcal{T}' \uplus [0 \mapsto (-, V, \mathcal{R}, [\![x.P]\!])], -, -)$, which implies $\mathcal{R} \Rrightarrow [\![P[V/x]]\!]$ and thus $[\![P[V/x]]\!] \neq \varnothing$. □

8 Conclusion

We present a verification logic for weak memory programs GPS+, by enhancing the GPS mechanism with two new forms of assertions: shareable assertions $\langle P\rangle$ and waiting-to-be-acquired assertions $\boxtimes P$. This change enables us to control more precisely the synchronisations that happen between threads, making the reasoning about relaxed atomics and fences possible.

Our work is closely related to GPS [23] and RSL [24], both of which focus on program verification under the C11 weak memory model. RSL was intended to provide support for reasoning about release-acquire accesses in the style of Concurrent Separation Logic (CSL) [19]. Our logic inherits several ideas from GPS, including per-location protocols and escrows, which are also relevant with a previous work [22]. Another important concept we borrow from GPS are ghost resources as PCMs. This idea is related with [5], [11], [16], and a recent work [13].

Future work includes the incorporation of *release sequence* and the consideration of more memory orders like *consume load*. The most recent work [21] demonstrates the power of GPS in reasoning about real code and inspires us to apply our logic to more real code.

References

1. ISO/IEC 9899:2011. Programming Language C. 2011.
2. Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 55–66, New York, NY, USA, 2011. ACM.
3. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09)*, pages 23–42, 2009.
4. P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A Logic for Time and Data Abstraction. In *ECOOP*, pages 207–231. 2014.
5. T. Dinsdale-Young, L. Birkedal, P. Gardner, M. J. Parkinson, and H. Yang. Views: Compositional Reasoning for Concurrent Programs. In *ACM POPL*, pages 287–300, 2013.
6. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *ECOOP*, pages 504–528, 2010.
7. M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-Guarantee Reasoning. In *ESOP*, pages 363–377, 2009.
8. Marko Doko and Viktor Vafeiadis. *A Program Logic for C11 Memory Fences*, pages 413–430. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.
9. X. Feng. Local Rely-guarantee Reasoning. In *ACM POPL*, pages 315–327, 2009.
10. M. He, V. Vafeiadis, S. Qin, and J. F. Ferreira. GPS+ : Reasoning about Fences and Relaxed Atomics (Technical Report), 2017. http://pls.tees.ac.uk/gpsp/report.pdf, School of Computing, Teesside University.
11. J. B. Jensen and L. Birkedal. Fictional separation logic. In *ESOP*, pages 377–396, 2012.
12. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5:596–619, 1983.
13. R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and Invariants As an Orthogonal Basis for Concurrent Reasoning. In *ACM POPL*, pages 637–650, 2015.
14. L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.
15. K. R. Leino, P. Müller, and J. Smans. Verification of Concurrent Programs with Chalice. In *Foundations of Security Analysis and Design V*, pages 195–222, 2009.
16. R. Ley-Wild and A. Nanevski. Subjective Auxiliary State for Coarse-grained Concurrency. In *ACM POPL*, pages 561–574, 2013.
17. J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *ACM POPL*, pages 378–391, 2005.
18. A. Nanevski, R. Ley-Wild, I. Sergey, and G. Delbianco. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP*, pages 290–310. 2014.
19. P. W. O'Hearn. Resources, Concurrency, and Local Reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, April 2007.
20. K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. In *ESOP*, pages 149–168. 2014.
21. J. Tassarotti, D. Dreyer, and V. Vafeiadis. Verifying Read-Copy-Update in a Logic for Weak Memory. In *ACM PLDI*, Portland, OR, USA, 2015.
22. A. Turon, D. Dreyer, and L. Birkedal. Unifying Refinement and Hoare-style Reasoning in a Logic for Higher-order Concurrency. In *ICFP*, pages 377–390, 2013.
23. A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. In *ACM OOPSLA*, pages 691–707, 2014.
24. V. Vafeiadis and C. Narayan. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In *ACM OOPSLA*, pages 867–884, 2013.
25. V. Vafeiadis and M. J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *18th International Conference on Concurrency Theory (CONCUR'07)*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271, 2007.