Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Applying and Extending the Weak-Memory Logic FSL++

Research in Computer Science Project

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

Author:
Gaurav Parthasarathy

Supervisors:
Dr. Alexander J. Summers
Prof. Dr. Peter Müller

October 19, 2017

Contents

1 Introduction

Enforcing a sequentially consistent memory model on modern hardware is expensive in terms of performance. As a result, various programming languages define weak-memory models which provide weaker guarantees than sequentially consistent models. This enables compilers to generate more efficient code for the underlying hardware.

Reasoning about weak-memory programs is challenging and therefore there is a need to develop weak-memory logics which can be used to prove such programs correct. FSL++ [9] is one such weak-memory logic for the C11 weak-memory model, which is the memory model defined in the 2011 C++ Standard [1]. A subset of the logic has been encoded in Viper [11, 14], a verification infrastructure developed by the Chair of Programming Methodology at ETH Zurich.

An important feature of FSL++ is *ghost state*, which is auxiliary state that is not part of the actual program and is only used for verification purposes. Ghost state is specified using a set of ghost locations. The set of resources which can be carried by a particular ghost location must form a *partial commutative monoid* (PCM). Using ghost state and choosing a fitting PCM for each ghost location is essential for the verification of non-trivial programs.

In this project we explore FSL++ by applying it to real-world examples. For these real-world examples we provide proofs using known PCMs. Furthermore, we propose a novel PCM which allows for more direct proofs. We also introduce an extension to FSL++, which makes the logic more expressive.

This report is structured as follows. Section 2 introduces FSL++ as well as other background material for the remaining sections. In Section 3 we present the real-world examples that we consider throughout the report along with FSL++ proofs using known PCMs. In Section 4 we propose an extension to FSL++ which can be applied to the presented examples. We introduce a novel PCM in Section 5 and in Section 6 we provide more direct proofs for the real-world examples compared to Section 3 using this PCM. Finally, we conclude and provide ideas for future work in Section 7.

2 Background

In this section we present FSL++. We also introduce basic concepts in separation logic and discuss common permission structures.

2.1 Separation logic and fractional permissions

Separation logic. Separation logic [12] is a permission-based program logic which extends Hoare logic [10]. It enables simple reasoning about properties of heap-based programs by only considering those heap locations which are relevant for the properties to be proved.

The formalization of separation logic relies on the notion of partial heaps. A partial heap is characterized by a partial function which maps pairs of references and fields (i.e. heap locations) to their corresponding values in the heap. Two states σ_1, σ_2 are called *compatible* in separation logic if the domains of their heaps are disjoint (i.e. the domains of their corresponding partial functions are disjoint) and they agree on the values of all local variables. For compatible states σ_1 and σ_2 we can define the union of σ_1 and σ_2 to be the state with the same local variables as σ_1 and σ_2 and whose heap is the (disjoint) union of the heaps of σ_1 and σ_2.

One of the main connectives introduced by separation logic is the *separating conjunction* $*$. A state σ satisfies $A * B$ (for assertions A, B) iff σ is the union of compatible states σ_1 and σ_2 where σ_1 satisfies A and σ_2 satisfies B.

An important assertion in separation logic is the *points-to assertion* $x.f \mapsto v$ where $x.f$ denotes some heap location and v is some value. A state σ satisfies $x.f \mapsto v$ if the domain of the partial function σ_h representing σ's heap only contains $x.f$ and $\sigma_h(x.f) = v$. To access or write to a heap location $x.f$ the assertion $x.f \mapsto v$ must be guaranteed (for any value v). If $x.f \mapsto v$ holds in some context then this can be interpreted as that context owning the heap location $x.f$. The assertion emp holds in a state σ iff the domain of the heap in σ is empty.

Fractional permissions. Fractional permissions [6] generalize the points-to assertion by annotating each such assertion with a fractional value $q \in \mathbb{Q} \cap [0, 1]$ and introduce the following equivalence:

$$x.f \overset{q_1}{\mapsto} v * x.f \overset{q_2}{\mapsto} v \Leftrightarrow \begin{cases} x.f \overset{q_1+q_2}{\mapsto} v & \text{if } q_1 + q_2 \leq 1 \\ \mathsf{false} & \text{otherwise} \end{cases}$$

The generalized points-to assertion $x.f \overset{q}{\mapsto} v$ for $q < 1$ can be interpreted as representing partial ownership of the heap location v. If $q = 1$ then it can be interpreted as full ownership. To read a location $x.f$ it suffices to guarantee $x.f \overset{q}{\mapsto} v$ for any $q \in \mathbb{Q} \cap (0, 1]$ and to write to a location $x.f$ the full ownership $x.f \overset{1}{\mapsto} v$ is required. Note that the separating conjunction must be adjusted to allow for these modified points-to assertions by associating a permission amount with each heap location in a state. This adjustment enables compatible states to have access to common heap locations as long

as the corresponding permission values do not add up to more than 1 and the common heap locations map to the same values.

2.2 Abstract predicates

Abstract predicates [13] can be used to express recursive specifications which potentially represent an unbounded number of heap locations and also allow abstracting over concrete assertions. For example, ownership to a linked list could be expressed using the abstract predicate $\mathit{list}(h)$ where h is the reference to the head of the list. $\mathit{list}(h)$ provides ownership to all heap locations of the list elements. One possible way to give a definition of $\mathit{list}(h)$ is the following:

$$\mathit{list}(h) \equiv \exists v.\, h.\mathit{data} \overset{1}{\mapsto} v * h.\mathit{next} \overset{1}{\mapsto} \_ * h.\mathit{next} \neq \mathit{null} \Rightarrow \mathit{list}(h.\mathit{next})$$

where $h.\mathit{data}$ holds the value of the element at h and $h.\mathit{next}$ points to the next element (or null if it is the last element in the list).

Just as with the points-to assertions we can generalize abstract predicates by annotating them with a fraction $\mathbb{Q}_{\geq 0}$. For an abstract predicate $p(x)$ we write $p(x)^q$ to annotate $p(x)$ with fraction q. We illustrate the meaning by example. If the above definition of $\mathit{list}(h)$ is used then $\mathit{list}(h)^{\frac{1}{2}}$ is equivalent to:

$$\exists v.\, h.\mathit{data} \overset{\frac{1}{2}}{\mapsto} v * h.\mathit{next} \overset{\frac{1}{2}}{\mapsto} \_ * h.\mathit{next} \neq \mathit{null} \Rightarrow \mathit{list}(h.\mathit{next})^{\frac{1}{2}}$$

All fractions which are part of the points-to assertions and abstract predicates in the definition of the predicate are multiplied by the fraction associated with the predicate itself[1]. This enables one to specify that multiple threads have reader access to the list (by giving, for example, one thread $\mathit{list}(h)^{\frac{1}{2}}$ and the other thread $\mathit{list}(h)^{\frac{1}{2}}$) while keeping all the advantages of abstract predicates.

The following equivalence holds for abstract predicate instances $p(h)$:

$$p(h)^{q_1} * p(h)^{q_2} \Leftrightarrow \begin{cases} p(h)^{q_1+q_2} & \text{if } p(h)^{q_1+q_2} \text{ is defined} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

2.3 FSL++

FSL++ [9] is a program logic for reasoning about memory accesses in the C11 memory model. It is an extension of fenced separation logic [8] (FSL), which

[1]Contrary to the points-to assertion the fraction q associated with an abstract predicate can be greater than 1. This is still consistent in cases where multiplying q with all the fractions associated with points-to assertions and abstract predicates in the definition does not yield points-to assertions associated with fractions that are greater than 1.

Memory Operation	`rel`	`acq`	`rel_acq`	`rlx`
Write	x			x
Read		x		x
Update	x	x	x	x

Table 1: For each memory operation the valid access types are marked for atomic locations.

itself extends relaxed separation logic [16] (RSL). It is unsound with respect to the original C11 memory model but it is sound with respect to a slightly stronger model (details are given in [9]). In this section we present the main parts of FSL++ which are required to understand the following sections. Note that we differentiate between statements and expressions while in [9] only expressions are used.

In the C11 memory model memory locations are classified either as non-atomic or as atomic. Data races are not permitted on non-atomic locations, but they are permitted on atomic locations. The C11 model provides different access types for memory operations on atomic locations (we consider writes, reads and updates). The access types are *release* (`rel`), *acquire* (`acq`), *release-acquired* (`rel_acq`) and *relaxed* (`rlx`). The different access types provide different guarantees. Table 1 shows which access types can be used for which memory operations.

2.3.1 Non-atomic locations

The rules for non-atomic locations in FSL++ are similar to those for standard heap accesses in concurrent separation logic. Let l be a non-atomic location.

Non-atomic allocations

$$\overline{\{\mathsf{emp}\}l := \texttt{alloc}()\{\mathsf{Uninit}(l)\}}$$

The $\mathsf{Uninit}(l)$ predicate represents ownership of a non-atomic location that has not been initialized yet.

Non-atomic write
Writing to a non-atomic location either requires the full permission to location l (which implies that l has already been initialized) or ownership of l via the $\mathsf{Uninit}(l)$ predicate. The rule is given by

$$\overline{\{l \overset{1}{\mapsto} \_ \vee \mathsf{Uninit}(l)\}[l]_{\texttt{na}} := v\{l \overset{1}{\mapsto} v\}}$$

Non-atomic read
To read from a non-atomic location l partial permission to l is required.

$$\overline{\{l \overset{k}{\mapsto} v \wedge k > 0\} x := [l]_{\mathtt{na}} \{x = v \wedge l \overset{k}{\mapsto} v \wedge k > 0\}}$$

The rules for non-atomic writes and reads ensure data race freedom.

2.3.2 Atomic locations

FSL++ reasons about atomic locations using location invariants. A location invariant $\mathcal{Q}$ is a function from values to assertions. For each atomic location a location invariant must be picked and all the memory operations on a particular atomic location are verified in FSL++ with respect to the selected invariant. Suppose location invariant $\mathcal{Q}$ is selected for atomic location l. This roughly means that whenever l holds value v then $\mathcal{Q}(v)$ holds at location l, i.e. the location in a sense owns the resources described by $\mathcal{Q}(v)$, as long as no thread has acquired ownership of $\mathcal{Q}(v)$ from the location. As a result, $\mathcal{Q}$ specifies the resources that must be given up for each value that is written and the resources that are acquired for each value that is read (as long as those resources were not already acquired by a different read).

Release writes
For a *release write* of value v to location l FSL++ requires that $\mathcal{Q}(v)$ is given up, where $\mathcal{Q}$ is the location invariant associated with l. This is reflected by the following rule:

$$\overline{\{\mathsf{Rel}(l, \mathcal{Q}) * \mathcal{Q}(v)\}[l]_{\mathtt{rel}} := v\{\mathsf{Init}(l)\}}$$

The intuition for the predicate $\mathsf{Rel}(l, \mathcal{Q})$ is that the logic needs to ensure that the correct location invariant is used for l (namely the one that was picked initially). Since the location invariant is fixed for each location, $\mathsf{Rel}(l, \mathcal{Q})$ is duplicable, i.e. it holds that

$$\mathsf{Rel}(l, \mathcal{Q}) \Leftrightarrow \mathsf{Rel}(l, \mathcal{Q}) * \mathsf{Rel}(l, \mathcal{Q})$$

$\mathsf{Init}(l)$ asserts that l is initialized.

Relaxed writes
Relaxed writes provide fewer guarantees than release writes in the C11 model. In particular, just giving up $\mathcal{Q}(v)$ for a relaxed write of v is not enough to

preserve the meaning of the location invariant in FSL++. Instead, once $\mathcal{Q}(v)$ is held, a *release fence* instruction must be executed which transforms $\mathcal{Q}(v)$ to $\triangle\mathcal{Q}(v)$, where $\triangle$ is a modality introduced in FSL++. This transformation is reflected by the following rule:

$$\overline{\{P\}\texttt{fence}_{\texttt{rel}}\{\triangle P\}}$$

For the relaxed write $\triangle\mathcal{Q}(v)$ must be given up:

$$\overline{\{\mathsf{Rel}(l,\mathcal{Q}) * \triangle\mathcal{Q}(v)\}[l]_{\texttt{rlx}} := v\{\mathsf{Init}(l)\}}$$

Hence one can informally state that the release fence instruction gives strong enough guarantees such that the resources, which were held right before the fence, can be transferred to the location.

Acquire reads
After an *acquire read* of value v in FSL++ the resources $\mathcal{Q}(v)$ are acquired, as long as the same thread did not already acquire those resources. FSL++ ensures that for each write there is at most one read which acquires the resources provided by that write. We do not give the FSL++ rules for acquire reads since they do not show up in the following sections.

Relaxed reads
After a *relaxed read* of value v the resources $\triangledown\mathcal{Q}(v)$ are acquired, as long as the same thread did not already acquire those resources. $\triangledown$ is a modality introduced by FSL++ and the intuition for it is that a relaxed read does not provide enough guarantees for the thread to directly use the resources that are acquired. Instead, the resources acquired must be first transformed using an *acquire fence* instruction which is reflected by the following rule:

$$\overline{\{\triangledown P\}\texttt{fence}_{\texttt{acq}}\{P\}}$$

FSL++ ensures that for each (release or relaxed) write there is at most one (acquire or relaxed) read which acquires the resources provided by that write. It is possible that one thread acquires one part of $\mathcal{Q}(v)$ by a (release or relaxed) read and another thread acquires another disjoint part of $\mathcal{Q}(v)$ by a (release or relaxed) read, where both read actions read the value v from the same write (details are given in [9]). We do not provide the rule for relaxed reads since they do not show up in the following sections.

Updates
Update actions are performed using specific read-modify-write operations.

$$\frac{\begin{array}{ll} x \notin FV(P) & P' \equiv \begin{cases} P & \text{if } \tau \in \{\texttt{rel}, \texttt{rel\_acq}\} \\ \triangle P & otherwise \end{cases} \\ \mathcal{Q}(v) \models A * T & \\ P * T \models \mathcal{Q}(v') & A' \equiv \begin{cases} A & \text{if } \tau \in \{\texttt{acq}, \texttt{rel\_acq}\} \\ \triangledown A & otherwise \end{cases} \\ P * \mathcal{Q}(v) \models \varphi & \\ & pure(\varphi) \end{array}}{\left\{ \begin{array}{l} \mathsf{Init}(l) * \mathsf{Rel}(l, \mathcal{Q}) * \\ \mathsf{RMWAcq}(l, \mathcal{Q}) * P' \end{array} \right\} x := \mathtt{CAS}_\tau(l, v, v') \left\{ \left(\begin{array}{l} x = v \quad ? \\ A' \wedge \varphi \quad : \\ \mathsf{Init}(l) * \mathsf{Rel}(l, \mathcal{Q}) * \\ \mathsf{RMWAcq}(l, \mathcal{Q}) * P' \end{array} \right) \right\}}$$

Figure 1: CAS−basic rule

One particular read-modify-write operation is the *compare-and-swap* (CAS) operation. Execution of $x := \mathtt{CAS}(l, v, v')$ reads from location l, if the value read is given by v then v' is written to l (in this case we call the CAS *successful*) and otherwise nothing is written (in this case we call the CAS *failed*). The value that was read is stored into x. The whole CAS operation is atomic.

One possible rule for the CAS in FSL++ is given in Figure 1. We call this rule CAS−*basic*. It is almost identical to the CAS rule given in [15]. The main difference is that in the presented rule the predicates $\mathsf{Init}(l), \mathsf{Rel}(l, \mathcal{Q}), \mathsf{RMWAcq}(l, \mathcal{Q})$ are only retained if the CAS has failed while in [15] the predicates are also retained if the CAS is successful[2]. This is not a big difference since these predicates are duplicable. The reason we present the rule like this is that the CAS rules provided in the original FSL++ paper given in [9] also follow this convention. Our presented rule has been proved sound [7]. The presentation of this rule is almost the same as in [14] (the φ part was added). The presented CAS rule summarizes all four possible access types.

Let us consider the `rel_acq` access type to understand the CAS rule better. The precondition of the conclusion includes the $\mathsf{RMWAcq}(l, \mathcal{Q})$ predicate and the already seen $\mathsf{Rel}(l, \mathcal{Q})$ and $\mathsf{Init}(l, \mathcal{Q})$ predicates. Additionally, the precondition includes an assertion P which is the part of the local state which the thread gives up for the CAS operation. We sometimes use $\mathsf{U}(l, \mathcal{Q})$ where

$$\mathsf{U}(l, \mathcal{Q}) := \mathsf{Init}(l, \mathcal{Q}) * \mathsf{Rel}(l, \mathcal{Q}) * \mathsf{RMWAcq}(l, \mathcal{Q})$$

[2]In [15] only a single predicate $\mathsf{U}(l, \mathcal{Q})$ is used but this is conceptually the same as the three predicates that we use.

$$\frac{\begin{array}{l} x \notin FV(P) \\ \mathcal{Q}(v) \models \exists z.\, \mathcal{A}(z) * \mathcal{T}(z) \\ \forall z. P * \mathcal{T}(z) \models \mathcal{Q}(v') \wedge \varphi(z) \\ \forall z. pure(\varphi(z)) \end{array} \qquad \begin{array}{l} P' \equiv \begin{cases} P & \text{if } \tau \in \{\texttt{rel}, \texttt{rel\_acq}\} \\ \triangle P & otherwise \end{cases} \\ \forall z. \mathcal{A}'(z) \equiv \begin{cases} \mathcal{A}(z) & \text{if } \tau \in \left\{ \begin{array}{l} \texttt{acq}, \\ \texttt{rel\_acq} \end{array} \right\} \\ \nabla \mathcal{A}(z) & otherwise \end{cases} \end{array}}{\left\{ \begin{array}{l} \mathsf{Init}(l) * \mathsf{Rel}(l, \mathcal{Q}) * \\ \mathsf{RMWAcq}(l, \mathcal{Q}) * P' \end{array} \right\} x := \mathtt{CAS}_\tau(l, v, v') \left\{ \left(\begin{array}{l} x = v \ ? \\ \quad \exists z.\, \mathcal{A}'(z) \wedge \varphi(z) \ : \\ \quad \mathsf{Init}(l) * \mathsf{Rel}(l, \mathcal{Q}) * \\ \quad \mathsf{RMWAcq}(l, \mathcal{Q}) * P' \end{array} \right) \right\}}$$

Figure 2: CAS−param rule

for readability reasons. $\mathsf{U}(l, \mathcal{Q})$ is duplicable, i.e. it holds that

$$\mathsf{U}(l, \mathcal{Q}) \Leftrightarrow \mathsf{U}(l, \mathcal{Q}) * \mathsf{U}(l, \mathcal{Q})$$

The postcondition of the conclusion guarantees A if the CAS operation is successful and otherwise no permission is transferred and P is retained.

Let us now consider the premise. In a successful CAS operation the value v is read and the value v' is written. To make sure the location invariant holds after v' is written $\mathcal{Q}(v')$ must be guaranteed. This must be achieved by using parts of $\mathcal{Q}(v)$ (since v is read) and P (since it is given up by the thread performing the CAS). In particular, the premise requires to extract a part T of $\mathcal{Q}(v)$ which is used to establish $Q(v')$. The remaining part A of $\mathcal{Q}(v)$ is given to the thread.

Finally, any pure assertion φ may be learnt by the thread performing the CAS operation by considering $P * \mathcal{Q}(v)$ in a successful CAS operation. An assertion is pure if it is independent of the heap.

For the `rlx` access type both A and P appear under modalities in the conclusion (it can be roughly interpreted as a relaxed read combined with a relaxed write). For the `acq` access type only P appears under a modality (it can be roughly interpreted as an acquire read combined with a relaxed write) and for the `rel` access type only A appears under a modality (it can be roughly interpreted as a relaxed read combined with a release write).

A different CAS rule which is taken from [9] is given in Figure 2. We call this the CAS−*param* rule. It is more flexible compared to the CAS−*basic* rule. Instead of splitting $\mathcal{Q}(v)$ into assertions A and T, $\mathcal{Q}(v)$ is split into $\mathcal{A}(z)$ and $\mathcal{T}(z)$ for some value z, where $\mathcal{A}$ and $\mathcal{T}$ are functions from values to assertions. Furthermore, φ is a function from values to pure assertions. This rule allows for easier proofs in certain cases as we will see in later sections.

$$\frac{\begin{array}{l}\mathcal{Q}(v) * P \Rightarrow \mathsf{false} \\ \tau \in \{\mathtt{rlx}, \mathtt{rel}, \mathtt{acq}, \mathtt{rel\_acq}\}\end{array}}{\left\{ \begin{array}{l} \mathsf{Init}(l) * \mathsf{Rel}(l, \mathcal{Q}) * \\ \mathsf{RMWAcq}(l, \mathcal{Q}) * P \end{array} \right\} x := \mathtt{CAS}_\tau(l, v, v') \left\{ \begin{array}{l} (x \neq v) \wedge \mathsf{Init}(l) * \mathsf{Rel}(l, \mathcal{Q}) * \\ \mathsf{RMWAcq}(l, \mathcal{Q}) * P \end{array} \right\}}$$

Figure 3: FSL++ CAS$-\bot$ rule which directly can be used to show that the CAS will not be successful.

$$\frac{\forall t. \left(\begin{array}{l} \{\mathsf{U}(l, \mathcal{Q}) * \mathcal{P}_{\mathsf{send}}(t)\} \\ z := \mathtt{CAS}_\tau(l, t, t+v) \\ \left\{ \begin{array}{l} (z = t \wedge \mathcal{R}(t)) \vee \\ (z \neq t \wedge \mathsf{U}(l, \mathcal{Q}) * \mathcal{P}_{\mathsf{send}}(t)) \end{array} \right\} \end{array} \right) \quad \begin{array}{l} \forall t. (P \Leftrightarrow \mathcal{P}_{\mathsf{send}}(t) * \mathcal{P}_{\mathsf{keep}}(t)) \\ \tau \in \{\mathtt{rlx}, \mathtt{rel}, \mathtt{acq}, \mathtt{rel\_acq}\} \\ \{x, z\} \cap FV(P) = \emptyset \end{array}}{\left\{ \; \mathsf{U}(l, \mathcal{Q}) * P \; \right\} x := \mathtt{fetch\_and\_add}_\tau(l, v) \left\{ \; \mathcal{R}(x) * \mathcal{P}_{\mathsf{keep}}(x) \; \right\}}$$

Figure 4: FSL++ fetch-and-add rule. $\mathsf{U}(l, \mathcal{Q})$ stands for $\mathsf{Init}(l) * \mathsf{Rel}(l, \mathcal{Q}) * \mathsf{RMWAcq}(l, \mathcal{Q})$.

If a thread owns resources that are incompatible with $\mathcal{Q}(v)$ then there definitely cannot be a successful CAS where v is read. This is captured by the CAS$-\bot$ rule in Figure 3.

Note that in FSL++ acquire reads and relaxed reads cannot acquire resources from a location invariant governing an atomic location that supports updates. The intuition behind this is that if there are only updates and writes which can transfer resources then automatically every read in a successful update operation reads from a write for which the corresponding resources have not been acquired yet. This justifies why $\mathcal{Q}(v)$ can always be used when value v is read in a successful CAS.

From the CAS rule a fetch-and-add rule can be derived (since the fetch-and-add operation can be implemented using the CAS operation[3]). It is given in Figure 4. The main difference between a fetch-and-add and the CAS operation is that the fetch-and-add operation is always successful (and it only allows incrementing the value at the atomic location but that can be adjusted).

Note that in the premise of the fetch-and-add rule, a CAS triple must be

[3]In practice the fetch-and-add operation is implemented more efficiently than by using the CAS operation naively, but this has no impact on how one reasons about the fetch-and-add since the effect of the fetch-and-add remains the same.

verified for every possible read value t. There may be cases where a value t cannot be read: in such cases one often can verify the CAS triple directly using the CAS$-\bot$ rule.

We also note that in some of the presented inference rules certain symbols are interpreted as actual values even though they should be interpreted as program expressions. Interpreting the symbols as values made the presentation easier. For example, the conclusion of the CAS$-$*basic* rule is given by

$$\left\{ \begin{array}{l} \mathsf{Init}(l) * \mathsf{Rel}(l, \mathcal{Q}) * \\ \mathsf{RMWAcq}(l, \mathcal{Q}) * P' \end{array} \right\} x := \mathtt{CAS}_\tau(l, v, v') \left\{ \left(\begin{array}{l} x = v \quad ? \\ A' \wedge \varphi \quad : \\ \mathsf{Init}(l) * \mathsf{Rel}(l, \mathcal{Q}) * \\ \mathsf{RMWAcq}(l, \mathcal{Q}) * P' \end{array} \right) \right\}$$

where we interpret v and v' as values. In practice v and v' are program expressions that have the type defined by the atomic location l. We assume a generalization of the rules where the CAS takes program expressions as arguments which are evaluated in the program state right before the CAS is executed. For the other rules we assume an analogous generalization.

Allocation of atomic locations
The rule for the allocation of atomics for which update instructions can be applied to is given by

$$\overline{\{\mathsf{emp}\} l := \mathtt{alloc}() \{l.\mathsf{Rel}(l, \mathcal{Q}) * \mathsf{RMWAcq}(l, \mathcal{Q})\}}$$

2.3.3 Ghost locations

FSL++ supports ghost state. Ghost state is state that can be used to reason about a program, but which is not part of the actual program state. It is often used to keep track of facts that cannot be expressed using just program state. In FSL++ ghost state is accessed through *ghost locations* (analogous to memory locations) which carry *ghost resources* (analogous to memory locations that are associated with permission values). The assertion $\boxed{\gamma : g}$ states that the ghost location γ carries the ghost resource g. Ghost resources on a single location have to form a *partial commutative monoid* (PCM). See Appendix A for the definition of partial commutative monoids.

Ghost state can be introduced at any point, as is reflected by the following rule:

$$\frac{\{P\} C \{Q\}}{\{P\} C \{Q * \exists \gamma. \boxed{\gamma : g}\}}$$

Let γ be a ghost location with ghost resources that form a partial commutative monoid with composition operation $\oplus$. The following holds:

$$\boxed{\gamma : g_1} * \boxed{\gamma : g_2} \Leftrightarrow \begin{cases} \boxed{\gamma : g_1 \oplus g_2} & \text{if } g_1 \oplus g_2 \text{ is defined} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

Ghost state has the important property that there is no difference between ghost state with and without the modalities $\triangle$, $\triangledown$:

$$\boxed{\gamma : g} \Leftrightarrow \triangle \boxed{\gamma : g} \Leftrightarrow \triangledown \boxed{\gamma : g}$$

A more detailed explanation of the modalities than the one in Section 2.3.2 is given in [9].

2.4 Permission structures

Permission structures are structures that can be used to reason about ownership of non-atomic locations. For example, the fractional permission model introduced in Section 2.1 corresponds to a particular permission structure.

Generally, permission structures[4] are algebraic structures of the form $(S, \oplus, \mu, \mathbb{1})$, where $(S, \oplus)$ forms a partial commutative monoid with neutral element μ (empty permission) and $\mathbb{1} \in S \setminus \{\mu\}$ is a maximal element (full permission) which means that $\mathbb{1} \oplus s$ is undefined for every $s \in S \setminus \{\mu\}$.

In the following we present three known permission structures. In later sections in FSL++ proofs some ghost locations will carry ghost resources (see Section 2.3.3) that form a PCM corresponding to one of these permission structures (since permission structures form a PCM, we can do this).

Fractional permission structure. The fractional permission structure directly corresponds to fractional permissions as introduced in Section 2.1. It is given by the algebraic structure $(\mathbb{Q} \cap [0, 1], \oplus, 0, 1)$ where $\oplus$ is defined as:

$$q_1 \oplus q_2 := \begin{cases} q_1 + q_2 & \text{if } q_1 + q_2 \leq 1 \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$

Counting permission structure. The counting permission structure was first introduced in [5]. It is a permission structure that defines a *source permission* that counts how many *access permissions* were given away. It can be described by the algebraic structure $(\mathbb{N} \times \{+, -\}, \oplus, 0^+, 0^-)$. We use

[4]The presentation of the formal definition is taken from [9].

the notation a^+ (i.e. a^-) for an element $(a, +) \in \mathbb{N} \times \{+, -\}$ (i.e. $(a, -)$). $\oplus$ is defined as:

$$a^+ \oplus b^+ := (a + b)^+$$
$$a^- \oplus b^- := \mathsf{undefined}$$
$$a^+ \oplus b^- := b^- \oplus a^+ := \begin{cases} (b - a)^- & \text{if } b - a \geq 0 \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$

a^- is the source permission and there can only be one such permission. As a sidenote: in [5] this structure is introduced without a neutral element (i.e. 0^+ is not an element of the set), which is fine in their context since they only require the structure to be a partial commutative semigroup.

***SFC* permission structure.** In [5] a structure combining the fractional and counting permission structure is introduced. The only difference to the counting permission structure is that the set is $\mathbb{Q}_{\geq 0} \times \{+, -\}$ instead of $\mathbb{N} \times \{+, -\}$. $\oplus$ is defined the same (as with the counting permission structure this structure is presented as a partial commutative semigroup in [5]). We call this structure the simple fractional-counting permission structure and we refer to is as the *SFC* permission structure.

Encoding counting permissions using fractional permissions. Instead of using the counting permission structure in the proofs we will encode the counting permission structure using the fractional permission structure since the main permission structure that we use corresponds to the fractional permission structure (see Section 2.1).

This can be done by introducing a fraction $\epsilon \in \mathbb{Q} \cap (0, 1)$ that is indivisible and small enough to work in all contexts, i.e. ϵ in the fractional permission structure models 1^+ in the counting permission model and a^- is modelled by $1 - a \cdot \epsilon$ (which is guaranteed to be greater than 0 since ϵ is chosen small enough). It is not clear how one would formally model ϵ such that it works in all contexts and we do not elaborate on this since all the proofs should work by just directly using the counting permission structure.

3 Main examples

In this section we introduce the three examples that we revisit in later sections. The first two examples are from the Facebook Folly library[5] and we present FSL++ proofs for both examples. To the best of our knowledge, these are the first FSL++ proofs for these examples. The third example is from the Rust standard library and we present the proof given in [9].

[5]The library is open source and can be found at `https://github.com/facebook/folly`.

```
bool try_lock_shared() {
  v0 := fetch_and_add_acq(bits, 4) //RMW1

  if(lsb(v0) == 1) {
    v1 := fetch_and_add_rel(bits, −4) //RMW2
    res := false
  } else {
    res := true
  }

  return res
}

void unlock_shared() {
  x := fetch_and_add_rel(bits, −4)
}

bool try_lock() {
  v := CAS_rel_acq(bits, 0, 1)
  return (v == 0)
}

void unlock(bool getRead) {
  x := fetch_and_and_rel(bits, ∼1)
}

void unlock_and_lock_shared() {
   x := fetch_and_add_acq(bits, 4) //RMW1
   unlock(true)
}
```

Figure 5: Pseudocode for the functions in RWSpin

3.1 Folly reader-writer spinlock

The Folly reader-writer spinlock [3] (we will refer to it as *RWSpin*) is a reader-writer lock which uses a single integer atomic location `bits`. It allows multiple readers to have access to the lock simultaneously while there is no writer and it allows a single writer to have access to the lock while there are no readers.

3.1.1 Implementation

The implementation of the most interesting functions in *RWSpin* is given in Figure 5. The idea behind the implementation is as follows. The least significant bit of `bits` is 1 if and only if there is a writer. If there are r

readers then the value of `bits` is at least $4 \cdot r$ (but not necessarily exactly equal to $4 \cdot r$).

The implementation we consider is a slightly simplified one; in the original implementation there is also the notion of an upgradeable bit (the second least significant bit, which is why increments of 4 are used instead of increments of 2) which enables a thread that wants the writer lock to block more threads from getting a reader lock. Generalizing the proofs to include this notion and verifying functions which involve the upgradeable bit should not be hard.

In `try_lock_shared` the thread attempts to get a reader lock. This is achieved by first incrementing `bits` by 4 atomically and then checking if the value updated had 0 as least significant bit (i.e. there was no writer). If this is the case the thread gets the reader lock and otherwise the thread decrements `bits` by 4 to undo the former action.

In `unlock_shared` a reader lock is unlocked by decrementing `bits` by 4.

In `try_lock` a thread attempts to get a writer lock by setting the least significant bit to 1 if there is no writer and there are no readers.

In `unlock` a writer lock is unlocked by updating the value read v by $v\&{\sim}1$ where $\sim$ denotes bitwise complement (i.e. by setting the least significant bit to 0). Note that in the original implementation the value is set to $v\&{\sim}3$ since the upgradeable bit is also taken into account.

The boolean parameter `getRead` in the `unlock` function is a *ghost parameter*, i.e. it does not affect the implementation but we use it for the specification. We interpret it as follows in the specification, which we provide later. If `unlock(false)` is called then the caller just wants to give up the writer lock. If `unlock(true)` is called then the caller wants to give up the writer lock and additionally wants to get a reader lock. In the latter case the specification will require the caller to provide additional resources which makes it possible to extract a reader lock using the same `unlock` code as in the former case.

In `unlock_and_lock_shared` a writer thread first increments by 4, which conceptually reserves a reader lock. Then the thread unlocks the writer lock and gets a reader lock (since the writer incremented by 4 before unlocking).

3.1.2 Resource

RWSpin can be used as a lock for any kind of data structure. In our specification we use an abstract predicate (see Section 2.2) *resource*(`bits`) to represent the permission to the data structure that is guarded by the reader-writer lock implemented using the atomic integer location `bits`. We make the assumption that the full permission (i.e. the permission needed by a

writer) is given by *resource*(`bits`)[1]. For a reader, partial permission is sufficient, hence *resource*(`bits`)^q for any $q \in (0, 1]$ is fine. We do not make the assumption that *resource*(`bits`)$^{q'} \Rightarrow \mathsf{false}$ if $q' > 1$ (note that this does not hold in general for abstract predicates, see the abstract predicates section for more details). There are cases where this assumption would be justified and in such cases the verification would be slightly easier.

3.1.3 Location invariant

The location invariant that we use for the FSL++ proofs for the integer atomic location `bits` is given by:

$$\begin{aligned}\mathcal{Q}(v) := \;& \text{let } n = \left\lfloor \frac{v}{4} \right\rfloor, w = \mathsf{lsb}(v) \text{ in} \\ & \exists n_r \in \mathbb{N}.\, v \geq 0 \wedge n_r \leq n \wedge (w = 1 \Rightarrow n_r = 0) \wedge \\ & \textit{resource}(\texttt{bits})^{(w=1\ ?\ 0\ :\ 1-n_r\cdot\epsilon)} * \\ & \boxed{\alpha : 1 - w} * \boxed{\beta : 1 - n_r \cdot \epsilon} * \boxed{\gamma : 1 - (n - n_r) \cdot \epsilon}\end{aligned}$$

where α,β,γ are ghost locations which are governed by the fractional permission structure introduced in Section 2.4. The meaning of the ϵ fraction is also explained in Section 2.4 (we are modelling counting permissions using fractional permissions: see the corresponding section for more details). The permission to the data structure guarded by the reader-writer lock, as explained before, is inside the *resource*(`bits`) abstract predicate. $\mathsf{lsb}(v)$ returns the least significant bit of v, if it is 1 then there is a writer and otherwise not.

The idea of the location invariant is as follows. As explained before, in the `try_lock_shared` function a thread tries to get a reader lock by incrementing by 4 and the thread gets the reader lock iff at the time of the update there was no writer. These are the only increments by 4 that are performed. If the value at the location is given by v the number of increments by 4 (for which no corresponding decrement by 4 was executed) is given by $\left\lfloor \frac{v}{4} \right\rfloor$ which we define to be n (since the only other operations are decrements by 4 or toggling the least significant bit). Some of these increments by 4 were performed by threads that now have a reader lock: we call these threads the *real readers* (they have read access to the resource). The rest of these increments by 4 were performed by threads that do not have a reader lock (and have not decremented by 4 yet): we call these threads the *fake readers* (they do not have any access to the resource).

Note that it is possible for real readers and fake readers to coexist at the same time. For example, while a fake reader has not decremented by 4 yet the writer may unlock the lock and another thread may become a real reader.

The number of real readers is given by the existentially quantified variable n_r and the number of fake readers is given by $n - n_r$. If there is a writer ($w = 1$) then we know that there cannot be a real reader, hence $n_r = 0$. Furthermore, if there is a writer then the writer has the full permission to the resource, hence there is no permission to *resource*(`bits`) in the location invariant. If there is no writer then we make sure that each real reader gets ϵ permission to *resource*(`bits`), which means there is $1 - n_r \cdot \epsilon$ permission to *resource*(`bits`) in the location invariant. So ϵ allows us to explicitly count how much permission we have given away.

The α ghost location is required to make sure that when the writer unlocks it can be verified that the value read before updating must have 1 as its least significant bit (we will see this in detail later, but the main reason is that *resource*(`bits`)^q is not necessarily inconsistent if $q > 1$). The β,γ ghost locations are used to make sure that when decrementing by 4 it can be verified that the value read is at least 4, and to track the real/fake readers.

3.1.4 Specification

The specification of the methods is given by:

$$\{\mathsf{U}(\texttt{bits}, \mathcal{Q})\}\texttt{bool try\_lock\_shared()} \left\{ y.\ \begin{array}{l}(y\ ?\\ R_\beta(\texttt{bits}) :\\ \mathsf{emp})\end{array} \right\}$$

$$\{\mathsf{U}(\texttt{bits}, \mathcal{Q}) * R_\beta(\texttt{bits})\}\texttt{void unlock\_shared()}\{\mathsf{emp}\}$$

$$\{\mathsf{U}(\texttt{bits}, \mathcal{Q})\}\texttt{bool try\_lock()}\{y.(y\ ?\ W_\alpha(\texttt{bits}) : \mathsf{U}(\texttt{bits}, \mathcal{Q}))\}$$

$$\left\{ \begin{array}{l}\mathsf{U}(\texttt{bits}, \mathcal{Q}) *\\ W_\alpha(\texttt{bits}) *\\ (\texttt{getRead}\ ?\\ \boxed{\gamma : \epsilon} : \mathsf{emp})\end{array} \right\}\texttt{void unlock(bool getRead)} \left\{ \begin{array}{l}(\texttt{getRead}\ ?\\ R_\beta(\texttt{bits}) :\\ \mathsf{emp})\end{array} \right\}$$

$$\{\mathsf{U}(\texttt{bits}, \mathcal{Q}) * W_\alpha(\texttt{bits})\}\texttt{void unlock\_and\_lock\_shared()}\ \{R_\beta(\texttt{bits})\}$$

where

$$W_\alpha(\texttt{bits}) \equiv \mathit{resource}(\texttt{bits})^1 * \boxed{\alpha : 1}$$
$$R_\beta(\texttt{bits}) \equiv \mathit{resource}(\texttt{bits})^\epsilon * \boxed{\beta : \epsilon}$$

Note that in the specification of, for example, `unlock_shared` we do not retain the permission to $\mathsf{U}(\texttt{bits}, \mathcal{Q})$. But this is not an issue since $\mathsf{U}(\texttt{bits}, \mathcal{Q})$ is duplicable, so a caller of `lock` can still retain permission to it. Also in some cases when $\mathsf{U}(\texttt{bits}, \mathcal{Q})$ is needed after a fetch-and-add operation we do not explicitly mention it in the proof for the same reason.

$\{\mathsf{U}(\texttt{bits}, \mathcal{Q})\}$
$v0 := \texttt{fetch\_and\_add}_{\texttt{acq}}(\texttt{bits}, 4)$ `//first RMW`
$\left\{(\mathsf{lsb}(v0) = 0 \ ? \ R_\beta(\texttt{bits}) \ : \ \mathsf{U}(\texttt{bits}, \mathcal{Q}) * \boxed{\gamma : \epsilon})\right\}$
`if (`$\mathsf{lsb}(v0)$ `== 1) {`
    $\left\{\mathsf{U}(\texttt{bits}, \mathcal{Q}) * \boxed{\gamma : \epsilon}\right\}$
    $v1 := \texttt{fetch\_and\_add}_{\texttt{rel}}(\texttt{bits}, -4)$ `//second RMW`
    $\{\mathsf{emp}\}$
    `res := false`
`} else {`
    $\{R_\beta(\texttt{bits})\}$
    `res := true`
`}`
$\{(\texttt{res} \ ? \ R_\beta(\texttt{bits}) \ : \ \mathsf{emp})\}$
`return res`

Figure 6: Proof outline for the `try_lock_shared` *function*

In all the proofs we assume that the value read in the fetch-and-add operations is non-negative. The cases where the value read is negative are trivial since $\mathcal{Q}(t) \Rightarrow \mathsf{false}$ for $t < 0$, hence the CAS$-\bot$ rule can be used to prove the premise of the fetch-and-add rule (i.e. such a value can never be read).

3.1.5 Proof of function `try_lock_shared`

We show the proof for the function `try_lock_shared`. There are two fetch-and-add operations that must be considered. The proof outline is given in Figure 6.

First RMW. We first verify the first fetch-and-add operation. We need to show

$$\{\mathsf{U}(\texttt{bits}, \mathcal{Q})\}\, v0 := \texttt{fetch\_and\_add}_{\texttt{acq}}(\texttt{bits}, 4) \left\{ \begin{array}{l} (\mathsf{lsb}(v0) = 0 \ ? \\ R_\beta(\texttt{bits}) : \\ \mathsf{U}(\texttt{bits}, \mathcal{Q}) * \boxed{\gamma : \epsilon}) \end{array} \right\}$$

using the fetch-and-add rule in Figure 4. We have $P \equiv \mathsf{emp}$, hence

$$\mathcal{P}_{\mathsf{send}}(t) := \mathsf{emp}$$
$$\mathcal{P}_{\mathsf{keep}}(t) := \mathsf{emp}$$

Clearly $P \Leftrightarrow \mathcal{P}_{\mathsf{send}}(t) * \mathcal{P}_{\mathsf{keep}}(t)$ holds for any t. For the premise of the fetch-and-add rule we still need to verify the CAS triple for all possible values t that are read. We do this with the CAS$-$*basic* rule given in Figure 1. Let t

be the value read. First suppose $\mathsf{lsb}(t) = 1$ (i.e. there is a writer). We then have

$$\mathcal{Q}(t) \Leftrightarrow \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in } t \geq 0 \wedge \boxed{\alpha : 0} * \boxed{\beta : 1} * \boxed{\gamma : 1 - n \cdot \epsilon}$$

Choose

$$T := \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in } t \geq 0 \wedge \boxed{\alpha : 0} * \boxed{\beta : 1} * \boxed{\gamma : 1 - (n+1) \cdot \epsilon}$$
$$A := \boxed{\gamma : \epsilon}$$

Then we have $\mathcal{Q}(t) \Leftrightarrow A * T$ (A is the permission that is given to the thread while T stays in the location invariant). Note that

$$T \Rightarrow \text{let } n = \left\lfloor \frac{t+4}{4} \right\rfloor \text{ in } t + 4 \geq 0 \wedge \boxed{\alpha : 0} * \boxed{\beta : 1} * \boxed{\gamma : 1 - n \cdot \epsilon}$$
$$\Rightarrow \mathcal{Q}(t+4)$$

Hence we can establish the location invariant for $t+4$ (we used $\mathsf{lsb}(t) = \mathsf{lsb}(t+4)$). The thread gets $\boxed{\gamma : \epsilon}$. Note that we still need to retain $\mathsf{U}(\mathtt{bits}, \mathcal{Q})$, which we can do using the frame rule and using the fact that $\mathsf{U}(\mathtt{bits}, \mathcal{Q}) \Leftrightarrow \mathsf{U}(\mathtt{bits}, \mathcal{Q}) * \mathsf{U}(\mathtt{bits}, \mathcal{Q})$. We show this here and in the remaining proofs we assume that this is no issue. The following derivation is correct:

$$\frac{\{\mathsf{U}(\mathtt{bits}, \mathcal{Q})\}\, \mathtt{x} := \mathtt{CAS}_{\mathtt{acq}}(\mathtt{bits}, t, t-4) \left\{ \begin{array}{l} (x = t \ ? \\ \boxed{\gamma : \epsilon} : \mathsf{U}(\mathtt{bits}, \mathcal{Q})) \end{array} \right\}}{\left\{ \begin{array}{l} \mathsf{U}(\mathtt{bits}, \mathcal{Q}) * \\ \mathsf{U}(\mathtt{bits}, \mathcal{Q}) \end{array} \right\} \mathtt{x} := \mathtt{CAS}_{\mathtt{acq}}(\mathtt{bits}, t, t-4) \left\{ \begin{array}{l} \mathsf{U}(\mathtt{bits}, \mathcal{Q}) * \\ (x = t \ ? \\ \boxed{\gamma : \epsilon} : \mathsf{U}(\mathtt{bits}, \mathcal{Q})) \end{array} \right\}}$$
$$\frac{}{\{\mathsf{U}(\mathtt{bits}, \mathcal{Q})\}\, \mathtt{x} := \mathtt{CAS}_{\mathtt{acq}}(\mathtt{bits}, t, t-4) \left\{ \begin{array}{l} (x = t \ ? \\ \mathsf{U}(\mathtt{bits}, \mathcal{Q}) * \boxed{\gamma : \epsilon} : \\ \mathsf{U}(\mathtt{bits}, \mathcal{Q})) \end{array} \right\}}$$

This concludes the proof for t where $\mathsf{lsb}(t) = 1$. Next, suppose $\mathsf{lsb}(t) = 0$ (i.e. there is no writer). We then have

$$\begin{array}{l} \mathcal{Q}(t) \Rightarrow \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in} \\ \qquad \exists n'_r \in \mathbb{N}.\, t \geq 0 \wedge n'_r \leq n \wedge \mathit{resource}(\mathtt{bits})^{1 - n'_r \cdot \epsilon} * \\ \qquad \boxed{\alpha : 1} * \boxed{\beta : 1 - n'_r \cdot \epsilon} * \boxed{\gamma : 1 - (n - n'_r) \cdot \epsilon} \end{array}$$

Choose

$$\begin{aligned} T := \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in} \\ \exists n'_r \in \mathbb{N}.\, t \geq 0 \land n'_r \leq n \land \mathit{resource}(\texttt{bits})^{1-(n'_r+1)\cdot\epsilon} * \\ \boxed{\alpha : 1} * \boxed{\beta : 1 - (n'_r + 1) \cdot \epsilon} * \boxed{\gamma : 1 - (n - n'_r) \cdot \epsilon} \\ A := \mathit{resource}(\texttt{bits})^{\epsilon} * \boxed{\beta : \epsilon} \end{aligned}$$

Note that

$$\begin{aligned} T \Rightarrow & \text{let } n = \left\lfloor \frac{t+4}{4} \right\rfloor \text{ in} \\ & \exists n'_r \in \mathbb{N}.\, t \geq 0 \land n'_r + 1 \leq n \land \mathit{resource}(\texttt{bits})^{1-(n'_r+1)\cdot\epsilon} * \\ & \boxed{\alpha : 1} * \boxed{\beta : 1 - (n'_r + 1) \cdot \epsilon} * \boxed{\gamma : 1 - (n - (n'_r + 1)) \cdot \epsilon} \\ & \Rightarrow_{\text{choosing } n_r := n'_r + 1} \mathcal{Q}(t+4) \end{aligned}$$

Additionally we have that A is exactly the permission that we want (permission for a reader lock) and since the fetch-and-add is of access type `acq` we get A directly. This concludes the proof for the first fetch-and-add operation.

Second RMW. Next, we verify the second fetch-and-add operation. We need to show

$$\left\{ \mathsf{U}(\texttt{bits}, \mathcal{Q}) * \boxed{\gamma : \epsilon} \right\} v1 := \texttt{fetch\_and\_add}_{\texttt{rel}}(\texttt{bits}, -4) \, \{\mathsf{emp}\}$$

We have $P := \boxed{\gamma : \epsilon}$. We choose

$$\begin{aligned} \mathcal{P}_{\mathsf{send}}(t) &:= \boxed{\gamma : \epsilon} \\ \mathcal{P}_{\mathsf{keep}}(t) &:= \mathsf{emp} \end{aligned}$$

Clearly $P \Leftrightarrow \mathcal{P}_{\mathsf{send}}(t) * \mathcal{P}_{\mathsf{keep}}(t)$ holds for any t. We still need to show the CAS triple in the premise of the fetch-and-add rule for all values t that are read.

Assume $t < 4$: we show that such a value cannot be read. We have $\mathcal{Q}(t) \Rightarrow \boxed{\gamma : 1} * \mathsf{true}$. Hence $\mathcal{P}_{\mathsf{send}}(t) * \mathcal{Q}(t) \Rightarrow \mathsf{false}$, therefore we may use the CAS$-\bot$ rule to verify the CAS triple.

Next, assume $t \geq 4$. We prove the CAS triple in this case using the standard CAS$-$*basic* rule. Choose

$$\begin{aligned} T &:= \mathcal{Q}(t) \\ A &:= \mathsf{emp} \end{aligned}$$

Then clearly $Q(t) \Leftrightarrow A * T$. We have

$$\begin{aligned}
\mathcal{P}_{\mathsf{send}}(t) * T \Rightarrow \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor, w = \mathsf{lsb}(t) \text{ in} \\
\exists n_r \in \mathbb{N}.\, t - 4 \geq 0 \land n_r \leq n \land (w = 1 \Rightarrow n_r = 0) \land \\
\mathit{resource}(\texttt{bits})^{(w=1\ ?\ 0\ :\ 1-n_r\cdot\epsilon)} * \boxed{\alpha : 1 - w} * \\
\boxed{\beta : 1 - n_r \cdot \epsilon} * \boxed{\gamma : 1 - (n - n_r - 1) \cdot \epsilon} \\
\Rightarrow \text{let } n = \left\lfloor \frac{t-4}{4} \right\rfloor, w = \mathsf{lsb}(t) \text{ in} \\
\exists n_r \in \mathbb{N}.\, t - 4 \geq 0 \land n_r \leq n \land (w = 1 \Rightarrow n_r = 0) \land \\
\mathit{resource}(\texttt{bits})^{(w=1\ ?\ 0\ :\ 1-n_r\cdot\epsilon)} * \boxed{\alpha : 1 - w} * \\
\boxed{\beta : 1 - n_r \cdot \epsilon} * \boxed{\gamma : 1 - (n - n_r) \cdot \epsilon} \\
\Rightarrow \mathcal{Q}(t - 4)
\end{aligned}$$

Note that $t - 4 \geq 0$ holds by assumption. This concludes the proof of the second fetch-and-add operation. Note that since we only transfer ghost resources, the modalities in the fetch-and-add rule do not matter, hence it would even be possible to verify this fetch-and-add operation using just the `rlx` access mode (instead of the `rel` access mode). Such a change could potentially lead to a gain in performance.

3.1.6 Proof of function `unlock_shared`

We show the proof for the function `unlock_shared`. We need to show

$$\{\mathsf{U}(\texttt{bits}, \mathcal{Q}) * R_\beta(\texttt{bits})\} x := \texttt{fetch\_and\_add}_{\texttt{rel}}(\texttt{bits}, 4) \{\mathsf{emp}\}$$

using the fetch-and-add rule. We have $P \equiv \mathit{resource}(\texttt{bits})^\epsilon * \boxed{\beta : \epsilon}$. We choose

$$\begin{aligned}
\mathcal{P}_{\mathsf{send}}(t) &:= \mathit{resource}(\texttt{bits})^\epsilon * \boxed{\beta : \epsilon} \\
\mathcal{P}_{\mathsf{keep}}(t) &:= \mathsf{emp}
\end{aligned}$$

Clearly $P \Leftrightarrow \mathcal{P}_{\mathsf{send}}(t) * \mathcal{P}_{\mathsf{keep}}(t)$ holds for every t. We still need to show the CAS triple in the premise of the fetch-and-add rule for every value t that is read.

Let t be the value that is read. Consider first the case $\left\lfloor \frac{t}{4} \right\rfloor = 0$. We have $\mathcal{Q}(t) \Rightarrow \boxed{\beta : 1} * \mathsf{true}$, hence $\mathcal{P}_{\mathsf{send}}(t) * \mathcal{Q}(t) \Rightarrow \mathsf{false}$, which means such a t cannot be read and the CAS triple can be shown using the CAS$-\bot$ rule.

Next, assume $\mathsf{lsb}(t) = 1$ (i.e. there is a writer); we also show that such a value cannot be read (since we have real read permission). For such values t it holds that $\mathcal{Q}(t) \Rightarrow \boxed{\beta : 1} * \mathsf{true}$, hence $\mathcal{P}_{\mathsf{send}}(t) * \mathcal{Q}(t) \Rightarrow \mathsf{false}$ and therefore we can prove the CAS triple using the CAS$-\bot$ rule.

Next, assume $\left\lfloor \frac{t}{4} \right\rfloor > 0$ and $\mathsf{lsb}(t) = 0$. Choose $T := \mathcal{Q}(t)$ and $A := \mathsf{emp}$. We show the CAS triple using the CAS$-$*basic* rule. We have

$$T \Rightarrow \text{ let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in } \exists n'_r \in \mathbb{N}.\, t - 4 \geq 0 \land n'_r \leq n \land \mathit{resource}(\mathtt{bits})^{1-n'_r \cdot \epsilon} * \boxed{\alpha : 1} * \boxed{\beta : 1 - n'_r \cdot \epsilon} * \boxed{\gamma : 1 - (n - n'_r) \cdot \epsilon}$$

Note that $t - 4 \geq 0$ since we assume $\left\lfloor \frac{t}{4} \right\rfloor > 0$. Hence

$$\begin{aligned}
\mathcal{P}_{\mathsf{send}}(t) * T \Rightarrow & \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in} \\
& \exists n'_r \in \mathbb{N}.\, t - 4 \geq 0 \land n'_r - 1 \leq n - 1 \land \\
& \mathit{resource}(\mathtt{bits})^{1-(n'_r-1)\cdot\epsilon} * \\
& \boxed{\alpha : 1} * \boxed{\beta : 1 - (n'_r - 1) \cdot \epsilon} * \boxed{\gamma : 1 - (n - n'_r) \cdot \epsilon} \\
\Rightarrow & \text{let } n = \left\lfloor \frac{t-4}{4} \right\rfloor \text{ in} \\
& \exists n'_r \in \mathbb{N}.\, t - 4 \geq 0 \land n'_r - 1 \leq n \land \\
& \mathit{resource}(\mathtt{bits})^{1-(n'_r-1)\cdot\epsilon} * \\
& \boxed{\alpha : 1} * \boxed{\beta : 1 - (n'_r - 1) \cdot \epsilon} * \boxed{\gamma : 1 - (n - (n'_r - 1)) \cdot \epsilon} \\
\Rightarrow_{\text{choosing } n_r := n'_r - 1} & \mathcal{Q}(t-4)
\end{aligned}$$

Note that since the fetch-and-add operation is of access type `rel` we can directly transfer the permission $\mathit{resource}(\mathtt{bits})^{\epsilon}$ to the location invariant (i.e. we need not worry about any modalities). This concludes the proof of `unlock_shared`.

3.1.7 Proof of function `try_lock`

We show the proof for the function `try_lock`. We need to show

$$\{\mathsf{U}(\mathtt{bits}, \mathcal{Q})\}\, \mathtt{v} := \mathtt{CAS}_{\mathtt{rel\_acq}}(\mathtt{bits}, 0, 1) \left\{ \begin{pmatrix} \mathtt{v} = 0 \ ? \\ \mathit{resource}(\mathtt{bits})^1 * \boxed{\alpha : 1} : \\ \mathsf{U}(\mathtt{bits}, \mathcal{Q}) \end{pmatrix} \right\}$$

using the CAS$-$*basic* rule. We have

$$Q(0) \Leftrightarrow \mathit{resource}(\mathtt{bits})^1 * \boxed{\alpha : 1} * \boxed{\beta : 1} * \boxed{\gamma : 1}$$

We choose

$$T := \boxed{\alpha : 0} * \boxed{\beta : 1} * \boxed{\gamma : 1}$$
$$A := \mathit{resource}(\texttt{bits})^1 * \boxed{\alpha : 1}$$

and $Q(0) \Leftrightarrow A * T$ holds. Furthermore, $T \Leftrightarrow Q(1)$ holds and A is exactly the permission needed for the writer lock. Since the CAS has access type `rel_acq` we need not worry about modalities with A. Actually it suffices to have access type `acq` since no permission is given up by the thread. Such a change could potentially lead to a gain in performance. This concludes the proof.

3.1.8 Proof of function `unlock`

We show the proof for the function `unlock`. Note that we have only presented a rule for fetch-and-add operations in Figure 4, but the rule can be adjusted in a straight forward fashion to support $\texttt{fetch\_and\_and}_{\texttt{rel}}(\texttt{bits}, \sim 1)$. Instead of $t + v$ in the premise we just use $\mathsf{clearlsb}(t)$, which maps t to the integer where the least significant bit is set to 0. We refer to this adjusted rule as the *fetch-and-clear rule*.

Case 1: `getRead` does not hold. We first consider the case where `getRead` does not hold. In this case we need to show

$$\left\{\ W_\alpha(\texttt{bits})\ \right\} \texttt{fetch\_and\_and}_{\texttt{rel}}(\texttt{bits}, \sim 1) \left\{\ \texttt{emp}\ \right\}$$

using the fetch-and-clear rule. Hence we have $P \equiv \mathit{resource}(\texttt{bits})^1 * \boxed{\alpha : 1}$. We choose

$$\mathcal{P}_{\mathsf{send}}(t) := \mathit{resource}(\texttt{bits})^1 * \boxed{\alpha : 1}$$
$$\mathcal{P}_{\mathsf{keep}}(t) := \mathsf{emp}$$

i.e. we choose to give everything away independent of the value that we read before writing the new value. Clearly $P \Leftrightarrow \mathcal{P}_{\mathsf{send}}(t) * \mathcal{P}_{\mathsf{keep}}(t)$ holds for every t.

Since there is a writer the least significant bit of the value that we read during the update must be 1. We still need to verify this. Let t be the value that we read. Suppose $\mathsf{lsb}(t) = 0$, then we have $\mathcal{Q}(t) \Rightarrow \boxed{\alpha : 1} * \mathsf{true}$. Hence $\mathcal{Q}(t) * \mathcal{P}_{\mathsf{send}}(t) \Rightarrow \mathsf{false}$ (since $\boxed{\alpha : 1} * \boxed{\alpha : 1} \Rightarrow \mathsf{false}$). Therefore using the CAS$-\bot$ rule we can show the CAS triple in the premise of the fetch-and-clear rule for such t (since reading that t is not possible, which is what we just proved).

Let t be the value that is read and now suppose $\mathsf{lsb}(t) = 1$. We will now use the CAS−*basic* rule to verify the premise of the fetch-and-clear rule for such t. Observe that in this case

$$\mathcal{Q}(t) \Leftrightarrow \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in } t \geq 0 \wedge \boxed{\alpha : 0} * \boxed{\beta : 1} * \boxed{\gamma : 1 - n \cdot \epsilon}$$

Since we do not want to acquire any permissions, we choose $A := \mathsf{emp}$ and $T := \mathcal{Q}(t)$ in the premise of the CAS rule. We have

$$\begin{aligned} \mathcal{P}_{\mathsf{send}}(t) * T \Rightarrow\ & \text{let } n = \left\lfloor \tfrac{\mathsf{clearlsb}(t)}{4} \right\rfloor \text{ in} \\ & \mathsf{clearlsb}(t) \geq 0 \wedge \mathit{resource}(\mathtt{bits})^1 * \boxed{\alpha : 1} * \\ & \boxed{\beta : 1} * \boxed{\gamma : 1 - n \cdot \epsilon} \\ \Rightarrow_{\text{choosing } n_r := 0}\ & \mathcal{Q}(\mathsf{clearlsb}(t)) \end{aligned}$$

Hence for these t the premise of the fetch-and-clear rule also holds. This concludes the proof.

Case 2: `getRead` holds. In this case we need to show

$$\left\{\ W_\alpha(\mathtt{bits}) * \boxed{\gamma : \epsilon}\ \right\} \mathtt{fetch\_and\_and}_{\mathtt{rel}}(\mathtt{bits}, \sim 1) \left\{\ R_\beta(\mathtt{bits})\ \right\}$$

The main difference to the first case is that we retain part of the ownership to the resource and we exchange $\boxed{\gamma : \epsilon}$ for $\boxed{\beta : \epsilon}$.

We need to prove the premise of the fetch-and-clear rule (i.e. the rule in Figure 4 updated as in the previous case). We have

$$P \equiv \mathit{resource}(\mathtt{bits})^1 * \boxed{\alpha : 1} * \boxed{\gamma : \epsilon}$$

We choose

$$\begin{aligned} \mathcal{P}_{\mathsf{send}}(t) &:= \left(\left\lfloor \frac{t}{4} \right\rfloor = 0\ ?\ P : \mathit{resource}(\mathtt{bits})^{1-\epsilon} * \boxed{\alpha : 1} * \boxed{\gamma : \epsilon} \right) \\ \mathcal{P}_{\mathsf{keep}}(t) &:= \left(\left\lfloor \frac{t}{4} \right\rfloor = 0\ ?\ \mathsf{emp} : \mathit{resource}(\mathtt{bits})^{\epsilon} \right) \end{aligned}$$

Since $\boxed{\gamma : \epsilon}$ is part of P it must hold that the value stored at the atomic location is at least 4. Setting $\mathcal{P}_{\mathsf{send}}(t)$ to P if $\left\lfloor \frac{t}{4} \right\rfloor = 0$ allows verifying this as we will see next. Furthermore, since in this case a reader lock must be acquired, we retain permission to the data structure guarded by the lock by setting $\mathcal{P}_{\mathsf{keep}}(t)$ to $\mathit{resource}(\mathtt{bits})^{\epsilon}$ if $\left\lfloor \frac{t}{4} \right\rfloor \neq 0$. Clearly $P \Leftrightarrow \mathcal{P}_{\mathsf{send}}(t) * \mathcal{P}_{\mathsf{keep}}(t)$ holds for every t.

Let t be the value that was read. We first prove the premise of the CAS triple in the fetch-and-clear rule if $\left\lfloor \frac{t}{4} \right\rfloor = 0$. Such t cannot be read

since intuitively $\boxed{\gamma : \epsilon}$ is held outside the invariant. Hence we show the premise again using the CAS$-\bot$ rule. Suppose $\lfloor \frac{t}{4} \rfloor = 0$. We then have $\mathcal{P}_{\mathsf{send}}(t) \Rightarrow \boxed{\gamma : \epsilon} * \mathsf{true}$ and $\mathcal{Q}(t) \Rightarrow \boxed{\gamma : 1} * \mathsf{true}$, hence $\mathcal{P}_{\mathsf{send}}(t) * \mathcal{Q}(t) \Rightarrow \mathsf{false}$. Therefore for such t the premise of the fetch-and-and rule is proved using the CAS$-\bot$ rule. Note that this proof is only possible since $\boxed{\gamma : \epsilon}$ is in the precondition of the `unlock` method.

Next, assume $\lfloor \frac{t}{4} \rfloor > 0$. We have

$$\mathcal{P}_{\mathsf{send}}(t) \Leftrightarrow \mathit{resource}(\mathtt{bits})^{1-\epsilon} * \boxed{\alpha : 1} * \boxed{\gamma : \epsilon}$$

In the case when $\mathsf{lsb}(t) = 0$ we can show analogously to the case when `getRead` does not hold that the CAS triple holds using the CAS$-\bot$ rule.

So we now assume $\lfloor \frac{t}{4} \rfloor > 0$ and $\mathsf{lsb}(t) = 1$. We prove the CAS triple using the CAS$-$*basic* rule. We have

$$\mathcal{Q}(t) \Leftrightarrow \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in } t \geq 0 \land n \geq 1 \land \boxed{\alpha : 0} * \boxed{\beta : 1} * \boxed{\gamma : 1 - n \cdot \epsilon}$$

Note that $n \geq 1$ holds since we assumed $\lfloor \frac{t}{4} \rfloor > 0$. We choose

$$T := \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in } t \geq 0 \land n \geq 1 \land \boxed{\alpha : 0} * \boxed{\beta : 1 - \epsilon} * \boxed{\gamma : 1 - n \cdot \epsilon}$$
$$A := \boxed{\beta : \epsilon}$$

The reason for this choice is that we need to extract $\boxed{\beta : \epsilon}$ to finally get the reader permission and we need not extract anything related to the predicate $\mathit{resource}(\mathtt{bits})$ since we chose $\mathcal{P}_{\mathsf{keep}}(t) \Leftrightarrow \mathit{resource}(\mathtt{bits})^{\epsilon}$ in this case. $\mathcal{Q}(t) \Leftrightarrow A * T$ holds. Furthermore, we have

$$\begin{aligned}
\mathcal{P}_{\mathsf{send}}(t) * T \Rightarrow & \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in} \\
& t \geq 0 \land 1 \leq n \land \mathit{resource}(\mathtt{bits})^{1-\epsilon} * \boxed{\alpha : 1} * \\
& \boxed{\beta : 1 - \epsilon} * \boxed{\gamma : 1 - (n-1) \cdot \epsilon} \\
& \Rightarrow_{\text{choosing } n_r := 1} \text{let } n = \left\lfloor \frac{\mathsf{clearlsb}(t)}{4} \right\rfloor \text{ in} \\
& \exists n_r \in \mathbb{N}.\, \mathsf{clearlsb}(t) \geq 0 \land n_r \leq n \land \mathit{resource}(\mathtt{bits})^{1-n_r \cdot \epsilon} * \\
& \boxed{\alpha : 1} * \boxed{\beta : 1 - n_r \cdot \epsilon} * \boxed{\gamma : 1 - (n - n_r) \cdot \epsilon} \\
& \Rightarrow Q(\mathsf{clearlsb}(t))
\end{aligned}$$

```
void unlock_and_lock_shared() {
    {U(bits, Q) * W_α(bits)}
    x := fetch_and_add_acq(bits, 4) //RMW1
    {U(bits, Q) * W_α(bits) * [γ : ε]}
    unlock(true)
    {R_β(bits)}
}
```

Figure 7: Proof outline for function ***unlock_and_lock_shared***

which concludes the proof of the premise of the fetch-and-and rule. Finally we have $\mathcal{P}_{\mathsf{keep}}(t) * A \Leftrightarrow \mathit{resource}(\mathtt{bits})^{\epsilon} * \boxed{\beta : \epsilon}$ which is exactly the postcondition of the `unlock` function in this case. This concludes the case when `getRead` holds.

3.1.9 Proof of function `unlock_and_lock_shared`

The proof outline for the function `unlock_and_lock_shared` is given in Figure 7. The first fetch-and-add operation can be verified analogously to the first fetch-and-add operation in the proof of the function `try_lock_shared` (see Section 3.1.5) in the case when fake read permission is obtained. The rest of the proof relies on the specification of `unlock`.

3.1.10 Using only a single ghost location for the readers

In the location invariant that we introduce in Section 3.1.3 for the proofs we use two separate ghost locations for the real and fake readers. A question that arises is if it is possible to use a single ghost location for both types of readers. For example, one could consider the following location invariant:

$$\begin{aligned}\mathcal{Q}(v) :=& \text{let } n = \left\lfloor \frac{v}{4} \right\rfloor, w = \mathsf{lsb}(v) \text{ in} \\ & \exists n_r \in \mathbb{N}.\, v \geq 0 \wedge n_r \leq n \wedge (w = 1 \Rightarrow n_r = 0) \wedge \\ & \mathit{resource}(\mathtt{bits})^{(w=1\ ?\ 0\ :\ 1-n_r\cdot\epsilon)} * \boxed{\alpha : 1 - w} * \boxed{\delta : 1 - n \cdot \epsilon}\end{aligned}$$

where α, δ are ghost locations governed by the fractional permission structure and δ tracks both types of readers. We have as real reader permission

$$\mathit{resource}(\mathtt{bits})^{\epsilon} * \boxed{\delta : \epsilon}$$

It turns out that all the functions except `unlock_shared` can be verified using this location invariant with respect to the (adjusted) specification given in

Section 3.1.4. The issue with `unlock_shared` is that given the real reader permission it is not possible to prove that there is no writer.

To see this, assume that there is a writer, i.e. a value t is read with $\mathsf{lsb}(t) = 1$. We then have $\mathcal{Q}(t) \Leftrightarrow \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in } t \geq 0 \wedge \boxed{\alpha : 0} * \boxed{\delta : 1 - n \cdot \epsilon}$. It does not hold that $\mathcal{Q}(t) * \mathit{resource}(\texttt{bits})^{\epsilon} * \boxed{\delta : \epsilon} \Rightarrow \mathsf{false}$ hence it cannot be shown that such t cannot be read using the CAS$-\bot$ rule. This means that in the proof outline of `unlock_shared` there are cases where the reader retains $\mathit{resource}(\texttt{bits})^{\epsilon}$ (which in reality of course cannot happen). Therefore at the end of the method we do not get emp[6]. So we cannot satisfy the specification in a precise manner.

We do not know if there is a location invariant, where each ghost location is governed by the fractional permission structure, that uses fewer ghost locations than the location invariant we use for all the proofs. In a later section we introduce a permission structure with which it is possible to express a location invariant using a single ghost location for both types of readers.

3.2 Folly barrier

The Folly barrier example [4] (we will refer to it as *Barrier*) gives a weak-memory implementation of a barrier. Barriers are used to synchronize a set of threads. The actual implementation uses arrays and other data structures, but we consider a simplified version that does not use any special data structures. Our version focuses on the part which deals with the different roles threads take when entering the barrier.

We provide two FSL++ proofs (with two different location invariants). The first proof uses a more complicated location invariant but does not need the CAS$-$*param* rule while the second proof uses a simpler location invariant but needs the CAS$-$*param* rule.

3.2.1 Implementation

The implementation of *Barrier* that we consider is given in Figure 8. It consists of a single function `wait`. The barrier implementation uses a single 32-bit atomic integer location `VRC` (short for *value-and-reader-count*). The 16 most significant bits of `VRC` encode the number of threads that have called `wait` (we call this the *value count*) and the 16 least significant bits of `VRC` encode the number of threads that have called `wait` and for which `wait` has not yet returned (we call this the *reader count*). Hence the reader count is always at most as big as the value count.

[6] Real permission cannot be leaked in classical separation logic, i.e. $\mathit{resource}(\texttt{bits})^{\epsilon} \implies \mathsf{emp}$ does not hold in general.

```
wait() {
  (r1, v1) := fetch_and_add_acq(VRC, (1, 1)) //first RMW

  if(v1+1 == n) {
        //update signal data structure
  }

  (r2, v2) := fetch_and_add_rel_acq(VRC, (−1, 0)) //second RMW

  if((r2, v2) == (1,n)) {
        //deallocate signal data structure
  }
}
```

Figure 8: Implementation of function in `wait` *in Barrier*

Instead of regarding `VRC` directly as an integer we represent the value as a tuple (r, v) where r is the reader count and v is the value count. This makes the presentation cleaner.

The idea of the implementation is as follows. There are n threads that are supposed to use the barrier (n is initialized in the actual implementation when the barrier is allocated; we do not deal with the allocation part here). We assume $n \geq 1$.

When a thread calls `wait`, the value and reader count are first incremented (since it has entered `wait` and `wait` has not yet returned); this corresponds to adding $(1, 1)$ in our representation. If the thread is the nth thread to enter `wait` (i.e. the last thread) then the thread updates a specific data structure associated with the barrier. We call this data structure the *signal data structure*, since updating it signals to all the threads that every thread has entered the barrier. We do not care about how the data structure represented. It must be ensured that at most one thread updates the signal data structure at any time. Hence the thread performing the update conceptually needs to get a token that represents the full permission to access the data structure.

Before `wait` returns, the thread (regardless of whether it is the last thread or not) decrements the reader count (this corresponds to adding $(-1, 0)$ in our representation). If the thread is the nth thread to leave `wait` then the thread deallocates the signal data structure, i.e. conceptually it again needs access to the same token as before.

Note that it is possible for the thread which updates the signal data structure (after the first fetch-and-add operation) to be different from the thread which deallocates the signal data structure (after the second fetch-and-add operation).

In the actual implementation once the signal data structure is updated, a new signal data structure is allocated and a new epoch starts (the old signal data structure still needs to be deallocated as described above), i.e. the barrier can be reused. We do not model this allocation and hence our implementation only models a single epoch. Effectively, we make the assumption that the barrier is used only once.

3.2.2 Token to signal data structure

We use a heap location *tok.val* as the token which is to be used for the signal data structure, i.e. the full permission to *tok.val* ($tok.val \overset{1}{\mapsto} \_$) models ownership of the signal data structure. So if in a thread's state $tok.val \overset{1}{\mapsto} \_$ holds then no other thread has access to the signal data structure at that time. It would be more general to use an abstract predicate as we did for *RWSpin* (see Section 3.1) but this would require additional ghost locations in the location invariant which would make the presentation harder.

3.2.3 Specification

The specification that we verify for `wait` itself is simple:

$$\{\mathsf{U}(\texttt{VRC}, \mathcal{Q})\}\,\texttt{wait()}\,\{\mathsf{true}\}$$

Note that the postcondition is true and not emp. The main reason is that in our proofs we do not make sure that `wait` is called at most n times, since there does not seem to be a simple way to do so. As a result the thread that updates the signal data structure may retain the permission to the data structure in our proofs even when the whole code has been executed (this scenario cannot occur if at most n threads enter the thread).

3.2.4 Location invariant A

Location invariant A for the atomic location `VRC` in *Barrier* is given by:

$$\begin{aligned} Q(r,v) := \; & \exists w \in \{0,1\}.\, 0 \le r \wedge r \le v \wedge (v < n \Rightarrow w = 0) \wedge \boxed{\alpha : 1 - r \cdot \epsilon} * \\ & ([(r,v) \neq (0,n) \wedge (v \le n)] \;?\; tok.val \overset{1-w}{\mapsto} \_ : \mathsf{emp}) * \\ & ([(r,v) \neq (0,n) \wedge (v \le n)] \;?\; \boxed{\beta : 1 - (r-w) \cdot \epsilon} : \mathsf{emp}) * \\ & \boxed{\gamma : 1 - (v > n \;?\; (v-n) \cdot \epsilon : 0)} \end{aligned}$$

where r is the reader count and v is the value count and α, β, γ are ghost locations which are governed by the fractional permission structure. We use ϵ to model counting permissions (see Section 2.4).

The idea is as follows. We have no easy way to ensure that only n threads will call `wait`. So our specification allows more than n threads to call `wait` but once that happens there are no functional guarantees (data race freedom is still guaranteed). As long as at most n threads call `wait` our specification works as expected.

We can say that the barrier has "finished its task" once either the value of `VRC` is $(0, n)$ (i.e. n threads have entered and left the barrier) or more than n threads have entered the barrier (i.e. $v > n$ which should not happen in practice). So the condition

$$[(r, v) \neq (0, n) \wedge (v \leq n)]$$

is true iff the barrier has not yet finished its task (assuming $0 \leq r \leq v$).

The existentially quantified variable w has actual value 1 if the nth thread has entered `wait` and has not yet left `wait`. Recall that the nth thread needs the token to set the promises; hence if $w = 1$ then the token is definitely not in the location invariant.

Note that $w \leq r$ is implied by the location invariant as long as the barrier has not finished its task, since then if $w > r$ then $\mathcal{Q}(r, v) \Rightarrow \boxed{\beta : 1 + (w - r) \cdot \epsilon} *$ true and since $(w - r) > 0$ we would have $\mathcal{Q}(r, v) \Rightarrow \mathsf{false}$.

The ghost location α tracks all the threads that have entered but not yet left `wait`. The ghost location β tracks the first $n - 1$ threads that enter `wait`. Ghost location γ tracks the threads that enter `wait` after n threads have already entered `wait` (i.e. those threads which should not occur); we track them so we can verify our specification without additional assumptions.

3.2.5 Proof outline (A)

The proof outline for `wait` with respect to the specification in Section 3.2.3 using location invariant A[7] is given in Figure 9. We make the assumption in our proofs that we never read reader or value counts that are negative. This assumption is fine since $\mathcal{Q}(r, v) \Rightarrow \mathsf{false}$ if $r < 0$ or $v < 0$, so we would be able to deal with that case using the CAS$-\bot$ rule (see Figure 3).

Note that we use true in cases where we do not really care what the permissions are. We are mainly interested that the permission for the signal data structure (i.e. the token) is received at the correct time. So we need to ensure that the thread updating the data structure and the thread deallocating the data structure get hold of the token when they perform these actions. We also do not explicitly carry along $\mathsf{U}(\mathtt{VRC}, \mathcal{Q})$ even though it is needed for the fetch-and-add operations to make the proof outline less cluttered (but

[7] see Section 3.2.4 for the definition of location invariant A

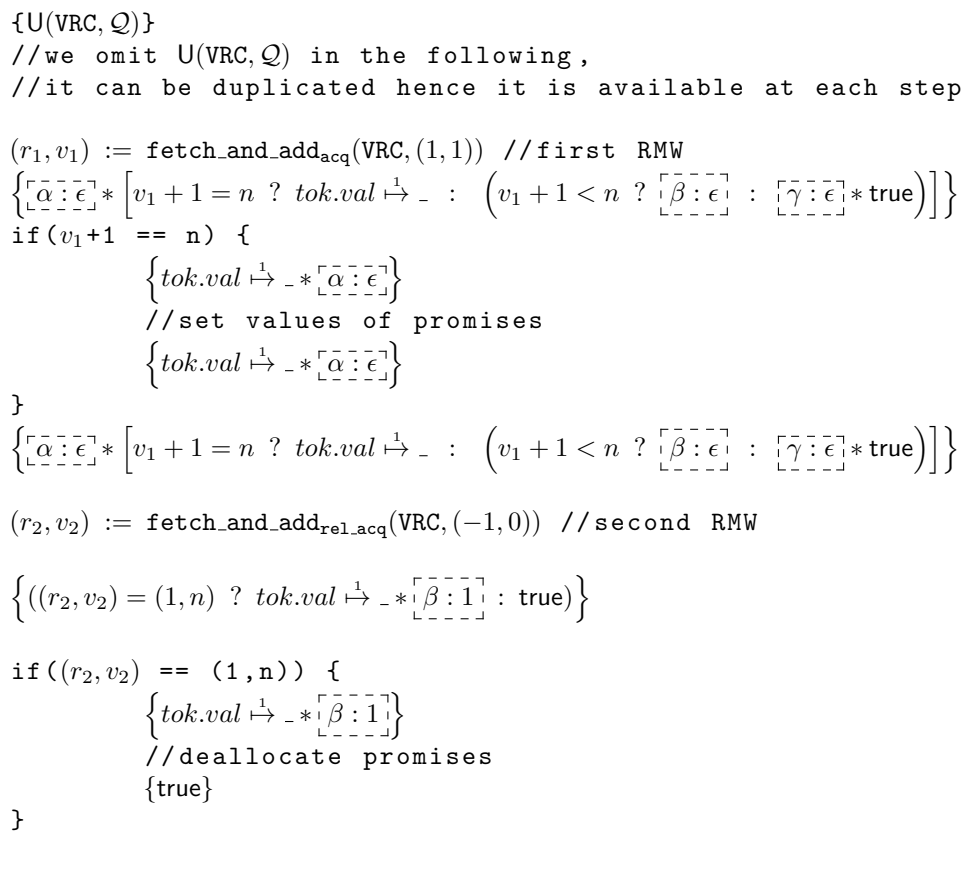

Figure 9: Proof outline for `wait` *in Barrier*

since $\mathsf{U}(\texttt{VRC}, \mathcal{Q})$ is duplicable we may assume it is always there when we need it).

3.2.6 Proof of first RMW (A)

We verify the first read-modify-write operation given in the proof outline in Figure 9 with respect to location invariant A. We have to show

$$\{\mathsf{U}(\texttt{VRC}, \mathcal{Q})\}$$
$$(r_1, v_1) := \texttt{fetch\_and\_add}_{\mathsf{acq}}(\texttt{VRC}, (1,1))$$
$$\left\{ \begin{array}{l} \boxed{\alpha : \epsilon} * \\ \left[v_1 + 1 = n \ ?\ tok.val \overset{1}{\mapsto} \_ \ :\ \left(v_1 + 1 < n \ ?\ \boxed{\beta : \epsilon} \ :\ \boxed{\gamma : \epsilon} * \mathsf{true}\right)\right] \end{array} \right\}$$

using the fetch-and-add rule in Figure 4. We have $P \equiv \mathsf{emp}$ and hence $\mathcal{P}_{\mathsf{send}}(t) := \mathcal{P}_{\mathsf{keep}}(t) := \mathsf{emp}$. We now show the CAS triple in the premise of the fetch-and-add rule for all possible values (r, v) using the CAS$-$*basic* rule.

Assume we read an arbitrary value (r, v) (we assume $r, v \geq 0$ since as explained in Section 3.2.5 other values cannot be read). We will make a case distinction on (r, v). In all cases we implicitly use the observation that if $[(r, v) \neq (0, n) \wedge (v \leq n)]$ does not hold then $[(r{+}1, v{+}1) \neq (0, n) \wedge (v{+}1 \leq n)]$ does not hold either. This means that if the permissions guarded by this condition in the location invariant are not there because the condition is false then after the first fetch-and-add operation we need not guarantee these permissions.

Case $v + 1 < n$**.** Then we have

$$\mathcal{Q}(r, v) \Leftrightarrow 0 \leq r \wedge r \leq v \wedge \boxed{\alpha : 1 - r \cdot \epsilon} * tok.val \overset{1}{\mapsto} \_ * \boxed{\beta : 1 - r \cdot \epsilon} * \boxed{\gamma : 1}$$

We choose

$$T := (0 \leq r) \wedge (r + 1 \leq v + 1) \wedge \boxed{\alpha : 1 - (r+1) \cdot \epsilon} * tok.val \overset{1}{\mapsto} \_ * \boxed{\beta : 1 - (r+1) \cdot \epsilon} * \boxed{\gamma : 1}$$
$$A := \boxed{\alpha : \epsilon} * \boxed{\beta : \epsilon}$$

It holds that $\mathcal{Q}(r, v) \Leftrightarrow A * T$ and $T \Rightarrow \mathcal{Q}(r + 1, v + 1)$. Since additionally A is exactly the permission we want in this case and the fetch-and-add is of access type `acq`, this case is verified.

Case $v + 1 = n$**.** In this case we have

$$\mathcal{Q}(r, v) \Leftrightarrow 0 \leq r \wedge r \leq v \wedge \boxed{\alpha : 1 - r \cdot \epsilon} * tok.val \overset{1}{\mapsto} \_ * \boxed{\beta : 1 - r \cdot \epsilon} * \boxed{\gamma : 1}$$

We choose

$$T := (0 \leq r) \land (r+1 \leq v+1) \land \boxed{\alpha : 1-(r+1)\cdot\epsilon} * \boxed{\beta : 1-r\cdot\epsilon} * \boxed{\gamma : 1}$$
$$A := \mathit{tok.val} \overset{1}{\mapsto} \_ * \boxed{\alpha : \epsilon}$$

since we need to get permission to the token. We have $\mathcal{Q}(r,v) \Leftrightarrow A * T$. Furthermore, it holds that

$$\begin{aligned} T \Rightarrow_{\text{choosing } w:=1} \quad & \exists w \in \{0,1\}.\, (0 \leq r+1) \land (r+1 \leq v+1) \land \\ & \boxed{\alpha : 1-(r+1)\cdot\epsilon} * \\ & ([(r,v) \neq (0,n) \land (v \leq n)]\ ?\ \mathit{tok.val} \overset{1-w}{\mapsto} \_ : \mathsf{emp}) * \\ & ([(r,v) \neq (0,n) \land (v \leq n)]\ ?\ \boxed{\beta : 1-(r+1-w)\cdot\epsilon} : \mathsf{emp}) * \\ & \boxed{\gamma : 1-(v+1>n\ ?\ (v+1-n)\cdot\epsilon : 0)} \\ \Rightarrow \mathcal{Q}(r+1, v+1) & \end{aligned}$$

Finally, since the fetch-and-add is of access type `acq` we need not worry about any modalities when acquiring the permission to the token which concludes this case.

Case $v+1 > n$**.** This is the case that should not happen if the barrier is used as intended, but we still guarantee that there are no data races if this scenario occurs. We consider two subcases $v = n$ and $v > n$. In both subcases we use the observation that $[(r+1, v+1) \neq (0,n) \land (v+1 \leq n)]$ is false, i.e. the permissions in the location invariant that are guarded by this condition need not be guaranteed after the update.

If $v > n$ we have

$$\mathcal{Q}(r,v) \Leftrightarrow 0 \leq r \land r \leq v \land \boxed{\alpha : 1-r\cdot\epsilon} * \boxed{\gamma : 1-(v-n)\cdot\epsilon}$$

We choose

$$T := (0 \leq r) \land (r \leq v) \land \boxed{\alpha : 1-(r+1)\cdot\epsilon} * \boxed{\gamma : 1-((v+1)-n)\cdot\epsilon}$$
$$A := \boxed{\alpha : \epsilon} * \boxed{\gamma : \epsilon}$$

We have $\mathcal{Q}(r,v) \Rightarrow A * T$. It holds that $T \Rightarrow \mathcal{Q}(r+1, v+1)$ and A is the permission we want in this case (additionally the fetch-and-add is of access type `acq`), so this subcase is verified.

Next, if $v = n$ we have

$$\begin{aligned} Q(r,v) \Leftrightarrow & \exists w \in \{0,1\}.\, 0 \le r \wedge r \le v \wedge \boxed{\alpha : 1 - r \cdot \epsilon} * \\ & ([(r,v) \neq (0,n) \wedge (v \le n)] \ ?\ tok.val \overset{1-w}{\mapsto} \_ : \mathsf{emp}) * \\ & ([(r,v) \neq (0,n) \wedge (v \le n)] \ ?\ \boxed{\beta : 1 - (r-w) \cdot \epsilon} : \mathsf{emp}) * \\ & \boxed{\gamma : 1} \end{aligned}$$

We choose

$$\begin{aligned} T &:= 0 \le r \wedge r \le v \wedge \boxed{\alpha : 1 - (r+1) \cdot \epsilon} * \boxed{\gamma : 1 - \epsilon} \\ A &:= \boxed{\alpha : \epsilon} * \boxed{\gamma : \epsilon} * \mathsf{true} \end{aligned}$$

We have $\mathcal{Q}(r,v) \Rightarrow A * T$. The permissions that are guarded by the condition $[(r,v) \neq (0,n) \wedge (v \le n)]$ are included in A (they are implicitly inside the true assertion of A; A is of the form $A_{left} * \mathsf{true}$ for some assertion A_{left}). Furthermore, $T \Rightarrow \mathcal{Q}(r+1, v+1)$ and A is exactly the permission we want in this case (additionally the fetch-and-add is of access type $\mathtt{acq}$), so this subcase is verified.

3.2.7 Proof of second RMW (A)

We verify the second read-modify-write operation given in the proof outline in Figure 9 with respect to location invariant A. We have to show

$$\left\{ \begin{array}{l} \mathsf{U}(\mathtt{VRC}, \mathcal{Q}) * \boxed{\alpha : \epsilon} * \\ \left[v_1 + 1 = n \ ?\ tok.val \overset{1}{\mapsto} \_ \ : \ \left(v_1 + 1 < n \ ?\ \boxed{\beta : \epsilon} \ : \ \boxed{\gamma : \epsilon} * \mathsf{true} \right) \right] \end{array} \right\}$$

$$(r_2, v_2) := \mathtt{fetch\_and\_add}_{\mathtt{rel\_acq}}(\mathtt{VRC}, (-1, 0))$$

$$\left\{ ((r_2, v_2) = (1, n) \ ?\ tok.val \overset{1}{\mapsto} \_ * \boxed{\beta : 1} : \mathsf{true}) \right\}$$

We have

$$P \equiv \boxed{\alpha : \epsilon} * \left[\begin{array}{ll} v_1 + 1 = n & ?\ tok.val \overset{1}{\mapsto} \_ \\ & : \ \left(v_1 + 1 < n \ ?\ \boxed{\beta : \epsilon} \ : \ \boxed{\gamma : \epsilon} * \mathsf{true} \right) \end{array} \right]$$

Instead of writing down $\mathcal{P}_{\mathsf{send}}(r,v)$ and $\mathcal{P}_{\mathsf{keep}}(r,v)$ for general (r,v) we will give the definitions in the different (disjoint) subcases for the sake of presentation. We make a case distinction on the value (r,v) that is read.

Case 1: $r = 0$**.** We show this case cannot occur by verifying the CAS triple using the CAS$-\bot$ rule. Define in this case

$$\begin{aligned} \mathcal{P}_{\mathsf{send}}(r,v) &:= P \\ \mathcal{P}_{\mathsf{keep}}(r,v) &:= \mathsf{emp} \end{aligned}$$

We have $\mathcal{Q}(0,v) \Rightarrow \boxed{\alpha : 1} * \mathsf{true}$ and $\mathcal{P}_{\mathsf{send}}(r,v) \Rightarrow \boxed{\alpha : \epsilon} * \mathsf{true}$. Hence $\mathcal{P}_{\mathsf{send}}(r,v) * \mathcal{Q}(0,v) \Rightarrow \mathsf{false}$ which means we can use the CAS$-\bot$ rule to show the CAS triple.

Case 2: $(r,v) \neq (1,n)$ **and** $r > 0$ **and** $v \leq n$**.** We verify this case using the CAS$-$*basic* rule. Define in this case

$$\mathcal{P}_{\mathsf{send}}(r,v) := P$$
$$\mathcal{P}_{\mathsf{keep}}(r,v) := \mathsf{emp}$$

We have

$$\mathcal{Q}(r,v) \Leftrightarrow \exists w \in \{0,1\}.\, 0 \leq r \wedge r \leq v \wedge (v < n \Rightarrow w = 0) \wedge \boxed{\alpha : 1 - r \cdot \epsilon} * tok.val \overset{1-w}{\mapsto} \_ * \boxed{\beta : 1 - (r-w) \cdot \epsilon} * \boxed{\gamma : 1}$$

where we used that $[(r,v) \neq (0,n) \wedge (v \leq n)]$ holds in this case. Choose $T := \mathcal{Q}(r,v)$ and $A := \mathsf{emp}$. We need to show that $\mathcal{P}_{\mathsf{send}}(r,v) * T \Rightarrow \mathcal{Q}(r-1,v)$ holds. Since $\mathcal{P}_{\mathsf{send}}(r,v)$ depends on the value (r_1,v_1) read in the first read-modify write operation it is cumbersome to show this implication directly covering all cases at once. Therefore we make a case distinction on (r_1,v_1) and show that in each case $\mathcal{P}_{\mathsf{send}}(r,v) * T \Rightarrow \mathcal{Q}(r-1,v)$.

If $v_1 + 1 = n$ then we have $\mathcal{P}_{\mathsf{send}}(r,v) \Leftrightarrow \boxed{\alpha : \epsilon} * tok.val \overset{1}{\mapsto} \_$, hence

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(r,v) * T \Rightarrow\ & (0 \leq r-1) \wedge (r-1 \leq v) \wedge \boxed{\alpha : 1 - (r-1) \cdot \epsilon} * \\ & tok.val \overset{1}{\mapsto} \_ * \boxed{\beta : 1 - (r-1) \cdot \epsilon} * \boxed{\gamma : 1} \\ \Rightarrow_{\text{choosing } w:=0}\ & \mathcal{Q}(r-1,v)\end{aligned}$$

where used that $tok.val \overset{1}{\mapsto} \_ * tok.val \overset{1}{\mapsto} \_ \Rightarrow \mathsf{false}$, hence we can conclude that the existential w must have value 1 in $\mathcal{Q}(r,v)$ when considering $\mathcal{P}_{\mathsf{send}}(r,v) * Q(r,v)$. Furthermore, we use the fact that in this case $[(r-1,v) \neq (0,n) \wedge (v \leq n)]$ holds.

If $v_1 + 1 < n$ then we have $\mathcal{P}_{\mathsf{send}}(r,v) \Leftrightarrow \boxed{\alpha : \epsilon} * \boxed{\beta : \epsilon}$, hence

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(r,v) * T \Rightarrow\ & \exists w \in \{0,1\}.\, 0 \leq r \wedge r \leq v \wedge (v < n \Rightarrow w = 0) \wedge \\ & \boxed{\alpha : 1 - (r-1) \cdot \epsilon} * tok.val \overset{1-w}{\mapsto} \_ * \\ & \boxed{\beta : 1 - ((r-1) - w) \cdot \epsilon} * \boxed{\gamma : 1} \\ & \Rightarrow \mathcal{Q}(r-1,v)\end{aligned}$$

Finally, if $v_1 + 1 > n$ then we have $\mathcal{P}_{\mathsf{send}}(r,v) \Leftrightarrow \boxed{\alpha : \epsilon} * \boxed{\gamma : \epsilon} * \mathsf{true}$. Conceptually, this cannot occur since the value count never becomes smaller.

This fact is captured, since we have $T \Rightarrow \boxed{\gamma : 1} * \mathsf{true}$, hence $\mathcal{P}_{\mathsf{send}}(r, v) * T \Rightarrow \mathsf{false}$, therefore $\mathcal{P}_{\mathsf{send}}(r, v) * T \Rightarrow \mathcal{Q}(r - 1, v)$ holds trivially.

Case 3: $(r, v) = (1, n)$**.** This is the most interesting case because here we need to make sure that we get access to the token after the fetch-and-add operation. We define

$$\mathcal{P}_{\mathsf{send}}(1, n) := \boxed{\alpha : \epsilon} * \left[v_1 + 1 = n \ ?\ \mathsf{emp} \ : \ \left(v_1 + 1 < n \ ?\ \mathsf{emp} \ : \ \boxed{\gamma : \epsilon}\right)\right]$$
$$\mathcal{P}_{\mathsf{keep}}(1, n) := \left[v_1 + 1 = n \ ?\ tok.val \overset{1}{\mapsto} \_ \ : \ \left(v_1 + 1 < n \ ?\ \boxed{\beta : \epsilon} \ : \ \mathsf{true}\right)\right]$$

We have

$$\mathcal{Q}(1, n) \Leftrightarrow \exists w \in \{0, 1\}. \boxed{\alpha : 1 - \epsilon} * tok.val \overset{1-w}{\mapsto} \_ * \boxed{\beta : 1 - (1 - w) \cdot \epsilon} * \boxed{\gamma : 1}$$

where we used that $[(r, v) \neq (0, n) \wedge (v \leq n)]$ holds in this case. Another way to write this is using a disjunction ($w = 0$ or $w = 1$):

$$\mathcal{Q}(1, n) \Leftrightarrow \boxed{\alpha : 1 - \epsilon} * \left[\left(tok.val \overset{1}{\mapsto} \_ * \boxed{\beta : 1 - \epsilon}\right) \vee \boxed{\beta : 1}\right] * \boxed{\gamma : 1}$$

Choose

$$T := \boxed{\alpha : 1 - \epsilon} * \boxed{\gamma : 1}$$
$$A := \left(tok.val \overset{1}{\mapsto} \_ * \boxed{\beta : 1 - \epsilon}\right) \vee \boxed{\beta : 1}$$

It holds that $\mathcal{Q}(1, n) \Leftrightarrow A * T$. We need to prove $\mathcal{P}_{\mathsf{send}}(1, n) * T \Rightarrow \mathcal{Q}(0, n)$. Note that

$$\mathcal{Q}(0, n) \Leftrightarrow \boxed{\alpha : 1} * \boxed{\gamma : 1}$$

If $v_1 + 1 = n$ or $v_1 + 1 < n$ then $\mathcal{P}_{\mathsf{send}}(1, n) = \boxed{\alpha : \epsilon}$ and therefore $\mathcal{P}_{\mathsf{send}}(1, n) * T \Rightarrow \mathcal{Q}(0, n)$.

In the remaining case $\mathcal{P}_{\mathsf{send}}(1, n) \Rightarrow \boxed{\alpha : \epsilon} * \boxed{\gamma : \epsilon}$, hence $\mathcal{P}_{\mathsf{send}}(1, n) * T \Rightarrow \mathsf{false}$ and therefore $\mathcal{P}_{\mathsf{send}}(1, n) * T \Rightarrow \mathcal{Q}(0, n)$ trivially.

So we have shown that we can establish the location invariant after the update. We still need to show that

$$\mathcal{P}_{\mathsf{keep}}(1, n) * A \Rightarrow tok.val \overset{1}{\mapsto} \_ * \boxed{\beta : 1}$$

If $v_1 + 1 = n$ then $\mathcal{P}_{\mathsf{keep}}(1, n) \Leftrightarrow tok.val \overset{1}{\mapsto} \_$ and we can conclude that $\mathcal{P}_{\mathsf{keep}}(1, n) * A \Rightarrow tok.val \overset{1}{\mapsto} \_ * \boxed{\beta : 1}$ where we used

$$tok.val \overset{1}{\mapsto} \_ * \left(tok.val \overset{1}{\mapsto} \_ * \boxed{\beta : 1 - \epsilon}\right) \Rightarrow \mathsf{false}$$

If $v_1 + 1 < n$ then $\mathcal{P}_{\mathsf{keep}}(1,n) \Leftrightarrow \boxed{\beta : \epsilon}$ and we can conclude that $\mathcal{P}_{\mathsf{keep}}(1,n) * A \Rightarrow \mathit{tok.val} \overset{1}{\mapsto} \_ * \boxed{\beta : 1}$ where we used

$$\boxed{\beta : \epsilon} * \boxed{\beta : 1} \Rightarrow \mathsf{false}$$

The remaining case $v_1 + 1 > n$ is slightly special. As argued before this case cannot occur. We can learn this fact by observing that

$$\mathcal{P}_{\mathsf{send}}(1,n) * \mathcal{Q}(1,t) \Rightarrow v_1 + 1 \leq n$$

since

$$\begin{aligned}
&\mathcal{P}_{\mathsf{send}}(1,n) \Rightarrow (v_1 + 1 > n \ ? \ \boxed{\alpha : \epsilon} * \boxed{\gamma : \epsilon} : \mathsf{true}) \\
&\mathcal{Q}(1,n) \Rightarrow \boxed{\gamma : 1} * \mathsf{true}
\end{aligned}$$

$v_1 + 1 \leq n$ is a pure assertion and hence we can instantiate φ in the basic CAS rule with it. Therefore we can learn this assertion in the postcondition of the CAS rule, which means in the case $v_1 + 1 > n$ we can conclude that $\mathcal{P}_{\mathsf{keep}}(1,n) * A \Rightarrow \mathit{tok.val} \overset{1}{\mapsto} \_ * \boxed{\beta : 1}$ holds trivially. This concludes the proof for case 3.

Case 4: $r > 0$ **and** $v > n$**.** We define

$$\begin{aligned}
&\mathcal{P}_{\mathsf{send}}(r,v) := \boxed{\alpha : \epsilon} \\
&\mathcal{P}_{\mathsf{keep}}(r,v) := \left[\begin{array}{ll} v_1 + 1 = n & ? \ \mathit{tok.val} \overset{1}{\mapsto} _ \\ & : \ \left(v_1 + 1 < n \ ? \ \boxed{\beta : \epsilon} \ : \ \boxed{\gamma : \epsilon} * \mathsf{true} \right) \end{array} \right]
\end{aligned}$$

We have

$$Q(r,v) = 0 \leq r \wedge r \leq v \wedge \boxed{\alpha : 1 - r \cdot \epsilon} * \boxed{\gamma : 1 - (v - n) \cdot \epsilon}$$

where we used that $[(r,v) \neq (0,n) \wedge (v \leq n)]$ does not hold. Choose $T := \mathcal{Q}(r,v)$ and $A := \mathsf{emp}$. We have

$$\begin{aligned}
\mathcal{P}_{\mathsf{send}}(r,v) * T \Rightarrow \ & 0 \leq r - 1 \wedge r - 1 \leq v \wedge \boxed{\alpha : 1 - (r-1) \cdot \epsilon} * \\
& \boxed{\gamma : 1 - (v - n) \cdot \epsilon} \\
\Rightarrow \ & \mathcal{Q}(r-1, v)
\end{aligned}$$

which concludes the proof for this case.

We have shown the fetch-and-add triple in all cases; this concludes the proof for the second read-modify-write operation.

3.2.8 Location invariant B

Location invariant B for the second proof is given by

$$\begin{aligned}\mathcal{Q}(r,v) := \exists w \in \{0,1\}.\, (0 \le r) \wedge (r \le v) \wedge (v < n \Rightarrow w = 0) \wedge \\ ([(r,v) \neq (0,n) \wedge (v \le n)]\ ?\ tok.val \overset{1-w}{\mapsto} \_ : \mathsf{emp}) * \\ \boxed{\delta : 1 - (r-w)\cdot\epsilon} * \boxed{\lambda : 1 - w}\end{aligned}$$

where δ and λ are ghost locations governed by the fractional permission structure and where ϵ is used to model the counting permission structure (see Section 2.4).

Note that this location invariant only has two ghost locations compared to the three ghost locations in location invariant A. It also has, in our opinion, an easier explanation. As we will see in the following sections the proof using this location invariant needs the CAS$-$*param* rule.

The idea of this location invariant is as follows. As in the location invariant A the existential w has actual value 1 if the nth thread has entered the `wait` function but has not yet left it yet. In that case there is no permission to the token $tok.val$ in the location invariant. The permission to $tok.val$ is also not in the location invariant if the barrier has finished its task, i.e. if $[(r,v) \neq (0,n) \wedge (v \le n)]$ holds (see Section 3.2.4 for a more detailed explanation).

Note that $w \le r$ is guaranteed by the location invariant in all cases. The ghost location δ tracks all the readers except the nth thread that entered wait, since in the proofs we make sure that the ith reader ($i \neq n$) gets $\boxed{\delta : \epsilon}$ after the read-modify-write operation. The ghost location λ tracks the nth thread that entered wait.

3.2.9 Proof outline (B)

The specification that we prove with respect to location invariant B is the same as in Section 3.2.3. The proof outline is given in Figure 10. As in Section 3.2.5 we do not consider cases where (r,v) is read and $r < 0$ or $v < 0$ since these cases cannot occur, which can be directly seen from the location invariant. We also do not carry around $\mathsf{U}(\mathtt{VRC}, \mathcal{Q})$ explicitly since it is duplicable and available in the precondition of `wait`.

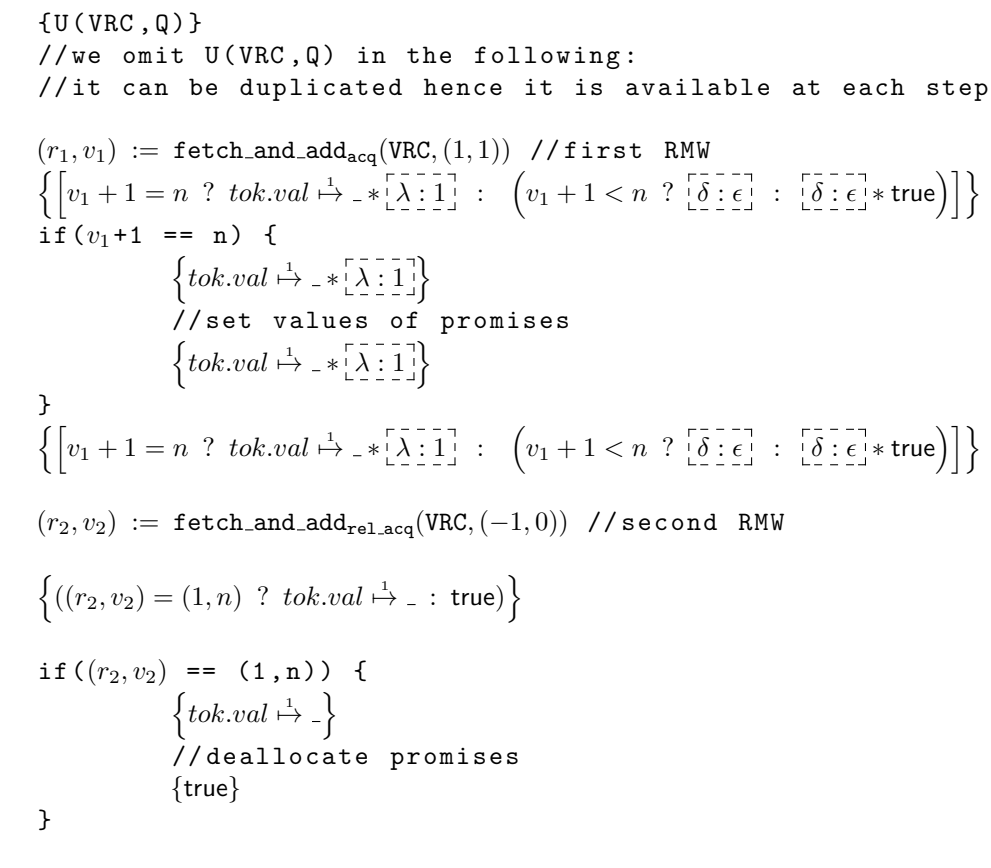

Figure 10: Proof outline for `wait` *in Barrier using location invariant B.*

3.2.10 Proof of first RMW (B)

We verify the first read-modify-write operation given in the proof outline in Figure 10 using location invariant B (Section 3.2.8). We have to show

$$\begin{array}{l}\{\mathsf{U}(\mathtt{VRC}, \mathcal{Q})\} \\ \quad (r_1, v_1) := \mathtt{fetch\_and\_add}_{\mathsf{acq}}(\mathtt{VRC}, (1,1)) \\ \left\{\left[\begin{array}{l} v_1 + 1 = n \ ? \\ \mathit{tok.val} \overset{1}{\mapsto} \_ * \boxed{\lambda : 1} \ : \ \left(v_1 + 1 < n \ ? \ \boxed{\delta : \epsilon} \ : \ \boxed{\delta : \epsilon} * \mathsf{true}\right) \end{array}\right]\right\}\end{array}$$

using the fetch-and-add rule in Figure 4. We have $P \equiv \mathsf{emp}$ and hence $\mathcal{P}_{\mathsf{send}}(t) := \mathcal{P}_{\mathsf{keep}}(t) := \mathsf{emp}$. We now show the CAS triple in the premise of the fetch-and-add rule for all possible values (r, v) using the CAS$-$*basic* rule. Assume we read an arbitrary value (r, v). We make a case distinction on (r, v).

Case $v + 1 < n$**.** We have

$$\mathcal{Q}(r, v) \Leftrightarrow (0 \leq r) \land (r \leq v) \land \mathit{tok.val} \overset{1}{\mapsto} \_ * \boxed{\delta : 1 - r \cdot \epsilon} * \boxed{\lambda : 1}$$

We choose

$$\begin{array}{l} T := (0 \leq r) \land (r \leq v) \land \mathit{tok.val} \overset{1}{\mapsto} \_ * \boxed{\delta : 1 - (r+1) \cdot \epsilon} * \boxed{\lambda : 1} \\ A := \boxed{\delta : \epsilon} \end{array}$$

and it holds that $\mathcal{Q}(r, v) \Leftrightarrow A * T$. We have

$$\begin{array}{l} T \Rightarrow (0 \leq r+1) \land (r+1 \leq v+1) \land \mathit{tok.val} \overset{1}{\mapsto} \_ * \boxed{\delta : 1 - (r+1) \cdot \epsilon} * \boxed{\lambda : 1} \\ \quad \Rightarrow \mathcal{Q}(r+1, v+1) \end{array}$$

Furthermore, A is exactly the permission needed in this case and since A only consists of ghost state we need not worry about modalities. This concludes this case.

Case $v + 1 = n$**.** We have

$$\mathcal{Q}(r, v) \Leftrightarrow (0 \leq r) \land (r \leq v) \land \mathit{tok.val} \overset{1}{\mapsto} \_ * \boxed{\delta : 1 - r \cdot \epsilon} * \boxed{\lambda : 1}$$

We choose

$$\begin{array}{l} T := (0 \leq r) \land (r \leq v) \land \boxed{\delta : 1 - r \cdot \epsilon} \\ A := \mathit{tok.val} \overset{1}{\mapsto} \_ * \boxed{\lambda : 1} \end{array}$$

and it holds that $\mathcal{Q}(r,v) \Leftrightarrow A * T$. We have

$$\begin{aligned} T \Rightarrow_{\text{choosing } w:=1} \; & \exists w \in \{0,1\}.\, (0 \le r+1) \wedge (r+1 \le v+1) \wedge \\ & (v+1 < n \Rightarrow w = 0) * tok.val \overset{1-w}{\mapsto} \_ * \\ & \boxed{\delta : 1 - ((r+1) - w) \cdot \epsilon} * \boxed{\lambda : 1 - w} \\ \Rightarrow \; & \mathcal{Q}(r+1, v+1) \end{aligned}$$

Furthermore, A is exactly the permission needed in this case and since the fetch-and-add operation is of type `acq` we get A directly. This concludes this case.

Case $v + 1 > n$**.** We have

$$\begin{aligned} \mathcal{Q}(r,v) \Leftrightarrow & \exists w \in \{0,1\}.\, (0 \le r) \wedge (r \le v) \wedge \\ & ([(r,v) \neq (0,n) \wedge (v \le n)] \; ? \; tok.val \overset{1-w}{\mapsto} \_ : \mathsf{emp}) * \\ & \boxed{\delta : 1 - (r - w) \cdot \epsilon} * \boxed{\lambda : 1 - w} \end{aligned}$$

and we choose

$$\begin{aligned} T := & \exists w \in \{0,1\}.\, (0 \le r) \wedge (r \le v) \wedge \\ & \boxed{\delta : 1 - ((r+1) - w) \cdot \epsilon} * \boxed{\lambda : 1 - w} \\ A := & \boxed{\delta : \epsilon} * \mathsf{true} \end{aligned}$$

where it holds that $\mathcal{Q}(r,v) \Rightarrow A * T$. Note that if $v = n$ then the permission potentially guarded by $[(r,v) \neq (0,n) \wedge (v \le n)]$ is absorbed by true in A. We have

$$\begin{aligned} T \Rightarrow \; & \exists w \in \{0,1\}.\, (0 \le r+1) \wedge (r+1 \le v+1) \wedge \\ & \boxed{\delta : 1 - ((r+1) - w) \cdot \epsilon} * \boxed{\lambda : 1 - w} \\ \Rightarrow \; & Q(r+1, v+1) \end{aligned}$$

Here we use that $[(r+1, v+1) \neq (0,n) \wedge (v+1 \le n)]$ does not hold. Furthermore, A is exactly the permission we want and and since the fetch-and-add operation is of type `acq` we get A directly. This concludes this case.

3.2.11 Proof of second RMW (B)

We verify the second read-modify-write operation given in the proof outline in Figure 9 using location invariant B. We have to show

$$\left\{ \mathsf{U}(\texttt{VRC}, \mathcal{Q}) * \left[\begin{array}{ll} v_1 + 1 = n & ?\ tok.val \overset{1}{\mapsto} \_ * \boxed{\lambda : 1} \\ & :\ \left(v_1 + 1 < n\ ?\ \boxed{\delta : \epsilon}\ :\ \boxed{\delta : \epsilon} * \mathsf{true} \right) \end{array} \right] \right\}$$
$$(r_2, v_2) := \texttt{fetch\_and\_add}_{\texttt{rel\_acq}}(\texttt{VRC}, (-1, 0))$$
$$\left\{ ((r_2, v_2) = (1, n)\ ?\ tok.val \overset{1}{\mapsto} \_ : \mathsf{true}) \right\}$$

We have

$$P \equiv \left[\begin{array}{ll} v_1 + 1 = n & ?\ tok.val \overset{1}{\mapsto} \_ * \boxed{\lambda : 1} \\ & :\ \left(v_1 + 1 < n\ ?\ \boxed{\delta : \epsilon}\ :\ \boxed{\delta : \epsilon} * \mathsf{true} \right) \end{array} \right]$$

Instead of writing down $\mathcal{P}_{\mathsf{send}}(r, v)$ and $\mathcal{P}_{\mathsf{keep}}(r, v)$ for general (r, v) we will give the definitions in the different (disjoint) subcases for the sake of presentation. We make a case distinction on the value (r, v) that is read to verify the CAS triple in the premise of the fetch-and-add rule.

Case 1: $r = 0$**.** We show that $(0, v)$ cannot be read by verifying the CAS triple using the CAS$-\bot$ rule. Define in this case

$$\mathcal{P}_{\mathsf{send}}(0, v) := P$$
$$\mathcal{P}_{\mathsf{keep}}(0, v) := \mathsf{emp}$$

We have

$$\mathcal{P}_{\mathsf{send}}(0, v) \Rightarrow (v_1 + 1 = n\ ?\ tok.val \overset{1}{\mapsto} \_ * \boxed{\lambda : 1} : \boxed{\delta : \epsilon} * \mathsf{true})$$

and

$$\mathcal{Q}(0, v) \Rightarrow \boxed{\delta : 1} * \boxed{\lambda : 1}$$

where we used that the existential w in the location invariant must have actual value 0 if $r = 0$ since otherwise the location invariant is inconsistent. Hence $\mathcal{P}_{\mathsf{send}}(0, v) * \mathcal{Q}(0, v) \Rightarrow \mathsf{false}$ and therefore we can prove the CAS triple using the CAS$-\bot$ rule. This concludes the case $r = 0$.

Case 2: $(r, v) = (1, n)$**.** We verify this case using the CAS$-$*param* rule. This is the only case where we use the CAS$-$*param* rule using location invariant B. Define in this case

$$\mathcal{P}_{\mathsf{send}}(1, n) := \left[v_1 + 1 = n\ ?\ \boxed{\lambda : 1}\ :\ \left(v_1 + 1 < n\ ?\ \boxed{\delta : \epsilon}\ :\ \boxed{\delta : \epsilon} \right) \right]$$
$$\mathcal{P}_{\mathsf{keep}}(1, n) := \left[v_1 + 1 = n\ ?\ tok.val \overset{1}{\mapsto} \_\ :\ (v_1 + 1 < n\ ?\ \mathsf{emp}\ :\ \mathsf{true}) \right]$$

It holds that $P \Leftrightarrow \mathcal{P}_{\mathsf{send}}(1,n) * \mathcal{P}_{\mathsf{keep}}(1,n)$. We have

$$\begin{aligned}\mathcal{Q}(1,n) \Leftrightarrow & \exists w \in \{0,1\}.\, tok.val \overset{1-w}{\mapsto} \_ * \boxed{\delta : 1-(1-w)\cdot\epsilon} * \boxed{\lambda : 1-w} \\ \Leftrightarrow & \exists w \in \{0,1\}.\ \begin{array}{l}(w = 1\ ? \\ \quad \boxed{\delta : 1} * \boxed{\lambda : 0} : tok.val \overset{1}{\mapsto} \_ * \boxed{\delta : 1-\epsilon} * \boxed{\lambda : 1})\end{array}\end{aligned}$$

Define

$$\begin{aligned}\mathcal{T}(z) &:= (z = 1\ ?\ \boxed{\delta : 1} * \boxed{\lambda : 0} : \boxed{\delta : 1-\epsilon} * \boxed{\lambda : 1} \wedge (z = 0)) \\ \mathcal{A}(z) &:= (z = 1\ ?\ \mathsf{emp} : tok.val \overset{1}{\mapsto} \_)\end{aligned}$$

It holds that $\mathcal{Q}(1,n) \Rightarrow \exists z.\, \mathcal{A}(z) * \mathcal{T}(z)$ (the existential z corresponds directly to the existential w in the location invariant). We need to show that

$$\forall z.(\mathcal{P}_{\mathsf{send}}(1,n) * \mathcal{T}(z) \Rightarrow \mathcal{Q}(0,n) \wedge \varphi(z))$$

holds, where we define

$$\varphi(z) := (v_1 + 1 = n\ ?\ z = 1 : z = 0)$$

Note that $\varphi(z)$ is pure for every z.

Since $\mathcal{P}_{\mathsf{send}}(r,v)$ depends on the value (r_1, v_1) read in the first read-modify write operation, it is cumbersome to show this implication directly covering all cases at once. Therefore we make a case distinction on (r_1, v_1) and show that in each case $\forall z.(\mathcal{P}_{\mathsf{send}}(1,n) * \mathcal{T}(z) \Rightarrow \mathcal{Q}(0,n) \wedge \varphi(z))$ holds.

First, assume $v_1 + 1 = n$. We have

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(1,n) &:= \boxed{\lambda : 1} \\ \varphi(z) &:= (z = 1)\end{aligned}$$

Let z be arbitrary. We have

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(1,n) * \mathcal{T}(z) \Rightarrow & \begin{array}{l}(z = 1\ \ ?\ \boxed{\delta : 1} * \boxed{\lambda : 1} \\ \qquad :\ (z = 0) \wedge \boxed{\delta : 1-\epsilon} * \boxed{\lambda : 1} * \boxed{\lambda : 1})\end{array} \\ & \Rightarrow (z = 1\ ?\ \mathcal{Q}(0,n) : \mathsf{false}) \\ & \Rightarrow (z = 1\ ?\ \mathcal{Q}(0,n) \wedge (z = 1) : \mathcal{Q}(0,n) \wedge (z = 1))\end{aligned}$$

which concludes the subcase $v_1 + 1 = n$.

Next, assume $v_1 + 1 \neq n$. We have

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(1,n) &:= \boxed{\delta : \epsilon} \\ \varphi(z) &:= (z = 0)\end{aligned}$$

Let z be arbitrary. We have

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(1,n) * \mathcal{T}(z) \Rightarrow\ & (z = 1 \ ?\ \boxed{\delta : 1} * \boxed{\lambda : 0} * \boxed{\delta : \epsilon} \\ & \quad :\ (z = 0) \wedge \boxed{\delta : 1 - \epsilon} * \boxed{\lambda : 1} * \boxed{\delta : \epsilon}) \\ & \Rightarrow (z = 1 \ ?\ \mathsf{false} : \mathcal{Q}(0,n) \wedge (z = 0)) \\ & \Rightarrow (z = 1 \ ?\ \mathcal{Q}(0,n) \wedge (z = 0) : \mathcal{Q}(0,n) \wedge (z = 0))\end{aligned}$$

which concludes the subcase $v_1 + 1 \neq n$.

So we have now proved in all cases that we can reestablish the location invariant. We still need to prove

$$\mathcal{P}_{\mathsf{keep}}(1,n) * (\exists z.\, \mathcal{A}(z) \wedge \varphi(z)) \Rightarrow tok.val \overset{1}{\mapsto} \_$$

We again consider different subcases (dependent on v_1). First, assume $v_1 + 1 = n$. Then we have $\varphi(z) \equiv (z = 1)$ and therefore $\mathcal{A}(z) \equiv \mathsf{emp}$. Since $\mathcal{P}_{\mathsf{keep}}(1,n) \equiv tok.val \overset{1}{\mapsto} \_$ in this case we are done.

Next, assume $v_1 + 1 < n$. Then we have $\varphi(z) \equiv (z = 0)$ and therefore $\mathcal{A}(z) \equiv tok.val \overset{1}{\mapsto} \_$. Since additionally $\mathcal{P}_{\mathsf{keep}}(1,n) \equiv \mathsf{emp}$ in this case, we are done.

Finally, assume $v_1 + 1 > n$. Then we have $\varphi(z) \equiv (z = 0)$ and therefore $\mathcal{A}(z) \equiv tok.val \overset{1}{\mapsto} \_$. But we have $\mathcal{P}_{\mathsf{keep}}(1,n) \equiv \mathsf{true}$. So we can only show $tok.val \overset{1}{\mapsto} \_ * \mathsf{true}$ which is not exactly what the specification demands.

We can fix this though. If one looks closely then the permission absorbed in true (this absorption occurs right after the first fetch-and-add) is

$$([(r_1, v_1) \neq (0,n) \wedge (v_1 \leq n)] \ ?\ tok.val \overset{\cdot}{\mapsto} \quad \mathsf{emp})$$

So the actual permission we have after the second fetch-and-add operation in this case is given by

$$tok.val \overset{1}{\mapsto} \_ * ([(r_1, v_1) \neq (0,n) * (v_1 \leq n)] \ ?\ tok.val \overset{1}{\mapsto} \_ : \mathsf{emp})$$

which implies $tok.val \overset{1}{\mapsto} \_$ (since $tok.val \overset{1}{\mapsto} \_ * tok.val \overset{1}{\mapsto} \_ \Rightarrow \mathsf{false}$). So the conclusion is that if we adjust the proof outline to carry along all the permission explicitly (without using true) then we can satisfy the specification. This concludes this case.

Case 3: $r > 0, v \leq n$ **and** $(r,v) \neq (1,n)$**.** In this case we verify the CAS-triple using the CAS$-$*basic* rule. Choose

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(r,v) &:= (v_1 + 1 \leq n \ ?\ P : \boxed{\delta : \epsilon}) \\ \mathcal{P}_{\mathsf{keep}}(r,v) &:= (v_1 + 1 \leq n \ ?\ \mathsf{emp} : \mathsf{true})\end{aligned}$$

It holds that $P \Leftrightarrow \mathcal{P}_{\mathsf{send}}(r,v) * \mathcal{P}_{\mathsf{keep}}(r,v)$. We have

$$\begin{aligned}\mathcal{Q}(r,v) \Leftrightarrow & \exists w \in \{0,1\}.\, (0 \leq r) \wedge (r \leq v) \wedge (v < n \Rightarrow w = 0) \wedge \\ & \mathit{tok.val} \overset{1-w}{\mapsto} \_ * \boxed{\delta : 1 - (r-w)\cdot\epsilon} * \boxed{\lambda : 1 - w}\end{aligned}$$

where we used that $[(r,v) \neq (0,n) \wedge (v \leq n)]$ holds. Choose

$$\begin{aligned}T &:= \mathcal{Q}(r,v) \\ A &:= \mathsf{emp}\end{aligned}$$

We need to show $\mathcal{P}_{\mathsf{send}}(r,v) * T \Rightarrow \mathcal{Q}(r-1,v)$. We do this by considering different subcases dependent on v_1. We use that $[(r-1,v) \neq (0,n) \wedge (v \leq n)]$ holds.

First, assume $v_1 + 1 = n$. Hence $\mathcal{P}_{\mathsf{send}}(r,v) \equiv \mathit{tok.val} \overset{1}{\mapsto} \_ * \boxed{\lambda : 1}$. We have

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(r,v) * T \Rightarrow & \begin{array}{l}(0 \leq r-1) \wedge (r-1 \leq v) \wedge (v < n \Rightarrow w = 0) \wedge \\ \mathit{tok.val} \overset{1}{\mapsto} \_ * \boxed{\delta : 1 - (r-1)\cdot\epsilon} * \boxed{\lambda : 1}\end{array} \\ & \Rightarrow_{\text{choosing } w:=0} \mathcal{Q}(r-1,v)\end{aligned}$$

where we used that the existentially quantified variable w in T must be 1. This concludes this case.

Next, assume $v_1 + 1 \neq n$. Hence $\mathcal{P}_{\mathsf{send}}(r,v) \equiv \boxed{\delta : \epsilon}$. We have

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(r,v) * T \Rightarrow & \begin{array}{l}\exists w \in \{0,1\}.\, (0 \leq r-1) \wedge (r-1 \leq v) \wedge (v < n \Rightarrow w = 0) \wedge \\ \mathit{tok.val} \overset{1-w}{\mapsto} \_ * \boxed{\delta : 1 - ((r-1)-w)\cdot\epsilon} * \boxed{\lambda : 1 - w}\end{array} \\ & \Rightarrow \mathcal{Q}(r-1,v)\end{aligned}$$

This concludes case 3.

Case 4: $r > 0, v > n$**.** In this case we verify the CAS-triple using the CAS$-$*basic* rule. We use that $[(r,v) \neq (0,n) \wedge (v \leq n)]$ and $[(r-1,v) \neq (0,n) \wedge (v \leq n)]$ both do not hold.

Choose

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(r,v) &:= \left[\begin{array}{l}(v_1 + 1 = n \ ? \\ \boxed{\lambda : 1} : \\ (v_1 + 1 < n \ ? \ \boxed{\delta : \epsilon} \ : \ \boxed{\delta : \epsilon}))\end{array}\right] \\ \mathcal{P}_{\mathsf{keep}}(r,v) &:= \left[\begin{array}{l}(v_1 + 1 = n \ ? \\ \mathit{tok.val} \overset{1}{\mapsto} \_ : \\ (v_1 + 1 < n \ ? \ \mathsf{emp} \ : \ \mathsf{true}))\end{array}\right]\end{aligned}$$

It holds that $P \Leftrightarrow \mathcal{P}_{\mathsf{send}}(r, v) * \mathcal{P}_{\mathsf{keep}}(r, v)$. We have

$$\mathcal{Q}(r, v) \Leftrightarrow \exists w \in \{0, 1\}.\, (0 \leq r) \land (r \leq v) \land \boxed{\delta : 1 - (r - w) \cdot \epsilon} * \boxed{\lambda : 1 - w}$$

Choose

$$T := \mathcal{Q}(r, v)$$
$$A := \mathsf{emp}$$

If $v_1 + 1 = n$ then we have $\mathcal{P}_{\mathsf{send}}(r, v) \Leftrightarrow \boxed{\lambda : 1}$, hence

$$\begin{aligned} \mathcal{P}_{\mathsf{send}}(r, v) * T &\Rightarrow (0 \leq r - 1) \land (r - 1 \leq v) \land \boxed{\delta : 1 - (r - 1) \cdot \epsilon} * \boxed{\lambda : 1} \\ &\Rightarrow_{\text{choosing } w:=0} \mathcal{Q}(r - 1, v) \end{aligned}$$

where we used that the existentially quantified variable w in T must be 1. If $v_1 + 1 \neq n$ then we have $\mathcal{P}_{\mathsf{send}}(r, v) \Leftrightarrow \boxed{\delta : \epsilon}$, hence

$$\begin{aligned} \mathcal{P}_{\mathsf{send}}(r, v) * T \Rightarrow\ & \exists w \in \{0, 1\}.\, (0 \leq r - 1) \land (r - 1 \leq v) \land \\ & \boxed{\delta : 1 - ((r - 1) - w) \cdot \epsilon} * \boxed{\lambda : 1 - w} \\ \Rightarrow\ & \mathcal{Q}(r - 1, v) \end{aligned}$$

This concludes the proof for case 4.

3.3 Rust atomic reference counter

The Rust atomic reference counter [2] (we will refer to it as *ARC*) allows managing multiple threads which have read access to some shared data. The implementation makes sure that the shared data is deallocated once no thread has access to it. We present the FSL++ proof given in [9] for the *ARC* example.

3.3.1 Implementation

The implementation for the *ARC* example is given in Figure 11 and is taken directly from [9].

`new(v)` allocates an atomic reference counter where the data guarded by the counter is stored at the non-atomic location `a.data` which is initialized to `v`. The atomic location `a.count` is initialized to 1 and counts the number of references to `a.data`. Conceptually, `new(v)` returns an ARC resource which can be used to access `a.data`, i.e. one ARC resource held by a thread symbolizes one reference to `a.data`.

```
ref new(v) {
  a := alloc()
  [a.data]_na := v
  [a.count]_rlx := 1
  return a
}

T read(a) {
  return [a.data]_na
}

void clone(a) {
  x := fetch_and_add_rlx(a.count, 1)
}

void drop(a) {
  y := fetch_and_add_rel(a.count, −1)
  if(y == 1) {
    fence_acq
    free(a)
  }
}
```

Figure 11: Implementation for ARC example. T is the type of `a.data`.

`clone` increments `a.count` by 1 and conceptually creates another ARC resource. This is only called by a function that already has one ARC resource. A thread which calls `clone` may give one ARC resource to another thread while keeping one ARC resource to itself.

Function `read` is used to read `a.data`. This is only called by threads that have access to an ARC resource.

`drop` is called once a thread does not need its ARC resource any more, hence `a.count` is decremented by 1. If it turns out that after decrementing `a.count` by 1 the value at `a.count` is 0 then no thread should have an ARC resource and no thread should ever get an ARC resource in the future (this is what we will prove). So the memory associated with the atomic reference counter can be deallocated (which is done using the `free(a)` instruction).

3.3.2 Location invariant

The location invariant for the atomic location `a.count` (which is taken from [9]) is given by:

$$\begin{aligned}\mathcal{Q}_{a,v}(c) := \ & \textbf{if } c = 0 \textbf{ then } \boxed{\gamma : 0^-} * \boxed{\delta : 0^-} \\ & \textbf{else } \exists f \in \mathbb{Q} \cap [0,1].\ \texttt{a.data} \overset{f}{\mapsto} v * \boxed{\gamma : (c-1+f)^-} * \\ & \qquad \boxed{\delta : (1-f)^-}\end{aligned}$$

where γ and δ are ghost locations which are governed by the *SFC* permission structure introduced in Section 2.4. Recall that in the *SFC* permission structure the set of elements is given by $\mathcal{Q} \times \{+, -\}$. An element a^- is a source permission (i.e. it can only appear once per ghost location) while an element a^+ is an access permission and it can appear multiple times. In the location invariant we only have source permissions because it can be used to track the access permissions held by the threads.

We note that in [9] the partial commutative monoid governing the ghost locations is presented differently than the *SFC* permission structure introduced in [5]. We observed that these two structures are equivalent.

The idea of the location invariant is as follows. Every time a thread calls `drop` it gives up its share of the permission to `a.data`. The existentially quantified variable f counts how much permission to `a.data` has been dropped in total. Hence when the atomic reference counter is allocated (i.e. $c = 1$) then the existentially quantified variable f has actual value 0 since `drop` has never been called.

If the value of the counter is $c > 0$ then there are c references to `a.data`. The ghost location γ inside the location invariant is used to count these. We

have $\boxed{\gamma : (c - 1 + f)^-}$. The reason the value of the ghost resource in γ is not only dependent on the number of references c but also on the amount of permission f dropped to `a.data` is mainly due to the scenario when the last thread holding permission to `a.data` calls `drop`.

When the last thread calls `drop` the value of the counter goes from 1 to 0. To be able to deallocate the counter the thread needs the full permission to `a.data`. We need to somehow be able to verify that the permission that the thread has together with the permission that is in the invariant equals the full permission. Including the f in the γ resource facilitates this (as we will see later).

The δ ghost location is used to be able to verify that a thread that has access to one reference of `a.data` cannot read the value 0 from the counter. Also it is used for the scenario when the last thread calls `drop`. Note how in the γ ghost location f appears in a positive form while in the δ ghost location f appears in a negative form. This "correlation" between γ and δ is important for the verification of the `drop` method, as we will see.

If $c = 0$ then no thread should hold any access permission to γ or δ, this is guaranteed by $\boxed{\gamma : 0^-} * \boxed{\delta : 0^-}$.

3.3.3 Specification

The specification (taken from [9]) is given by

$$\begin{aligned}
&\{\mathsf{emp}\}\texttt{new(v)}\ \{a.\mathsf{ARC}_{\gamma,\delta}(a, v)\}\\
&\{\mathsf{ARC}_{\gamma,\delta}(a, v)\}\texttt{read(a)}\ \{\mathsf{ARC}_{\gamma,\delta}(a, v)\}\\
&\{\mathsf{ARC}_{\gamma,\delta}(a, v)\}\texttt{clone(a)}\ \{\mathsf{ARC}_{\gamma,\delta}(a, v) * \mathsf{ARC}_{\gamma,\delta}(a, v)\}\\
&\{\mathsf{ARC}_{\gamma,\delta}(a, v)\}\texttt{drop(a)}\ \{\mathsf{emp}\}
\end{aligned}$$

where the *ARC predicate* $\mathsf{ARC}_{\gamma,\delta}(a, v)$ is defined as:

$$\begin{aligned}
\mathsf{ARC}_{\gamma,\delta}(a, v) :=& \mathsf{U}(\texttt{a.count}, \mathcal{Q}_{a,v}) *\\
&\exists q \in \mathbb{Q} \cap (0, 1].\, \texttt{a.data} \overset{q}{\mapsto} v * \boxed{\gamma : (1 - q)^+} * \boxed{\delta : q^+}
\end{aligned}$$

Note that the fraction q appears in negative form in the γ ghost location and in positive form in the δ ghost location. This is exactly the opposite situation compared to the existentially quantified variable f in the location invariant. This is important for the verification of the `drop` method.

We now give the proofs from [9] for the `clone` and `drop` methods since they are the main reason for the setup of the location invariant. The proofs for the `new` and `read` methods are fairly straight-forward and can be looked up in [9].

3.3.4 Proof of function `clone`

We provide the proof for `clone` (taken from [9]). We need to show

$$\{\mathsf{ARC}_{\gamma,\delta}(a,v)\}\ \mathtt{x} := \mathtt{fetch\_and\_add}_{\mathtt{rlx}}(\mathtt{a.count},1)\left\{\begin{array}{l}\mathsf{ARC}_{\gamma,\delta}(a,v) * \\ \mathsf{ARC}_{\gamma,\delta}(a,v)\end{array}\right\}$$

using the fetch-and-add rule. Note that the fetch-and-add has access type `rlx` and there is no fence instruction inside `clone`. Hence the only way to get two ARC predicates from a single one is by transferring ghost state.

We have

$$P \equiv \exists q \in (0,1].\, \mathtt{a.data} \overset{q}{\mapsto} v * \boxed{\gamma : (1-q)^+} * \boxed{\delta : q^+}$$

and we choose

$$\mathcal{P}_{\mathsf{send}}(t) := (t = 0\ ?\ P : \mathsf{emp})$$
$$\mathcal{P}_{\mathsf{keep}}(t) := (t = 0\ ?\ \mathsf{emp} : P)$$

Clearly $P \Leftrightarrow \mathcal{P}_{\mathsf{send}}(t) * \mathcal{P}_{\mathsf{keep}}(t)$ holds for all t. We still need to prove the CAS triple in the premise of the fetch-and-add rule for every read value t.

Assume $t = 0$. We show that we cannot read this value (intuitively, since we have a reference to `a.data`). We have $\mathcal{Q}(0) \Leftrightarrow \boxed{\gamma : 0^-} * \boxed{\delta : 0^-}$ and $\mathcal{P}_{\mathsf{send}}(0) \Rightarrow \boxed{\delta : q^+} * \mathsf{true}$ for some $q > 0$. Since $0^- \oplus q^+$ is undefined for $q > 0$ we conclude $\mathcal{P}_{\mathsf{send}}(0) * \mathcal{Q}(0) \Rightarrow \mathsf{false}$. Therefore we can use the CAS$-\bot$ rule to prove the CAS triple, which shows 0 cannot be read.

Next, assume $t > 0$. In this case we show the CAS triple using the CAS$-$*basic* rule. We have

$$\begin{aligned}\mathcal{Q}(t) &\Leftrightarrow \exists f \in \mathbb{Q} \cap [0,1].\, \mathtt{a.data} \overset{f}{\mapsto} v * \boxed{\gamma : (t-1+f)^-} * \boxed{\delta : (1-f)^-} \\ &\Leftrightarrow \exists f \in \mathbb{Q} \cap [0,1].\, \mathtt{a.data} \overset{f}{\mapsto} v * \boxed{\gamma : ((t+1)-1+f)^-} * \boxed{\gamma : 1^+} * \\ &\quad \boxed{\delta : (1-f)^-} \\ &\Leftrightarrow \mathcal{Q}(t+1) * \boxed{\gamma : 1^+}\end{aligned}$$

where we used that

$$(t-1+f)^- = ((t+1)-1+f-1)^- = ((t+1)-1+f)^- \oplus 1^+$$

So we can choose $T := \mathcal{Q}(t+1)$ and $A := \boxed{\gamma : 1^+}$. This verifies the CAS triple where the thread receives $\boxed{\gamma : 1^+}$ (since it is ghost state we need not worry about modalities).

$\{\mathsf{ARC}_{\gamma,\delta}(a,v)\}$
`y :=` $\texttt{fetch\_and\_add}_{\texttt{rel}}(\texttt{a.count}, -1,)$
$\left\{(y = 1 \ ?\ \texttt{a.data} \overset{q}{\mapsto} v * \triangledown\texttt{a.data} \overset{1-q}{\mapsto} v \ :\ \mathsf{emp})\right\}$ *for some* q
`if(y == 1) {`
  $\left\{\texttt{a.data} \overset{q}{\mapsto} v * \triangledown\texttt{a.data} \overset{1-q}{\mapsto} v\right\}$
  $\texttt{fence}_{\mathsf{acq}}$
  $\left\{\texttt{a.data} \overset{1}{\mapsto} v\right\}$
  `free(a)`
  $\{\mathsf{emp}\}$
`}`
$\{\mathsf{emp}\}$

Figure 12: Proof outline for function **`drop`** *in ARC.*

After the fetch-and-add the thread has resources (note that $\mathcal{P}_{\mathsf{keep}}(t) \Leftrightarrow P$ for $t > 0$).

$$
\begin{array}{l}
\mathcal{P}_{\mathsf{keep}}(t) * A \Leftrightarrow \\
\exists q \in \mathbb{Q} \cap (0,1].\, \texttt{a.data} \overset{q}{\mapsto} v * \boxed{\gamma : (1-q)^+} * \boxed{\delta : q^+} * \boxed{\gamma : 1^+} \ \Leftrightarrow \\
\exists q \in \mathbb{Q} \cap (0,1].\, \texttt{a.data} \overset{q}{\mapsto} v * \boxed{\gamma : (2-q)^+} * \boxed{\delta : q^+} \Leftrightarrow \\
\exists q \in \mathbb{Q} \cap (0,1].\, \texttt{a.data} \overset{\frac{q}{2}}{\mapsto} v * \boxed{\gamma : (1-\frac{q}{2})^+} * \boxed{\delta : \frac{q}{2}^+} * \\
\qquad\qquad \texttt{a.data} \overset{\frac{q}{2}}{\mapsto} v * \boxed{\gamma : (1-\frac{q}{2})^+} * \boxed{\delta : \frac{q}{2}^+} \Rightarrow_{\text{choosing } q' := \frac{q}{2}, q'' := \frac{q}{2}} \\
\exists q' \in \mathbb{Q} \cap (0,1].\, \texttt{a.data} \overset{q'}{\mapsto} v * \boxed{\gamma : (1-q')^+} * \boxed{\delta : q'^+} * \\
\qquad\qquad \exists q'' \in \mathbb{Q} \cap (0,1].\, \texttt{a.data} \overset{q''}{\mapsto} v * \boxed{\gamma : (1-q'')^+} * \boxed{\delta : q^+}
\end{array}
$$

If we also carry along $\mathsf{U}(\texttt{a.count}, \mathcal{Q}_{a,v})$ (which is in the precondition and is duplicable) then we can show that

$$\mathcal{P}_{\mathsf{keep}}(t) * A \Rightarrow \mathsf{ARC}_{\gamma,\delta}(a,v) * \mathsf{ARC}_{\gamma,\delta}(a,v)$$

This concludes the proof for `clone`.

3.3.5 Proof of function `drop`

We provide the proof for `drop` (taken from [9]). The proof outline for `drop` is given in Figure 12. The only interesting part in the proof outline is the

fetch-and-add, which we verify now. We have

$$P \equiv \texttt{a.data} \overset{q}{\mapsto} v * \boxed{\gamma : (1-q)^+} * \boxed{\delta : q^+}$$

for some $q \in (0, 1]$. We choose

$$\mathcal{P}_{\mathsf{send}}(t) := (t = 1 \ ? \ \boxed{\gamma : (1-q)^+} * \boxed{\delta : q^+} : P)$$
$$\mathcal{P}_{\mathsf{keep}}(t) := (t = 1 \ ? \ \texttt{a.data} \overset{q}{\mapsto} v : \mathsf{emp})$$

It holds that $P \Leftrightarrow \mathcal{P}_{\mathsf{send}}(t) * \mathcal{P}_{\mathsf{keep}}(t)$. We still need to prove the CAS triple in the premise of the fetch-and-add rule. Analogously to the proof for `clone` we can show that $t = 0$ is never read.

Assume $t > 1$ is read. In this case we want to give away all our permission and extract nothing from the location invariant. We prove the CAS triple in the premise of the fetch-and-add rule using the CAS$-$*basic* rule. We have

$$\mathcal{Q}(t) \Leftrightarrow \exists f' \in \mathbb{Q} \cap [0,1].\, \texttt{a.data} \overset{f'}{\mapsto} v * \boxed{\gamma : (t-1+f')^-} * \boxed{\delta : (1-f')^-}$$

We choose $T := \mathcal{Q}(t)$ and $A := \mathsf{emp}$. We have $\mathcal{P}_{\mathsf{send}}(t) \equiv P$. Furthermore:

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(t) * \mathcal{Q}(t) \Rightarrow \ & \exists f' \in \mathbb{Q} \cap [0,1].\, \texttt{a.data} \overset{f'+q}{\mapsto} v * \\ & \boxed{\gamma : ((t-1)-1+(f'+q))^-} * \boxed{\delta : (1-(f'+q))^-} \\ & \Rightarrow_{\text{choosing } f := f'+q} \mathcal{Q}(t-1)\end{aligned}$$

which concludes the proof for $t > 1$ (note that since the fetch-and-add is of access type `rel` we can give away the permission to `a.data`).

Next, assume $t = 1$ is read. In this case we want to extract all the permission to `a.data` from the location invariant and give away all the ghost permission. We use the CAS$-$*param* rule given in Figure 2 to verify the CAS triple in the premise of the fetch-and-add rule. We have

$$\mathcal{P}_{\mathsf{send}}(1) := \boxed{\gamma : (1-q)^+} * \boxed{\delta : q^+}$$
$$\mathcal{P}_{\mathsf{keep}}(1) := \texttt{a.data} \overset{q}{\mapsto} v$$

We choose

$$\mathcal{T}(z) := \boxed{\gamma : z^-} * \boxed{\delta : (1-z)^-}$$
$$\mathcal{A}(z) = \texttt{a.data} \overset{z}{\mapsto} v$$

We have $\mathcal{Q}(1) \Rightarrow \exists z.\, \mathcal{A}(z) * \mathcal{T}(z)$. Note that for all z

$$\boxed{\gamma : (1-q)^+} * \boxed{\gamma : z^-} \Rightarrow z + q \geq 1$$
$$\boxed{\delta : q^+} * \boxed{\delta : (1-z)^-} \Rightarrow z + q \leq 1$$

Let z be arbitrary, hence we have

$$\mathcal{P}_{\mathsf{send}}(1) * \mathcal{T}(z) \Rightarrow \boxed{\gamma : (z - (1-q))^-} * \boxed{\delta : ((1-z) - q)^-} \wedge (z + q = 1)$$
$$\Rightarrow \mathcal{Q}(0) \wedge (z + q = 1)$$

and also $(z + q = 1)$ is pure, so we can use $(z + q = 1)$ as $\varphi(z)$ in the CAS$-$*param* rule, i.e. we may learn this fact once the CAS operation is finished.

Furthermore, because the access type of the fetch-and-add is `rel` the thread gets

$$\exists z. \bigtriangledown \mathcal{A}(z) \wedge (z + q = 1)$$

which implies

$$\bigtriangledown \mathcal{A}(1 - q)$$

Hence after the fetch-and-add operation the thread is left with

$$\mathcal{P}_{\mathsf{keep}}(1) * \bigtriangledown \mathcal{A}(1-q) \Leftrightarrow \mathtt{a.data} \overset{q}{\mapsto} v * \bigtriangledown \mathtt{a.data} \overset{1-q}{\mapsto} v$$

which after the fence acquire instruction results in the full permission to `a.data`. This concludes the proof of the fetch-and-add in the `drop` function.

4 Extension of Location Invariants

In this section we consider an extension of FSL++ location invariants, which is targeted towards read-modify-write operations. First, we motivate the extension by using the main examples introduced in Section 3, we then provide a formal definition and finally we apply the extension to the main examples.

4.1 Motivation

In all three examples considered in Section 3 the corresponding location invariants contain existential quantifiers. The main reason for the existential quantifiers in all of these examples is that there is a value that cannot, in general, be expressed precisely in terms of the value stored at the atomic location governed by the location invariant. For example, consider the location invariant for the *RWSpin* example (Section 3.1):

$$\begin{aligned}\mathcal{Q}(v) := \ & \text{let } n = \left\lfloor \frac{v}{4} \right\rfloor, w = \mathsf{lsb}(v) \text{ in} \\ & \exists n_r \in \mathbb{N}.\, v \geq 0 \wedge n_r \leq n \wedge (w = 1 \Rightarrow n_r = 0) \wedge \\ & \mathit{resource}(\mathtt{bits})^{(w=1 \ ? \ 0 \ : \ 1-n_r \cdot \epsilon)} * \\ & \boxed{\alpha : 1 - w} * \boxed{\beta : 1 - n_r \cdot \epsilon} * \boxed{\gamma : 1 - (n - n_r) \cdot \epsilon}\end{aligned}$$

Recall that if the least significant bit (w) is 1 then there is no writer and otherwise there is a writer. A thread that wants to acquire a reader lock increments the atomic location by 4 and if the value that was incremented indicated that there was no writer ($w = 0$) then the thread has acquired a reader lock and otherwise not. If the reader lock was not acquired then the thread decrements the atomic location by 4.

The issue is that by just looking at the value v stored at the atomic location, the number of threads that have acquired a reader lock ("real readers") cannot be expressed if there is no writer, since there can simultaneously be threads that have incremented by 4 with a reader lock and threads that have incremented by 4 without a reader lock (before decrementing by 4).

This is the reason why in the location invariant the number of real readers is expressed using the existentially quantified variable n_r. In the proof for *RWSpin* the functions are verified using the idea that n_r gives the number of real readers and it is ensured that for each operation performed n_r's value stays consistent with this idea. The question is now, given that n_r is not uniquely defined by the value at the atomic location, whether the location invariant allows for n_r to have different meanings.

To answer this question consider the specification for the first operation needed to acquire a reader lock:

$$\{\mathsf{U}(\mathtt{bits}, \mathcal{Q})\}\, x := \mathtt{fetch\_and\_add}_{\mathtt{acq}}(\mathtt{bits}, 4) \left\{ \begin{array}{l} (\mathsf{lsb}(x) = 0 \ ? \\ \ \mathit{resource}(\mathtt{bits})^{\epsilon} * \boxed{\beta : \epsilon} : \\ \ \boxed{\gamma : \epsilon}) \end{array} \right\}$$

This reflects the explanation given above. If right before the increment by 4 there was no writer, then the thread gets some permission to the resource after the update, hence n_r inside the location invariant is incremented by 1, which is consistent with our idea since the number of real readers increases by 1.

It turns out that the following specification can also be verified using the same location invariant

$$\{\mathsf{U}(\mathtt{bits}, \mathcal{Q})\}\, x := \mathtt{fetch\_and\_add}_{\mathtt{acq}}(\mathtt{bits}, 4) \left\{ \boxed{\gamma : \epsilon} \right\}$$

So the thread does not get any permission to the resource even if there was no writer when the update occurred. In this case n_r did not change and hence n_r does not count the number of real readers. So the location invariant allows n_r to have a different meaning than what was originally intended. Of course using this second proof, the function to acquire a reader lock cannot be verified any more, because no permission to *resource*(`bits`) is acquired.

So the location invariant allows for local proofs that do not work in the grand scheme. This can be seen as a form of non-determinism, in the sense that the specification and accompanying proof that is used to verify the fetch-and-add operation is not determined by the specific update.

We observe in this example that we know when n_r *should* be incremented by 1 and when it *should not* be incremented by 1. The problem is that the existential quantifier does not precisely express this conceptual understanding.

We therefore propose extending the location invariant with a transition function which defines how such existentials should change after read-modify-write operations dependent on the values read, written and potentially more. In the example above this transition function would state that if the atomic location with value v, where $\mathsf{lsb}(v) = 0$, is updated to $v + 4$ then n_r should be incremented by 1. If v, where $\mathsf{lsb}(v) = 1$, is updated to a value $v + 4$ then n_r should not change.

4.2 Formal definition

We generalize the location invariant governing an atomic location a_{loc} to have the following type:

$$\mathcal{G} : \mathit{Values} \times S \longrightarrow \mathit{Assertions}$$

where *Values* is the type of values that can be stored at a_{loc} and S is a type which can be chosen for different location invariants. We call such a location invariant a *generalized location invariant*.

Suppose location a_{loc} is governed by such a generalized location invariant $\mathcal{G}$. One way to think about the generalized location invariant is that if location a_{loc} holds value v then $\mathcal{G}(v, r)$ holds for some r of type S at a_{loc}, i.e. the location in a sense owns the resources described by $\mathcal{G}(v, r)$. We call r the *witness* of a_{loc} with respect to $\mathcal{G}$. This description holds as long as acquire reads and relaxed reads of this location cannot acquire any resources from the location invariant (see also the discussion about location invariants in Section 2.3.2). The value of the witness need not be determined uniquely by the value of v.

This generalization allows making location invariants in standard FSL++, which contain existential quantifiers, more precise. For example, in *RWSpin* the original location invariant is of the form $\mathcal{Q}(v) = \exists n_r.\, \mathcal{P}(v, n_r)$ where $\mathcal{P}$ has type

$$\mathcal{P} : \mathit{Values} \times \mathbb{N} \longrightarrow \mathit{Assertions}$$

As explained previously, the idea is that n_r gives the number of real readers but the location invariant $\mathcal{Q}$ can be used in ways where this is not the case. $\mathcal{P}$ can be utilized as a generalized location invariant for *RWSpin*, where we choose the type S to be $\mathbb{N}$. The second argument of $\mathcal{P}$ is a possible witness for the existentially quantified variable n_r in $\mathcal{Q}$. The generalized location invariant makes this witness explicit and this enables specifying how n_r should evolve, as we will see shortly. This makes it possible to force that n_r gives the number of real readers, which is what makes the generalized location invariant more precise than the original location invariant in this case.

We now introduce *transition functions* which allow specifying how the witness should evolve. Suppose a read-modify-write operation is executed on the atomic location in *RWSpin* where the value at the time of the update is v and the witness is given by n_r (i.e. $\mathcal{P}(v, n_r)$ holds right before the read-modify-write operation has begun executing and hence the number of real readers is given by n_r). Ideally, the transition function for the generalized location invariant $\mathcal{P}$ should take the value v, the witness n_r, the updated value v' as arguments and output the number of real readers n_r' once the read-modify-write operation has finished executing (i.e. $\mathcal{P}(v', n_r')$ holds after the operation). This would make the transition deterministic and would prohibit proofs from giving n_r a different meaning other than the number of real readers after the CAS has finished executing. Hence the proof of the fetch-and-add operation in Section 4.1 which ultimately leads to the method not being verified would not be possible using the correct transition function. Unfortunately it is not always possible to decide how n_r should evolve by just looking at these three parameters, as we will see later on. It turns out that one more parameter is needed, namely an assertion describing the (partial) state of the thread executing the read-modify-write operation. While this makes it possible to define a proper transition function in all of our examples, it does not get rid of the non-determinism with respect to the proof choice in cases where the transition function depends on the state of the thread, as we will see.

Formally, a transition function $t_{\mathcal{G}}$ which extends a generalized location invariant $\mathcal{G}$ is a *partial function* which must have the following type:

$$t_{\mathcal{G}} : \mathit{Values} \times S \times \mathit{Values} \times \mathit{Assertions} \hookrightarrow S$$

Suppose a_{loc} is governed by $\mathcal{G}$. Further, assume that a_{loc} holds the value v_0 and the witness of a_{loc} with respect to $\mathcal{G}$ is r_0. Then for a read-modify-write operation that updates the value to v_1 the assertion $\mathcal{G}(v_1, t_{\mathcal{G}}(v_0, r_0, v_1, P))$

$$
\begin{array}{l}
x \notin FV(P) \\
P \Leftrightarrow P_{keep} * P_{send} \\
\forall r. \left(\begin{array}{c} \mathsf{defined}(t_{\mathcal{G}}(v, r, v', P)) \\ \mathcal{G}(v, r) \Rightarrow \mathcal{A}(r) * \mathcal{T}(r) \\ P_{send} * \mathcal{T}(r) \Rightarrow \mathcal{G}(v', t_{\mathcal{G}}(v, r, v', P)) \wedge \varphi(r) \\ \mathit{pure}(\varphi(r)) \end{array} \right) \\
\hline
\{\mathsf{U}(l, \mathcal{G}) * P\} x := \mathtt{CAS}_{\mathtt{rel\_acq}}(a_{loc}, v, v') \left\{ \begin{array}{l} (x = v \wedge P_{keep} * \exists r.\, \mathcal{A}(r) \wedge \varphi(r)) \\ \vee (x \neq v \wedge \mathsf{U}(l, \mathcal{G}) * P) \end{array} \right\}
\end{array}
$$

Figure 13: CAS rule for location invariants extended with transition functions

must be guaranteed (i.e. the witness of a_{loc} changes from r_0 to $t_{\mathcal{G}}(v_0, r_0, v_1, P)$). The parameter P describes part of the state of the thread executing the read-modify-write operation. This idea is reflected by an adjusted proof rule for CAS operations given in Figure 13. There may be many possible witnesses for each value that is read. As a result, in the CAS rule the split into the part that is extracted from the location invariant and the part that is kept in the location invariant must be verified for every witness. These two parts may be chosen separately for each witness. This is one difference to the CAS$-$*basic* rule given in Figure 1 and is an aspect that is similar to the CAS$-$*param* rule given in Figure 2. It must be guaranteed that for every possible witness the transition function is defined. In the conclusion of the rule after the CAS has executed it is in general not known what the witness was, i.e. the exact resources $\mathcal{A}(r)$ which are acquired may not be known. In practice $\mathcal{A}(r)$ is either not dependent on r or $\varphi(r)$ gives enough information about r to be able to deduce what the acquired resources are.

There is an issue with the CAS rule in Figure 13. The transition function requires the assertion in the precondition of the CAS statement as an argument. The problem is that a user of the logic may decide which part of the precondition to plug in due to the frame rule in separation logic. For example, the following is valid (assuming x does not occur in R):

$$
\frac{\{A\}\, x := \mathtt{CAS}_{\mathtt{rel\_acq}}(a_{loc}, v, v')\, \{B\}}{\{A * R\}\, x := \mathtt{CAS}_{\mathtt{rel\_acq}}(a_{loc}, v, v')\, \{B * R\}}
$$

As a result, depending on which parts of the precondition are framed away the transition function might return a different result. Therefore there still is non-determinism present in terms of the specification/proof chosen for a CAS and hence there can still be proofs that work locally but not globally.

Cases where the transition functions are independent of the precondition are deterministic, as the situation described in Section 4.1.

We note that this choice that the user of the logic may make, is essential for the logic to work for the CAS rules presented in Section 2.3.2, since in those rules all of the resources in the precondition are lost to the location invariant. But in this new CAS rule there is the choice of retaining part of the resources in the precondition, so here framing would not be necessary.

Even if framing were never applied for the CAS there is still non-determinism present due to the nature of the fetch-and-add rule (see Figure 4). Recall that in the fetch-and-add rule it must be specified which resources should be used for the CAS and which resources should be retained. This case could be avoided by changing the fetch-and-add rule to always use all the resources for the CAS and hence letting the CAS rule decide which resources should be retained.

It would be interesting to investigate if it is possible to extend the logic to ensure that no resources are framed away in such situations. This would make it possible to always use the full state belonging to a thread executing the CAS for the transition function, which would lead to a deterministic situation.

4.3 Extending the location invariant of *RWSpin*

We define the generalized location invariant of *RWSpin* to be:

$$\begin{aligned}\mathcal{G}_{RWSpin}(v, n_r) := \ & \text{let } n = \left\lfloor \frac{v}{4} \right\rfloor, w = \mathsf{lsb}(v) \text{ in} \\ & v \geq 0 \land n_r \leq n \land (w = 1 \Rightarrow n_r = 0) \land \\ & \mathit{resource}(\texttt{bits})^{(w=1 \ ? \ 0 \ : \ 1-n_r \cdot \epsilon)} * \\ & \boxed{\alpha : 1 - w} * \boxed{\beta : 1 - n_r \cdot \epsilon} * \boxed{\gamma : 1 - (n - n_r) \cdot \epsilon}\end{aligned}$$

where $n_r \in \mathbb{N}$. This is exactly the same as the location invariant given in Section 3.1.3 except that we have removed the existential quantifier.

We extend this location invariant with the transition function $t_{\mathcal{G}_{RWSpin}}$

where

$$t_{\mathcal{G}_{RWSpin}}(v, n_r, v', P) = \begin{cases} n_r + 1 & if\, v' = v + 4, \mathsf{lsb}(v) = 0 \\ n_r & if\, v' = v + 4, \mathsf{lsb}(v) = 1 \\ n_r - 1 & if\, v' = v - 4, P \not\models \boxed{\gamma : \epsilon} \\ n_r & if\, v' = v - 4, P \models \boxed{\gamma : \epsilon} \\ 0 & if\, v = 0, v' = 1 \\ n_r & if\, v' = \mathsf{clearlsb}(v), \mathsf{lsb}(v) = 1, P \not\models \boxed{\gamma : \epsilon} \\ n_r + 1 & if\, v' = \mathsf{clearlsb}(v), \mathsf{lsb}(v) = 1, P \models \boxed{\gamma : \epsilon} \\ \mathsf{undefined} & otherwise \end{cases}$$

The two cases where $v' = v + 4$ correspond to the cases we discussed when motivating transition functions.

Note when $v' = v - 4$ then the transition function decides on the result by looking at P. The interesting scenario in this case is when a thread has access to permission $resource(\texttt{bits})^{\epsilon} * \boxed{\beta : \epsilon}$ and $\boxed{\gamma : \epsilon}$ (i.e. the thread incremented twice by 4 and has not decremented by 4 yet, once the thread got the real reader permission $resource(\texttt{bits})^{\epsilon} * \boxed{\beta : \epsilon}$, and once the thread did not get real reader permission, i.e. the thread got $\boxed{\gamma : \epsilon}$). So now when such a thread decrements by 4 then it is possible to either give up the real reader permission or $\boxed{\gamma : \epsilon}$. In this transition function we would like to enforce that $\boxed{\gamma : \epsilon}$ is given up in this case to make sure that a thread can remain a real reader as long as possible.

A similar situation exists for $v' = \mathsf{clearlsb}(v)$ and $\mathsf{lsb}(v) = 1$ where the transition function again decides based on P. In this case the scenario occurs in *RWSpin*. The case where the thread does not hold permission to $\boxed{\gamma : \epsilon}$ corresponds to the normal writer unlock and the case where the thread holds permission to $\boxed{\gamma : \epsilon}$ corresponds to the writer unlock that is executed when the function `unlock_and_lock_shared` is executed. There we want that the $\boxed{\gamma : \epsilon}$ permission is exchanged for $\boxed{\beta : \epsilon}$ permission i.e. the number of real readers increases by 1.

These two cases which depend on P still contain ambiguities with respect to the proof chosen as explained in Section 4.2.

4.4 Extending the location invariant of *Barrier*

We define the generalized location invariant of *Barrier* to be:

$$\begin{aligned}\mathcal{G}_{Barrier}((r,v),w) := \; & (0 \le r) \wedge (r \le v) \wedge (v < n \Rightarrow w = 0) \wedge \\ & ([(r,v) \neq (0,n) \wedge (v \le n)] \; ? \; tok.val \overset{1-w}{\mapsto} \_ : \mathsf{emp}) * \\ & \boxed{\delta : 1 - (r-w)\cdot\epsilon} * \boxed{\lambda : 1-w}\end{aligned}$$

where $w \in \{0,1\}$. This is exactly the same as the location invariant given in Section 3.2.8 except that we have removed the existential quantifier.

We extend this location invariant with the transition function $t_{\mathcal{G}_{Barrier}}$ where

$$t_{\mathcal{G}_{Barrier}}((r,v),w,(r',v'),P) = \begin{cases} 0 & \textit{if } (r',v') = (r+1,v+1) \\ & \textit{and } v+1 < n \\ 1 & \textit{if } (r',v') = (r+1,v+1) \\ & \textit{and } v+1 = n \\ w & \textit{if } (r',v') = (r+1,v+1) \\ & \textit{and } v+1 > n \\ w & \textit{if } (r',v') = (r-1,v), P \models \boxed{\delta : \epsilon} \\ 0 & \textit{if } (r',v') = (r-1,v), P \not\models \boxed{\delta : \epsilon} \\ \mathsf{undefined} & \textit{otherwise} \end{cases}$$

The cases where $(r',v') = (r+1,v+1)$ correspond to the first read-modify-write operation in the function `wait`. In the original invariant it is possible to keep $w = 0$ in all these cases, even though this does not work out to verify the whole function. Using the transition function we can resolve this ambiguity (without inspecting P).

The remaining cases correspond to the second read-modify-write operation. Using the transition function we try to resolve the ambiguity when a thread holds $\boxed{\delta : \epsilon} * \boxed{\lambda : 1}$ (even though this scenario does not occur in the original function). As discussed in Section 4.2 this does not entirely resolve the ambiguity.

4.5 Extending the location invariant of *ARC*

We define the generalized location invariant of *ARC* to be:

$$\begin{aligned}\mathcal{G}_{ARC}(c,f) := \; & \textbf{if } c = 0 \textbf{ then } \boxed{\gamma : 0^-} * \boxed{\delta : 0^-} \\ & \textbf{else } \texttt{a.data} \overset{f}{\mapsto} v * \boxed{\gamma : (c-1+f)^-} * \boxed{\delta : (1-f)^-}\end{aligned}$$

where $f \in \mathbb{Q} \cap [0,1]$. This is exactly the same as the location invariant given in Section 3.3.2 except that we have removed the existential quantifier.

We extend this location invariant with the transition function $t_{\mathcal{G}_{ARC}}$ where

$$t_{\mathcal{G}_{ARC}}(c, f, c', P) := \begin{cases} f & \textit{if } c' = c + 1 \\ f + q & \textit{if } \begin{array}{l} c' = c - 1, c' > 0 \textit{ and} \\ P \models \boxed{\delta : q^+} \textit{ for some } q \end{array} \\ 0 & \textit{if } c' = c - 1, c' = 0 \\ \mathsf{undefined} & \textit{otherwise} \end{cases}$$

The case $c' = c+1$ corresponds to the `clone` function. In the original location invariant given in Section 3.3.2 it is possible to change f when incrementing by 1 which then leads to the postcondition of `clone` not being verified. This is possible since one can extract

$$\left(\triangledown \texttt{a.data} \overset{f}{\mapsto} v\right) * \boxed{\gamma : (1-f)^+} * \boxed{\delta : f^+}$$

from the original location invariant while still making sure that the location invariant holds for $c + 1$ (we omit the proof). $f \in [0, 1]$ corresponds to the permission amount to `a.data` held by the location invariant right before the update, so we can extract all of it (it is guarded by a modality due to the `rlx` access type). Therefore the witness for the existential quantifier after the update is 0 in this case. If $f > 0$ then it is not possible to construct two ARC resources and therefore the postcondition of `clone` cannot be verified.

Using the generalized location invariant extended with the presented transition function resolves this ambiguity without inspecting P.

The two cases where $c' = c - 1$ both correspond to the `drop` function.

4.6 Summary

Generalized location invariants along with transition functions allow for more expressive location invariants which make certain specification and proof choices for CAS operations deterministic. This determinism ensures that always the correct proof is picked (in the respective cases). As a result, generalized location invariants are helpful for tools such as Viper which automate the FSL++ CAS rule and where there is no backtracking mechanism which explores different proofs.

In cases where the transition function depends on the precondition of the CAS triple there still is non-determinism with respect to the proof choice. It seems as if being able to talk about all the resources that a thread owns would be useful in this case. Such a notion does not exist in separation logic since it does not fit with the notion of framing and breaks modularity.

5 The *EFC* permission structure

In this section we introduce a permission structure which combines the fractional and counting permission structures in a different way than the *SFC* permission structure given in Section 2.4.

5.1 Motivation

In the *ARC* example (see Section 3.3) there is a single atomic location which counts the number of references that exist to the non-atomic location `a.data`. In the proof which we presented (and which was taken from [9]) the ownership of such a reference was modelled using an ARC resource (which can be defined in terms of permission to `a.data` and permission to certain ghost locations).

The `clone` method conceptually gets an additional reference to `a.data` by incrementing the counter at the atomic location, i.e. in the formal specification an additional ARC resource is generated. The `drop` method conceptually destroys a reference to `a.data` by decrementing the counter at the atomic location, i.e. in the formal specification an ARC resource is given up. Additionally, if a thread gives up the last available reference to `a.data` then it deallocates the memory associated with `a.data`. Hence in the proof this thread needs to get hold of the full permission to `a.data`.

To be able to verify these two methods the proof uses the following location invariant for the atomic location (it is given in Section 3.3.2):

$$\begin{aligned}\mathcal{Q}_{a,v}(c) := \ & \textbf{if } c = 0 \textbf{ then } \boxed{\gamma : 0^-} * \boxed{\delta : 0^-} \\ & \textbf{else } \exists f \in \mathbb{Q} \cap [0,1].\ \texttt{a.data} \overset{f}{\mapsto} v * \boxed{\gamma : (c-1+f)^-} * \\ & \qquad \boxed{\delta : (1-f)^-}\end{aligned}$$

where γ and δ are ghost locations which are governed by the *SFC* permission structure introduced in Section 2.4. This location invariant does two important things:

1. Track the number of ARC resources that are held by the threads
2. Track the total amount of permission to `a.data` held by the ARC resources

If the value at the atomic location is c then c ARC resources are held by the threads, the location invariant tries to capture this using the assertion $\boxed{\gamma : (c-1+f)^-}$ (recall that a^- is a source permission, i.e. there can only ever be one such permission and it is used to distribute and collect access permissions which are of the form b^+, where $a, b \in \mathbb{Q}_{\geq 0}$).

At first glance it might seem unclear why to be able to assert that there are c ARC resources one must use $(c-1+f)^-$, since this value is dependent on f. The main reason is that when $c = 1$ then one needs this value to be f^- to then be able to verify that the last thread gets the full permission to `a.data`.

$1 - f$ is the total amount of permission to `a.data` held by those c ARC resources collectively (where f is the existentially quantified variable in the location invariant). This is captured inside the location invariant partially by the assertion $\texttt{a.data} \overset{f}{\mapsto} v$, i.e. all the permission that is not in the c ARC resources is held in the location invariant. Also note that the assertion $\boxed{\delta : (1-f)^-}$ holds inside the location invariant.

One question that may arise here is that why one needs $\boxed{\delta : (1-f)^-}$ if one can assert $\texttt{a.data} \overset{f}{\mapsto} v$. One reason is again the $c = 1$ case. The interplay of $(1-f)^-$ being associated with δ and f^- being associated with γ along with the ghost permissions inside the ARC resource is used to deduce that the last thread gets the full permission to `a.data`.

This setup using the *SFC* permission structure is very clever, but we aim to answer the question if there is a permission structure which allows to express the above two aspects in a more direct manner.

Suppose there exists a permission structure that like the *SFC* permission structure has two types of permissions: a source permission and an access permission. There can only be one source permission, which is used to distribute and collect multiple access permissions. In general the source permission appears inside the location invariant to summarize what access permissions have been given away to the threads that have interacted with the atomic location.

Further assume that in this permission structure an access permission consists of some number of *entities* which are associated with a permission value. The source permission counts how many entities have been given away as well as the total sum of the permission values that are associated with these entities. Then we could express the above two aspects using a single ghost location with the source permission ghost resource $(c, 1-f)^-$ in the location invariant (if $c > 0$), where c is the number of entities that have been given away (i.e. ARC resources in our case) and $1 - f$ is the total sum of the permission values associated with these entities[8]. A single ARC resource held by a thread would then be represented by the access permission $(1, q)^+$ where $q \in \mathbb{Q}_{>0}$ and permission $\texttt{a.data} \overset{q}{\mapsto} \_$. Conceptually, the first

[8]We will see later that one can define the permission structure in a way such that this ghost location also suffices for the case when the last thread calls `drop`.

component of such an access permission states that it only holds *one ARC resource* and the second component states *the amount of permission it holds to the memory location* `a.data`. This would lead to a location invariant that expresses the situation in a more direct manner than the original location invariant and is hence easier to explain.

Such a permission structure can be seen as a combination of the counting permission structure and the fractional permission structure. The counting permission structure has a notion of *entities* since there is a resource that is not splittable (1^+) and the fractional permission structure allows a resource to always be splittable (as long as some positive permission amount is associated with it) which is needed if one wants the source permission to be able to give away entities associated with arbitrary fractional permission amounts. Note that the source permission in the *SFC* permission structure can only count how much permission was given away and does not have a concept of *entities*, hence it is slightly less expressive than the proposed concept.

Next, we show that such a permission structure exists. In a later section we will then reprove the ARC example using this permission structure. Additionally we will see that there are also location invariants for the *RWSpin* and *Barrier* examples using this permission structure which use fewer ghost locations than in the proofs presented in Section 3.

5.2 Formalization of the *EFC* permission structure

In this section we formally define the permission structure proposed in the previous section. We call the permission structure the *entity fractional-counting permission structure* and we will refer to it as the *EFC* permission structure.

The *EFC* permission structure[9] is given by the following algebraic structure

$$\left((\mathbb{N}_{>0} \times \mathbb{Q}_{>0} \times \{-,+\}) \cup \{(a,0)^+, (a,0)^- | a \in \mathbb{N}\}, \oplus, (0,0)^+, (0,0)^-\right)$$

where the partial binary operation $\oplus$ is defined as

$$(c,d)^+ \oplus (a,b)^- := (a,b)^- \oplus (c,d)^+ := \begin{cases} (a-c, b-d)^- & \begin{array}{r} \textit{if } a-c \geq 0 \textit{ and} \\ b-d \geq 0 \textit{ and} \\ \left(\begin{array}{r} a-c=0 \Rightarrow \\ b-d=0 \end{array}\right) \end{array} \\ \mathsf{undefined} & \textit{otherwise} \end{cases}$$

$$(a,b)^+ \oplus (c,d)^+ := (a+c, b+d)$$
$$(a,b)^- \oplus (c,d)^- := \mathsf{undefined}$$

[9] We write $(a,b,+)$ as $(a,b)^+$ and $(a,b,-)$ as $(a,b)^-$.

Claim: The above algebraic structure is a permission structure.

Proof. Define $S := (\mathbb{N}_{>0} \times \mathbb{Q}_{>0} \times \{-, +\}) \cup \{(a, 0)^+, (a, 0)^- | a \in \mathbb{N}\}$ to be the set of elements in the algebraic structure.

We first show that $(S, \oplus)$ forms a partial commutative monoid with neutral element $(0, 0)^+$ (see Appendix A for the definition of a partial commutative monoid). For elements $(a, b)^+ \in S$ and $(a, b)^- \in S$ where a, b are arbitrary it holds that

$$(a, b)^+ \oplus (0, 0)^+ = (a + 0, b + 0)^+ = (a, b)^+ = (0 + a, 0 + b)^+ = (0, 0)^+ \oplus (a, b)^+$$
$$(a, b)^- \oplus (0, 0)^+ = (a - 0, b - 0)^- = (a, b)^- = (0, 0)^+ \oplus (a, b)^-$$

hence $(0, 0)^+$ is a neutral element.

Commutativity follows more or less directly from the definition of $\oplus$ and the commutativity of $+$.

For associativity we need to show $(x \oplus y) \oplus z \cong x \oplus (y \oplus z)$ for arbitrary $x, y, z \in S$. We call an element in the structure *positive* if it is of the form $(a, b)^+$ and *negative* if it is of the form $(a, b)^-$. If x, y, z are all positive then both sides are defined and they evaluate to the same value. If at least two of x, y, z are negative then both sides are undefined. Suppose just one of x, y, z is negative. We need to consider three cases. In the following we assume that a, b, c, d, e, f are arbitrary and that $(a, b)^+, (a, b)^-, (c, d)^+, (c, d)^-, (e, f)^+, (e, f)^-$ are elements in S.

Assume y is negative. We need to show

$$((a, b)^+ \oplus (c, d)^-) \oplus (e, f)^+ \cong (a, b)^+ \oplus ((c, d)^- \oplus (e, f)^+)$$

We first show that that the left hand side is defined iff the right hand side is defined.

Assume $((a, b)^+ \oplus (c, d)^-) \oplus (e, f)^+$ is defined. Hence $c - a - e \geq 0$, $d - b - f \geq 0$ and $c - a - e = 0 \Rightarrow d - b - f = 0$ holds. Therefore $c - e \geq 0$, $d - f \geq 0$. Furthermore, if $c - e = 0$ then $a = 0$ (since $c - a - e \geq 0$) then $b = 0$ (since $(0, x)^+ \in S$ iff $x = 0$) and hence $d - f = 0$ (since $c - a - e = 0 \Rightarrow d - b - f = 0$). We conclude that $(c, d)^- \oplus (e, f)^+$ is defined. Therefore $(a, b)^+ \oplus ((c, d)^- \oplus (e, f)^+)$ is defined (follows directly from the definedness of the left hand side).

Next, assume $(a, b)^+ \oplus ((c, d)^- \oplus (e, f)^+)$ is defined. Hence $c - a - e \geq 0$, $d - b - f \geq 0$ and $c - a - e = 0 \Rightarrow d - b - f = 0$ holds. Therefore $c - a \geq 0$, $d - b \geq 0$. Furthermore, if $c - a = 0$ then $e = 0$, hence $f = 0$ and therefore $d - b = 0$. We conclude that $(a, b)^+ \oplus (c, d)^-$ is defined. Therefore $((a, b)^+ \oplus (c, d)^-) \oplus (e, f)^+$ is defined (follows directly from the definedness of the right hand side).

If both sides are defined then they evaluate to the same value, which is given by $(c-a-e, d-b-f)^-$.

Next, assume x is negative. We need to show

$$((a,b)^- \oplus (c,d)^+) \oplus (e,f)^+ \cong (a,b)^- \oplus ((c,d)^+ \oplus (e,f)^+)$$

If both sides are defined then they evaluate to the same value, which is given by $(a-c-e, b-d-f)^-$. We show that the left hand side is defined iff the right hand side is defined.

Suppose $((a,b)^- \oplus (c,d)^+) \oplus (e,f)^+$ is defined. Hence $a-c-e \geq 0$, $b-d-f \geq 0$ and $a-c-e=0 \Rightarrow b-d-f=0$. This directly shows that $(a,b)^- \oplus ((c,d)^+ \oplus (e,f)^+)$ is defined.

Suppose $(a,b)^- \oplus ((c,d)^+ \oplus (e,f)^+)$ is defined. Hence $a-c-e \geq 0$, $b-d-f \geq 0$ and $a-c-e=0 \Rightarrow b-d-f=0$. Therefore $a-c \geq 0$ and $b-f \geq 0$. Furthermore, if $a-c=0$ then $e=0$, hence $a-c-e=0$ and therefore $b-d-f=0$. Also since $e=0$ we conclude $f=0$, therefore $b-d=0$. We conclude that $(a,b)^- \oplus (c,d)^+$ is defined and hence $((a,b)^- \oplus (c,d)^+) \oplus (e,f)^+$ is defined (follows directly from the definedness of the right hand side).

Next, assume z is negative. We need to show

$$((a,b)^+ \oplus (c,d)^+) \oplus (e,f)^- \cong (a,b)^+ \oplus ((c,d)^+ \oplus (e,f)^-)$$

using the commutativity property, this is equivalent to showing

$$(e,f)^- \oplus ((c,d)^+ \oplus (a,b)^+) \cong ((e,f)^- \oplus (c,d)^+) \oplus (a,b)^+$$

but we showed this already in the case when x is negative. Hence associativity holds in all cases.

We have shown that $(S, \oplus)$ forms a partial commutative monoid with neutral element $(0,0)^+$.

Next we show that $(0,0)^-$ is a maximal element with respect to S and neutral element $(0,0)^+$. We have for $(a,b)^+ \in S$ where a, b are arbitrary and $(a,b)^+ \neq (0,0)^+$ that

$$(0,0)^- \oplus (a,b)^+ = (0-a, 0-b)^+ =_{a>0 \text{ or } b>0} \mathsf{undefined}$$

Furthermore, $(0,0)^- \oplus (c,d)^- = \mathsf{undefined}$ holds for arbitrary $(c,d)^- \in S$. Hence $(0,0)^-$ is a maximal element with respect to S. This concludes the proof. □

5.3 Understanding the *EFC* permission structure

We relate the formal definition of the *EFC* permission structure to the motivation. A source permission in the permission structure is given by $(c, s)^-$. c is the number of entities that have been given away (the *entity count*) and s is the sum of the permission amounts associated with these entities (the *entity permission sum*), where $c \in \mathbb{N}$ and $s \in \mathbb{Q}_{\geq 0}$. There can only be one source permission (like in the counting and *SFC* permission structures).

Furthermore, we have that if $c = 0$ then $s = 0$. The motivation for this is that if no entities have been given away then the entity permission sum must be 0. We do not have that if $s = 0$ then $c = 0$ even though this would also be a possibility. The reason we do not enforce this condition is that it turns out to be useful to be able to give away entities that are associated with a permission amount of 0 (see the next section to see where this helps). Hence this means that it is possible for the entity permission sum to be 0 even though entities have been given away.

We call $(n, p)^+ \in S$ an access permission; there can be multiple access permissions. n denotes the number of entities that are held in this access permission and p denotes the total permission associated with these n entities. Here again $n \in \mathbb{N}$ and $p \in \mathbb{Q}_{\geq 0}$. We also have that $n = 0 \Rightarrow p = 0$, so only zero permission amount can be associated with 0 entities (otherwise we cannot guarantee the source permission property that if $c = 0$ then $s = 0$).

The full permission is given by $(0, 0)^-$, i.e. a source permission, where no entities have been given away. The empty permission is given by $(0, 0)^+$, i.e. an access permission, where no entity is contained.

6 Using the *EFC* permission structure

In this section we prove the examples which were already proved in Section 3 using different location invariants. In particular, all the location invariants in this section make use of the *EFC* permission structure introduced in Section 5. Due to the expressiveness of this permission structure it is possible to use fewer ghost locations in the location invariants without making the reasoning about the proofs harder.

6.1 Folly reader-writer spinlock

RWSpin was introduced and proved in Section 3.1. In this section we give a different proof using the *EFC* permission structure.

6.1.1 Location invariant

The location invariant using the *EFC* permission structure for the integer atomic location `bits` is given by:

$$\begin{aligned}\mathcal{Q}^{EFC}(v) := \ & \text{let } n = \left\lfloor \frac{v}{4} \right\rfloor, w = \mathsf{lsb}(v) \text{ in} \\ & \exists s \in \mathbb{Q} \cap [0,1).\, v \geq 0 \wedge (w = 1 \Rightarrow s = 0) \wedge \\ & \mathit{resource}(\texttt{bits})^{(w=1\ ?\ 0\ :\ 1-s)} * \\ & \boxed{\alpha : 1 - w} * \boxed{\lambda : (n, s)^{-}}\end{aligned}$$

where α is a ghost location governed by the fractional permission structure and λ is a ghost location governed by the *EFC* permission structure.

The idea of the location invariant is that the existentially quantified variable s expresses the amount of permission that was given away to real readers, hence it must be less than 1 (otherwise at some point no more real readers can be supported). If there is a writer in the system then s must be 0 (since there cannot be any real readers if there is a writer).

λ is used to track the real and fake readers. Each real and fake reader gets one λ entity, hence the entity count of λ is given by n. Furthermore, in the proof we make sure that the permission associated with the entity given to a fake reader is given by 0 and the permission associated with the entity given to a real reader is some positive value (not necessarily ϵ). Hence the entity sum of λ is given by the sum of permissions held by real readers which is equal to s. This example illustrates why allowing entities to be associated with permission value 0 is useful.

Note that in the original location invariant given in Section 3.1.3 we needed two ghost locations to track the real and fake readers; here we only need one. Also here we allow readers to potentially have different permission amounts to the resource and not just ϵ permission. We could adjust the location invariant to only allow for ϵ permission amounts to the resource. This choice is not significant since if we enforced ϵ permission amounts then there would still be an existentially quantified variable (for the number of real readers) and the proof would be analogous.

The α ghost location has the same meaning as in the original location invariant. It is needed to ensure that if a thread has a writer lock then it must hold that $\mathsf{lsb}(v) = 1$ where v is the value at the atomic location.

6.1.2 Specification

The specification of the functions is very similar to the specification in Section 3.1.4. It mainly differs in the fact that it suffices for a reader to have

arbitrary, positive permission amount to the resource. The specification is given by:

$$\{\mathsf{U}(\texttt{bits},\mathcal{Q})\}\texttt{bool try\_lock\_shared()}\{y.(y\ ?\ R_\lambda^{EFC}(\texttt{bits}) : \mathsf{emp})\}$$

$$\left\{\begin{array}{l}\mathsf{U}(\texttt{bits},\mathcal{Q})*\\ R_\lambda^{EFC}(\texttt{bits})\end{array}\right\}\texttt{void unlock\_shared()}\{\mathsf{emp}\}$$

$$\{\mathsf{U}(\texttt{bits},\mathcal{Q})\}\texttt{bool try\_lock()}\{y.(y\ ?\ W_\alpha^{EFC}(\texttt{bits}) : \mathsf{U}(\texttt{bits},\mathcal{Q}))\}$$

$$\left\{\begin{array}{l}\mathsf{U}(\texttt{bits},\mathcal{Q})*\\ W_\alpha^{EFC}(\texttt{bits})*\\ (\texttt{getRead}\ ?\\ \quad \boxed{\lambda:(1,0)^+}:\\ \quad \mathsf{emp})\end{array}\right\}\texttt{void unlock(bool getRead)}\left\{\begin{array}{l}(\texttt{getRead}\ ?\\ R_\lambda^{EFC}(\texttt{bits}) : \mathsf{emp})\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{U}(\texttt{bits},\mathcal{Q})*\\ W_\alpha^{EFC}(\texttt{bits})\end{array}\right\}\texttt{void unlock\_and\_lock\_shared()}\left\{R_\lambda^{EFC}(\texttt{bits})\right\}$$

where

$$W_\alpha^{EFC}(\texttt{bits}) \equiv \mathit{resource}(\texttt{bits})^1 * \boxed{\alpha:1}$$
$$R_\lambda^{EFC}(\texttt{bits}) \equiv \exists q\in(0,1].\ \mathit{resource}(\texttt{bits})^q * \boxed{\lambda:(1,q)^+}$$

We make the same assumptions in the proofs regarding $\mathsf{U}(\texttt{bits},\mathcal{Q}^{EFC})$ and reading negative values as in Section 3.1.4.

6.1.3 Proof of function `try_lock_shared`

The proof outline is given in Figure 14.

First RMW. We first prove the first fetch-and-add operation. We need to show

$$\{\mathsf{U}(\texttt{bits},\mathcal{Q})\}\, v0 := \texttt{fetch\_and\_add}_{\mathsf{acq}}(\texttt{bits},4)\left\{\begin{array}{l}(\mathsf{lsb}(v0)=0\ ?\\ R_\lambda^{EFC}(\texttt{bits}) :\\ \quad \mathsf{U}(\texttt{bits},\mathcal{Q})*\\ \quad \boxed{\lambda:(1,0)^+}\quad)\end{array}\right\}$$

using the fetch-and-add rule. We have $P \equiv \mathsf{emp}$ and hence we choose $\mathcal{P}_{\mathsf{send}}(t) := \mathcal{P}_{\mathsf{keep}}(t) := \mathsf{emp}$. We verify the CAS triple in the premise of the fetch-and-add rule for every possible value t that could be read using the

$\{\mathsf{U}(\texttt{bits}, \mathcal{Q}^{EFC})\}$
$v0 := \texttt{fetch\_and\_add}_{\texttt{acq}}(\texttt{bits}, 4)$ `//first RMW`
$\left\{\mathsf{lsb}(v0) = 0 \ \texttt{?} \ \ R_\lambda^{EFC}(bits) \ \texttt{:} \ \ \mathsf{U}(\texttt{bits}, \mathcal{Q}^{EFC}) * \boxed{\lambda : (1,0)^+}\right\}$
`if (`$\mathsf{lsb}(v0)$ `== 1) {`
    $\left\{\mathsf{U}(\texttt{bits}, \mathcal{Q}^{EFC}) * \boxed{\lambda : (1,0)^+}\right\}$
    $v1 := \texttt{fetch\_and\_add}_{\texttt{rel}}(\texttt{bits}, -4)$ `//second RMW`
    $\{\mathsf{emp}\}$
    `res := false`
`} else {`
    $\{R_\lambda^{EFC}\}$
    `res := true`
`}`
$\{(res \ ? \ R_\lambda(\texttt{bits}) : \mathsf{emp})\}$
`return res`

Figure 14: Proof outline for the `try_lock_shared` *function with respect to the new location invariant*

CAS$-$*basic* rule. Suppose $\mathsf{lsb}(t) = 0$. We then have

$$\mathcal{Q}^{EFC}(t) \Leftrightarrow \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in } \exists s \in \mathbb{Q} \cap [0,1).\, t \geq 0 \wedge \mathit{resource}(\texttt{bits})^{1-s} * \boxed{\alpha : 1} * \boxed{\lambda : (n,s)^-}$$

Choose

$$T := \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in } \exists s' \in \mathbb{Q} \cap [0,1).\, t \geq 0 \wedge \mathit{resource}(\texttt{bits})^{1-s'} * \boxed{\alpha : 1} * \boxed{\lambda : (n+1,s')^-}$$

$$A := \exists q \in \mathbb{Q} \cap (0,1].\, \mathit{resource}(\texttt{bits})^q * \boxed{\lambda : (1,q)^+}$$

It holds that $\mathcal{Q}^{EFC}(t) \Rightarrow A * T$; this can be seen by observing

$$\exists s \in \mathbb{Q} \cap [0,1).\, t \geq 0 \wedge \mathit{resource}(\texttt{bits})^{1-s} * \boxed{\lambda : (n,s)^-} \Leftrightarrow$$
$$\exists s \in \mathbb{Q} \cap [0,1).\, \exists q \in \mathbb{Q} \cap (0,1].\, (q < 1-s) \wedge (t \geq 0) \wedge \mathit{resource}(\texttt{bits})^{1-s-q} * \mathit{resource}(\texttt{bits})^q * \boxed{\lambda : (n+1, s+q)^-} * \boxed{\lambda : (1,q)^+}$$

Furthermore, we have $T \Rightarrow \mathcal{Q}(t+4)$. Since $A \Leftrightarrow R_\lambda^{EFC}$ and the fetch-and-add operation has access type `acq`, the case $\mathsf{lsb}(t) = 0$ has been proved. Next,

suppose $\mathsf{lsb}(t) = 1$. In this case we have

$$\mathcal{Q}^{EFC}(t) \Leftrightarrow \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in } t \geq 0 \land \boxed{\alpha : 0} * \boxed{\lambda : (n, 0)^-}$$

We choose

$$T := \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in } t \geq 0 \land \boxed{\alpha : 0} * \boxed{\lambda : (n+1, 0)^-}$$
$$A := \boxed{\lambda : (1, 0)^+}$$

It holds that $\mathcal{Q}^{EFC}(t) \Rightarrow A * T$ and $T \Rightarrow \mathcal{Q}^{EFC}(t+4)$. Furthermore, since A is exactly the permission we need to receive in this case and it only contains ghost permissions, the case $\mathsf{lsb}(t) = 1$ has been proved (we still need to retain $\mathsf{U}(\texttt{bits}, \mathcal{Q}^{EFC})$, but we can get hold of this analogously to the proof in Section 3.1.5).

Second RMW. For the second fetch-and-add operation we need to show

$$\left\{ \mathsf{U}(\texttt{bits}, \mathcal{Q}^{EFC}) * \boxed{\lambda : (1, 0)^+} \right\} v1 := \texttt{fetch\_and\_add}_{\texttt{rel}}(\texttt{bits}, -4) \, \{\mathsf{emp}\}$$

using the fetch-and-add rule. We have $P \equiv \boxed{\lambda : (1, 0)^+}$ and we choose

$$\mathcal{P}_{\mathsf{send}}(t) := \boxed{\lambda : (1, 0)^+}$$
$$\mathcal{P}_{\mathsf{keep}}(t) := \mathsf{emp}$$

Clearly $P \Leftrightarrow \mathcal{P}_{\mathsf{keep}}(t) * \mathcal{P}_{\mathsf{send}}(t)$. We verify the CAS triple for the possible values that could be read in the premise of the fetch-and-add rule. Let t be the value that is read.

First, we show that $t < 4$ is not possible by verifying the CAS triple using the CAS$-\bot$ rule. If $t < 4$ we have $\mathcal{Q}^{EFC}(t) \Rightarrow \boxed{\lambda : (0, 0)^- * \mathsf{true}}$ (note we use here that $(0, x)^-$ is only an element of the *EFC* permission structure if $x = 0$). Hence $\mathcal{P}_{\mathsf{send}}(t) * \mathcal{Q}^{EFC} \Rightarrow \mathsf{false}$ since $(0, 0)^- \oplus (1, 0)^+$ is undefined. Therefore we can show the CAS triple using the CAS$-\bot$ rule.

Next, assume $t \geq 4$. In this case we prove the CAS triple using the CAS$-$*basic* rule. Choose

$$T := \mathcal{Q}^{EFC}(t)$$
$$A := \mathsf{emp}$$

We have

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(t) * T \Rightarrow & \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor, w = \mathsf{lsb}(v) \text{ in} \\ & \exists s \in \mathbb{Q} \cap [0,1).\, (t-4) \geq 0 \wedge (w = 1 \Rightarrow s = 0) \wedge \\ & \mathit{resource}(\mathtt{bits})^{(w=1\ ?\ 0\ :\ 1-s)} * \\ & \boxed{\alpha : 1 - w} * \boxed{\lambda : (n-1, s)^-} \\ \Rightarrow & \mathcal{Q}^{EFC}(t-4)\end{aligned}$$

Since we only transfer ghost permission, this concludes the proof. Note that the proof would also work for access type `rlx` instead of `rel` as already observed in Section 3.1.7.

6.1.4 Proof of function `unlock_shared`

We need to show

$$\{\mathsf{U}(\mathtt{bits}, \mathcal{Q}) * R^{EFC}_{\lambda}(\mathtt{bits})\} x := \mathtt{fetch\_and\_add}_{\mathtt{rel}}(\mathtt{bits}, 4) \{\mathsf{emp}\}$$

using the fetch-and-add rule. We have $P \equiv \mathit{resource}(\mathtt{bits})^{\epsilon} * \boxed{\lambda : (1,q)^+}$ for some $q \in (0,1]$. We choose

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(t) &:= P \\ \mathcal{P}_{\mathsf{keep}}(t) &:= \mathsf{emp}\end{aligned}$$

We verify the CAS triple in the premise of the fetch-and-add rule for every value t that could be read.

Suppose $t < 4$. We have $\mathcal{Q}^{EFC}(t) \Rightarrow \boxed{\lambda : (0,0)^-} * \mathsf{true}$ (here we used that if $(0,x)^-$ is in the *EFC* permission structure then $x = 0$). Hence $\mathcal{P}_{\mathsf{send}}(t) * \mathcal{Q}^{EFC}(t) \Rightarrow \mathsf{false}$ since $(0,0)^- \oplus (1,q)$ is undefined for any q. Hence we can use the CAS$-\bot$ rule to verify the CAS triple which shows that $t < 4$ cannot be read.

Next, assume $\mathsf{lsb}(t) = 1$ and $t \geq 4$. We have $\mathcal{Q}^{EFC}(t) \Rightarrow \boxed{\lambda : (n,0)^-} * \mathsf{true}$ where $n = \left\lfloor \frac{t}{4} \right\rfloor$. $\mathcal{P}_{\mathsf{send}}(t) * \mathcal{Q}^{EFC}(t) \Rightarrow \mathsf{false}$ since $(n,0)^- \oplus (1,q)^+$ is undefined for $q > 0$ and $n \geq 1$ ($n \geq 1$ follows from $t \geq 4$). Hence we can use the CAS$-\bot$ rule to verify the CAS triple which shows that $\mathsf{lsb}(t) = 1$ cannot be read.

Next, assume $t \geq 4$ and $\mathsf{lsb}(t) = 0$. We have

$$\begin{aligned}\mathcal{Q}^{EFC}(t) \Leftrightarrow & \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in} \\ & \exists s \in \mathbb{Q} \cap [0,1).\, t \geq 0 \wedge \mathit{resource}(\mathtt{bits})^{1-s} * \boxed{\alpha : 1} * \boxed{\lambda : (n,s)^-}\end{aligned}$$

We choose

$$T := \mathcal{Q}^{EFC}(t)$$
$$A := \mathsf{emp}$$

We have

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(t) * T \Rightarrow & \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in} \\ & \exists s \in \mathbb{Q} \cap [0,1).\, t - 4 \geq 0 \wedge \mathit{resource}(\mathtt{bits})^{1-(s-q)} * \\ & \boxed{\alpha : 1} * \boxed{\lambda : (n-1, s-q)^-} \\ \Rightarrow & \mathcal{Q}^{EFC}(t-4)\end{aligned}$$

where we used that $s - q \geq 0$ (since otherwise $\boxed{\lambda : (n,s)^-} * \boxed{\lambda : (1,q)^+}$ would be undefined) and $s - q < 1$ (since $s < 1$). Because the fetch-and-add has access type `rel` this concludes the proof.

6.1.5 Proof of function `try_lock`

We show the proof for the function `try_lock`. We need to show

$$\left\{\mathsf{U}(\mathtt{bits}, \mathcal{Q}^{EFC})\right\}\, v := \mathtt{CAS}_{\mathtt{rel\_acq}}(\mathtt{bits}, 0, 1) \left\{ \left(\begin{array}{l} \mathtt{v} = 0 \ ? \\ \ \mathit{resource}(\mathtt{bits})^1 * \boxed{\alpha : 1} : \\ \ \mathsf{U}(\mathtt{bits}, \mathcal{Q}) \end{array} \right) \right\}$$

We use the CAS$-$*basic* rule. We have $P \equiv \mathsf{emp}$, hence $\mathcal{P}_{\mathsf{send}}(t) := \mathcal{P}_{\mathsf{keep}}(t) := \mathsf{emp}$. It holds that

$$\mathcal{Q}^{EFC}(0) \Leftrightarrow \mathit{resource}(\mathtt{bits})^1 * \boxed{\alpha : 1} * \boxed{\lambda : (0,0)^-}$$

where we used that $(0,x)^-$ is in the *EFC* permission structure iff $x = 0$. Choose

$$T := \boxed{\alpha : 0} * \boxed{\lambda : (0,0)^-}$$
$$A := \mathit{resource}(\mathtt{bits})^1 * \boxed{\alpha : 1}$$

We have $T \Rightarrow \mathcal{Q}^{EFC}(1)$. Furthermore, since $A \Leftrightarrow W_\alpha^{EFC}$ and the CAS has access type `rel_acq`, this concludes the proof. Actually it suffices to have access type `acq` since no permission is given up by the thread as was also observed in Section 3.1.7.

6.1.6 Proof of function `unlock`

We adjust the fetch-and-add rule as in Section 3.1.8 to be able to verify the $\texttt{fetch\_and\_and}_{\texttt{rel}}(\texttt{bits}, \sim 1)$ operation.

Case 1: `getRead` does not hold. We need to show

$$\{\ W_\alpha^{EFC}(\texttt{bits})\ \}\ \texttt{fetch\_and\_and}_{\texttt{rel}}(\texttt{bits}, \sim 1)\ \{\ \texttt{emp}\ \}$$

The proof is completely analogous to the proof in Section 3.1.8, so we do not show it.

Case 2: `getRead` holds. We need to show

$$\left\{\ W_\alpha^{EFC}(\texttt{bits}) * \boxed{\lambda : (1,0)^+}\ \right\}\ \texttt{fetch\_and\_and}_{rel}(\texttt{bits}, \sim 1)\ \{\ R_\lambda(\texttt{bits})\ \}$$

We have $P \equiv \mathit{resource}(\texttt{bits})^1 * \boxed{\alpha : 1} * \boxed{\lambda : (1,0)^+}$. Choose

$$\mathcal{P}_{\mathsf{send}}(t) := \mathit{resource}(\texttt{bits})^{\frac{1}{2}} * \boxed{\alpha : 1} * \boxed{\lambda : (1,0)^+}$$
$$\mathcal{P}_{\mathsf{keep}}(t) := \mathit{resource}(\texttt{bits})^{\frac{1}{2}}$$

We show the CAS triple in the premise of the fetch-and-add rule for all values that could be read using the CAS$-$*basic* rule. Let t be the value that is read.

Suppose $\mathsf{lsb}(t) = 0$. In this case $\mathcal{Q}^{EFC} \Rightarrow \boxed{\alpha : 1} * \mathsf{true}$ and hence $P * \mathcal{Q}^{EFC} \Rightarrow \mathsf{false}$. Therefore the CAS triple can be shown using the CAS$-\bot$ rule which means such a value cannot be read.

Next, suppose $t < 4$. In this case $\mathcal{Q}^{EFC} \Rightarrow \boxed{\lambda : (0,0)^-} * \mathsf{true}$. Hence $P * \mathcal{Q}^{EFC} \Rightarrow \mathsf{false}$. Therefore the CAS triple can be shown using the CAS$-\bot$ rule which means such a value cannot be read.

Finally, suppose $\mathsf{lsb}(t) = 1$ and $t \geq 4$. In this case we have

$$\mathcal{Q}^{EFC}(t) \Leftrightarrow \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in } t \geq 0 \wedge \boxed{\alpha : 0} * \boxed{\lambda : (n,0)^-}$$

Choose

$$T := \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in } t \geq 0 \wedge \boxed{\alpha : 0} * \boxed{\lambda : (n+1, \tfrac{1}{2})^-}$$
$$A := \boxed{\lambda : (1, \tfrac{1}{2})^+}$$

We have

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(t) * T \Rightarrow \text{let } n = \left\lfloor \frac{t}{4} \right\rfloor \text{ in }& \\ t \geq 0 \wedge \mathit{resource}(\texttt{bits})^{\frac{1}{2}} * \boxed{\alpha : 1} * \boxed{\lambda : (n, \tfrac{1}{2})^-}& \\ \Rightarrow \mathcal{Q}^{EFC}(\mathsf{clearlsb}(t))&\end{aligned}$$

Since the fetch-and-add is of access type `rel`, we can release resources into the location invariant without any modalities and A only consists of ghost state, so we are fine. Furthermore, it holds that

$$\mathcal{P}_{\mathsf{keep}}(t) * A \Rightarrow R_{\lambda}^{EFC}$$

which concludes the proof.

6.1.7 Proof of function `unlock_and_lock_shared`

The proof for `unlock_and_lock_shared` consists of essentially the proof of the first fetch-and-add operation in `try_lock_shared` where there is a writer, i.e. where $\boxed{\lambda : (1,0)^{+}}$ is obtained and the remaining part relies on the specification of `unlock`.

6.2 Folly barrier

Barrier was introduced and proved in Section 3.2. In this Section we give a different proof using the *EFC* permission structure.

6.2.1 Specification

We verify the same specification as in Section 3.2.3 for the function `wait`.

6.2.2 Location invariant

The location invariant using the *EFC* permission structure for the atomic location `VRC` is given by:

$$\begin{aligned}\mathcal{Q}^{EFC}(r,v) := \; & \exists s \in \{0,1\}.\, 0 \leq r \wedge r \leq v \wedge (v < n \Rightarrow s = 0) \wedge \boxed{\eta : (r,s)^{-}} * \\ & ([(r,v) \neq (0,n) \wedge (v \leq n)] \; ? \; tok.val \overset{1-s}{\mapsto} \_ : \mathsf{emp})\end{aligned}$$

which is similar to the location invariant given in Section 3.2.8. η is a ghost location governed by the *EFC* permission structure.

The idea of the location invariant is as follows. The ghost location η tracks all the threads that enter `wait` (note: here we only need one ghost location to do this compared to two ghost locations in the location invariant given by Section 3.2.8). For each thread that has entered and has not yet left `wait` one entity of η is given, hence the entity count is given by r. In general the permission amount associated with such an entity is 0. But the nth thread that enters wait gets an entity with permission value 1 along with the token after executing the first fetch-and-add operation. Hence the

entity sum is always either 0 or 1 and is given by the existentially quantified variable s. Right after the nth thread has executed the first fetch-and-add operation the entity sum s is 1. This is another example where associating zero permission value with an entity is useful. As explained in Section 3.2.4, if $[(r, v) \neq (0, n) \wedge (v \leq n)]$ does not hold then the barrier has finished its task and the token need not be tracked any more.

6.2.3 Proof outline

The proof outline for `wait` with respect to $\mathcal{Q}^{EFC}$ is given by Figure 15.

6.2.4 Proof of first RMW

We give the proof for the first fetch-and-add operation. We have to show

$$
\begin{array}{l}
\{\mathsf{U}(\texttt{VRC}, Q)\} \\
(r_1, v_1) := \texttt{fetch\_and\_add}_{\mathsf{acq}}(\texttt{VRC}, (1,1)) \\
\left\{ \left[\begin{array}{l} v_1 + 1 = n \ ?\ tok.val \overset{1}{\mapsto} \_ * \boxed{\eta : (1,1)^+} \ : \\ \left(v_1 + 1 < n \ ? \ \boxed{\eta : (1,0)^+} \ : \ \boxed{\eta : (1,0)^+} * \mathsf{true} \right) \end{array} \right] \right\}
\end{array}
$$

using the fetch-and-add rule. We have $P \equiv \mathsf{emp}$, hence $\mathcal{P}_{\mathsf{send}}(t) := \mathcal{P}_{\mathsf{keep}}(t) := \mathsf{emp}$. We show the CAS triple in the fetch-and-add rule for every value that could be read using the CAS−*basic* rule. Let (r, v) be the value read (we assume that $r, v \geq 0$ since negative values cannot be read anyway, as can be seen directly from the invariant).

Case $v + 1 < n$**.** We have

$$\mathcal{Q}^{EFC}(r, v) := 0 \leq r \wedge r \leq v \wedge \boxed{\eta : (r, 0)^-} * tok.val \overset{1}{\mapsto} \_$$

Choose

$$
\begin{array}{l}
T := 0 \leq r \wedge r \leq v \wedge \boxed{\eta : (r+1, 0)^-} * tok.val \overset{1}{\mapsto} \_ \\
A := \boxed{\eta : (1,0)^+}
\end{array}
$$

It holds that $\mathcal{Q}^{EFC}(r, v) \Leftrightarrow A * T$ and $T \Rightarrow \mathcal{Q}^{EFC}(r+1, v+1)$. Furthermore, since A only consists of ghost state and is exactly the permission that is needed, we have shown this case.

Case $v + 1 = n$**.** We have

$$\mathcal{Q}^{EFC}(r, v) := 0 \leq r \wedge r \leq v \wedge \boxed{\eta : (r, 0)^-} * tok.val \overset{1}{\mapsto} \_$$

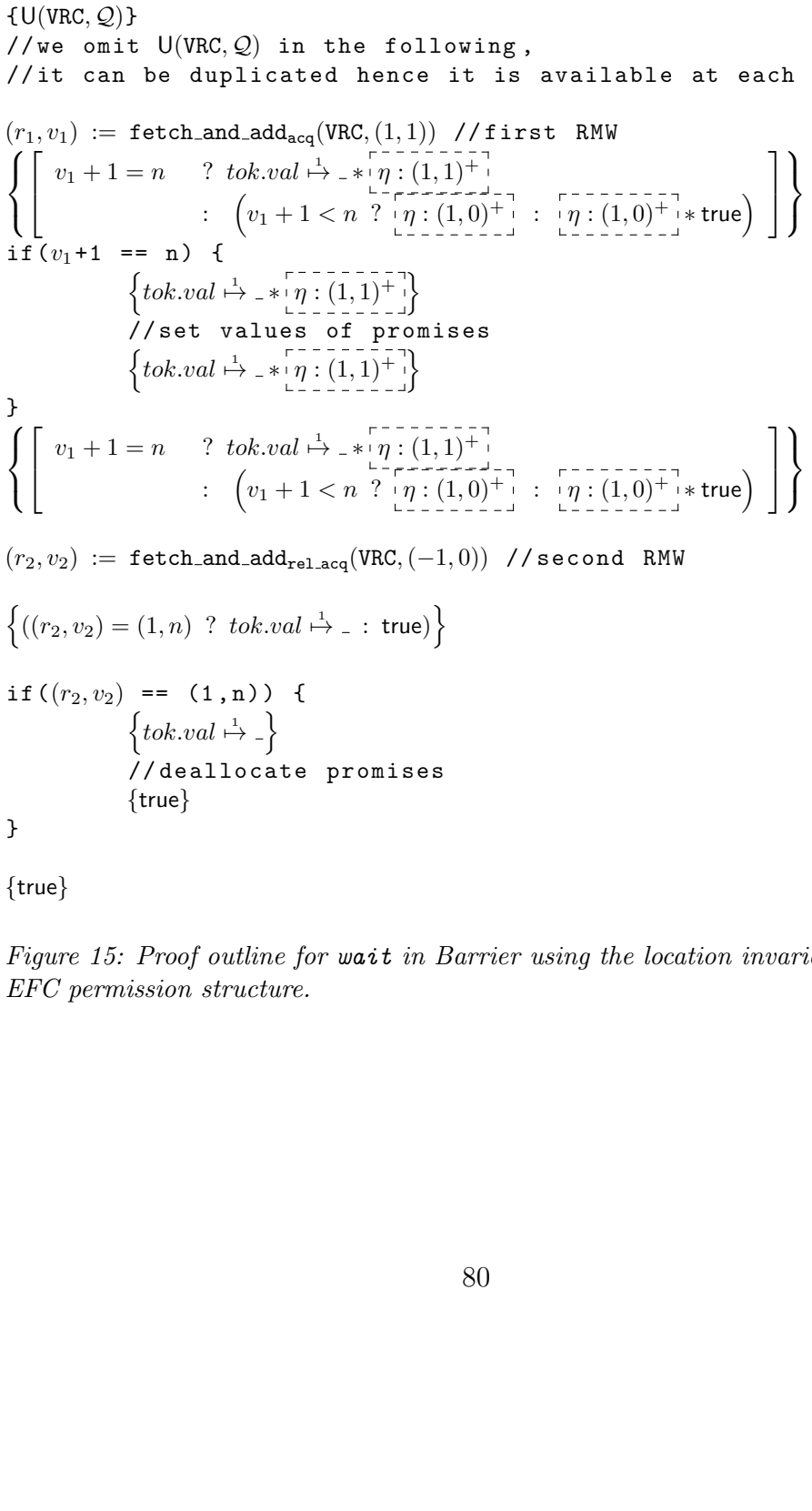

Figure 15: Proof outline for `wait` *in Barrier using the location invariant with the EFC permission structure.*

Choose

$$T := 0 \le r \land r \le v \land \boxed{\eta : (r+1,1)^-}$$
$$A := \boxed{\eta : (1,1)^+} * tok.val \overset{1}{\mapsto} \_$$

It holds that $\mathcal{Q}^{EFC}(r,v) \Leftrightarrow A * T$ and

$$\begin{aligned} T \Rightarrow_{\text{choosing } s:=1} & \exists s \in \{0,1\}.\, 0 \le r+1 \land r+1 \le v+1 \land \\ & (v+1 < n \Rightarrow s = 0) \land \boxed{\eta : (r+1,s)^-} * tok.val \overset{1-s}{\mapsto} \_ \\ \Rightarrow \mathcal{Q}^{EFC}(r+1, v+1) & \end{aligned}$$

where we used that $[(r+1, v+1) \neq (0,n) \land (v+1 \le n)]$ holds. Since additionally the fetch-and-add has access type `acq` , we need not worry about modalities for the permission extracted from the invariant. This concludes the case $v+1 = n$.

Case $v+1 > n$**.** We have

$$\begin{aligned} & \exists s \in \{0,1\}.\, 0 \le r \land r \le v \land \boxed{\eta : (r,s)^-} * \\ & ([(r,v) \neq (0,n) \land (v \le n)] \ ?\ tok.val \overset{1-s}{\mapsto} \_ : \mathsf{emp}) \end{aligned}$$

Choose

$$T := \exists s \in \{0,1\}.\, 0 \le r \land r \le v \land \boxed{\eta : (r+1,s)^-}$$
$$A := \boxed{\eta : (1,0)^+} * \mathsf{true},$$

It holds that $\mathcal{Q}^{EFC}(r,v) \Rightarrow A * T$. Note that true absorbs the permission that is potentially guarded by the condition $[(r,v) \neq (0,n) \land (v \le n)]$. Furthermore, since $v+1 > n$ the condition $[(r+1, v+1) \neq (0,n) \land (v+1 \le n)]$ does not hold, therefore we conclude that $T \Rightarrow \mathcal{Q}^{EFC}(r+1, v+1)$. Since the fetch-and-add has access type `acq` we need not worry about any modalities for the permission extracted from the location invariant. This concludes the case $v+1 > n$.

6.2.5 Proof of second RMW

We give the proof for the second fetch-and-add operation. We have to show

$$\left\{ \mathsf{U}(\mathcal{Q}^{EFC}(r,v)) * \left[\begin{array}{l} v_1 + 1 = n \ ?\ tok.val \overset{1}{\mapsto} \_ * \boxed{\eta : (1,1)^+} \ : \\ \qquad \left(v_1 + 1 < n \ ?\ \boxed{\eta : (1,0)^+} \ : \ \boxed{\eta : (1,0)^+} * \mathsf{true} \right) \end{array} \right] \right\}$$
$$(r_2, v_2) := \texttt{fetch\_and\_add}_{\texttt{rel\_acq}}(\texttt{VRC}, (-1,0))$$
$$\left\{ ((r_2, v_2) = (1,n) \ ?\ tok.val \overset{1}{\mapsto} \_ : \mathsf{true}) \right\}$$

using the fetch-and-add rule. We have

$$P \equiv \left[\begin{array}{l} v_1 + 1 = n \ ?\ tok.val \overset{1}{\mapsto} \_ * \boxed{\eta : (1,1)^+} \ : \\ \qquad \left(v_1 + 1 < n \ ?\ \boxed{\eta : (1,0)^+} \ : \ \boxed{\eta : (1,0)^+} * \mathsf{true}\right) \end{array}\right]$$

Instead of writing down $\mathcal{P}_{\mathsf{send}}(r, v)$ and $\mathcal{P}_{\mathsf{keep}}(r, v)$ for general (r, v) we will give the definitions in the different (disjoint) subcases for the sake of presentation. We make a case distinction on the value (r, v) that is read to verify the CAS triple in the premise of the fetch-and-add rule.

Case 1: $r = 0$**.** We show that $(0, v)$ cannot be read by verifying the CAS triple using the CAS$-\bot$ rule. In this case define

$$\mathcal{P}_{\mathsf{send}}(0, v) := P$$
$$\mathcal{P}_{\mathsf{keep}}(0, v) := \mathsf{emp}$$

Hence

$$\mathcal{P}_{\mathsf{send}}(0, v) \Rightarrow (v_1 + 1 = n \ ?\ tok.val \overset{1}{\mapsto} \_ * \boxed{\eta : (1,1)^+} : \boxed{\eta : (1,0)^+} * \mathsf{true})$$

and we have

$$\mathcal{Q}^{EFC}(0, v) \Rightarrow \boxed{\eta : (0,0)^-} * \mathsf{true}$$

where we used that $(0, x)^-$ is an element in the *EFC* permission structure iff $x = 0$. We conclude that

$$\mathcal{P}_{\mathsf{send}}(0, v) * \mathcal{Q}^{EFC}(0, v) \Rightarrow \mathsf{false}$$

where we use that $(0, 0)^- \oplus (1, x)^+$ is undefined for any x. Therefore we can use the CAS$-\bot$ rule to verify the CAS triple. This concludes this case.

Case 2: $(r, v) = (1, n)$**.** In this case we show the CAS triple using the CAS$-$*param* rule. Define in this case

$$\mathcal{P}_{\mathsf{send}}(1, n) := \left[\begin{array}{l} v_1 + 1 = n \ ?\ \boxed{\eta : (1,1)^+} \ : \\ \qquad \left(v_1 + 1 < n \ ?\ \boxed{\eta : (1,0)^+} \ : \ \boxed{\eta : (1,0)^+}\right) \end{array}\right]$$
$$\mathcal{P}_{\mathsf{keep}}(1, n) := \left[\begin{array}{l} v_1 + 1 = n \ ?\ tok.val \overset{1}{\mapsto} \_ \ : \\ \qquad (v_1 + 1 < n \ ?\ \mathsf{emp} \ : \ \mathsf{true}) \end{array}\right]$$

It holds that $P \Leftrightarrow \mathcal{P}_{\mathsf{send}}(1, n) * \mathcal{P}_{\mathsf{keep}}(1, n)$. We have

$$\begin{aligned} \mathcal{Q}^{EFC}(1, n) \Rightarrow & \exists s \in \{0, 1\}. \boxed{\eta : (1, s)^-} * tok.val \overset{1-s}{\mapsto} \_ \\ \Rightarrow & \exists s. (s = 1 \ ?\ \boxed{\eta : (1,1)^-} : \boxed{\eta : (1,0)^-} * tok.val \overset{1}{\mapsto} \_ \wedge (s = 0)) \end{aligned}$$

Choose

$$\mathcal{T}(z) := (z = 1\ ?\ \boxed{\eta : (1,1)^-} : \boxed{\eta : (1,0)^-} \wedge (s = 0))$$
$$\mathcal{A}(z) := (z = 1\ ?\ \mathsf{emp} : tok.val \overset{1}{\mapsto} \_)$$

It holds that $\mathcal{Q}^{EFC}(1,n) \Rightarrow \exists z.\, \mathcal{A}(z) * \mathcal{T}(z)$. We show

$$\forall z.(\mathcal{P}_{\mathsf{send}}(1,n) * \mathcal{T}(z) \Rightarrow \mathcal{Q}^{EFC}(0,n) \wedge \varphi(z))$$

where $\varphi(z)$ must be pure. We choose

$$\varphi(z) := (v_1 + 1 = n\ ?\ z = 1 : z = 0)$$

Clearly the chosen $\varphi(z)$ is pure for every z.

Let z be arbitrary. We make a case distinction on v_1. If $v_1 + 1 = n$ we have

$$\mathcal{P}_{\mathsf{send}}(1,n) \Leftrightarrow \boxed{\eta : (1,1)^+}$$

Hence

$$\begin{aligned}
\mathcal{P}_{\mathsf{send}}(1,n) * \mathcal{T}(z) \Rightarrow\ & (z = 1\ ? \\
& \boxed{\eta : (1,1)^+} * \boxed{\eta : (1,1)^-} : \\
& (z = 0) \wedge \boxed{\eta : (1,1)^+} * \boxed{\eta : (1,0)^-}) \\
\Rightarrow & (z = 1\ ?\ \boxed{\eta : (0,0)^-} : \mathsf{false}) \\
\Rightarrow & \boxed{\eta : (0,0)^-} \wedge (z = 1) \\
\Rightarrow & \mathcal{Q}^{EFC}(0,n) \wedge \varphi(z)
\end{aligned}$$

where we used that $(1,1)^+ \oplus (1,0)^-$ is undefined.

If $v_1 + 1 \neq n$ we have

$$\mathcal{P}_{\mathsf{send}}(1,n) \Leftrightarrow \boxed{\eta : (1,0)^+}$$

Hence

$$\begin{aligned}
\mathcal{P}_{\mathsf{send}}(1,n) * \mathcal{T}(z) \Rightarrow\ & (z = 1\ ? \\
& \boxed{\eta : (1,0)^+} * \boxed{\eta : (1,1)^-} : \\
& \boxed{\eta : (1,0)^+} * \boxed{\eta : (1,0)^-} \wedge (z = 0)) \\
\Rightarrow & (z = 1\ ?\ \mathsf{false} : (z = 0) \wedge \boxed{\eta : (0,0)^-}) \\
\Rightarrow & \boxed{\eta : (0,0)^-} \wedge (z = 0) \\
\Rightarrow & \mathcal{Q}^{EFC}(0,n) \wedge \varphi(z)
\end{aligned}$$

where we used that $(1,0)^+ \oplus (1,1)^-$ is undefined. So we have shown

$$\mathcal{P}_{\mathsf{send}}(1,n) * \mathcal{T}(z) \Rightarrow \mathcal{Q}^{EFC}(0,n) \wedge \varphi(z)$$

for arbitrary z. We still need to prove that

$$\mathcal{P}_{\mathsf{keep}}(1,n) * \exists z.\, A(z) \wedge \varphi(z) \Rightarrow tok.val \overset{1}{\mapsto} \_$$

If $v_1 + 1 = n$ then we have $\varphi(z) = (z = 1)$, therefore $\exists z.\, A(z) \wedge \varphi(z) \Rightarrow \mathsf{emp}$ and $\mathcal{P}_{\mathsf{keep}}(1,n) \Leftrightarrow tok.val \overset{1}{\mapsto} \_$. We conclude that in the case $v_1 + 1 = n$ the statement holds.

If $v_1 + 1 < n$ then we have $\varphi(z) = (z = 0)$, therefore $\exists z.\, A(z) \wedge \varphi(z) \Rightarrow tok.val \overset{1}{\mapsto} \_$ and $\mathcal{P}_{\mathsf{keep}}(1,n) \Leftrightarrow \mathsf{emp}$. We conclude that in the case $v_1 + 1 < n$ the statement holds.

If $v_1 + 1 > n$ then we have $\varphi(z) = (z = 0)$, therefore $\exists z.\, A(z) \wedge \varphi(z) \Rightarrow tok.val \overset{1}{\mapsto} \_$ and $\mathcal{P}_{\mathsf{keep}}(1,n) \Leftrightarrow \mathsf{true}$. So in this case we can show $\mathcal{P}_{\mathsf{keep}}(1,n) * \exists z.\, A(z) \wedge \varphi(z) \Rightarrow tok.val \overset{1}{\mapsto} \_ * \mathsf{true}$. As can be seen from the proof for the first fetch-and-add, the true potentially absorbed $tok.val \overset{1}{\mapsto} \_$ and nothing else. Hence we can show that true can be replaced by emp (using a more precise specification for the first fetch-and-add). Therefore the statement also holds in the case $v_1 + 1 > n$.

Case 3: $r > 0, v \leq n$ **and** $(r,v) \neq (1,n)$**.** In this case we show the CAS triple using the CAS$-$*basic* rule. In the proof we use that $[(r,v) \neq (0,n) \wedge (v \leq n)]$ and $[(r-1,v) \neq (0,n) \wedge (v \leq n)]$ both hold.

We define in this case

$$\mathcal{P}_{\mathsf{send}}(r,v) := (v_1 + 1 \leq n \;?\; P : \boxed{\eta : (1,0)^+})$$
$$\mathcal{P}_{\mathsf{keep}}(r,v) := (v_1 + 1 \leq n \;?\; \mathsf{emp} : \mathsf{true})$$

It holds that $P \Leftrightarrow \mathcal{P}_{\mathsf{send}}(r,v) * \mathcal{P}_{\mathsf{keep}}(r,v)$. We have

$$\mathcal{Q}^{EFC}(r,v) \Leftrightarrow \exists s \in \{0,1\}.\, 0 \leq r \wedge r \leq v \wedge (v < n \Rightarrow s = 0) \wedge \boxed{\eta : (r,s)^-} * tok.val \overset{1-s}{\mapsto} \_$$

Choose

$$T := \mathcal{Q}^{EFC}(r,v)$$
$$A := \mathsf{emp}$$

If $v_1 + 1 = n$, then we have $\mathcal{P}_{\mathsf{send}}(r,v) \Leftrightarrow tok.val \overset{1}{\mapsto} \_ * \boxed{\eta : (1,1)^+}$, hence

$$\mathcal{P}_{\mathsf{send}}(r,v) * T \Rightarrow 0 \leq r-1 \wedge r-1 \leq v \wedge (v < n \Rightarrow s = 0) \wedge \boxed{\eta : (r-1,0)^-} * tok.val \overset{1}{\mapsto} \_ \Rightarrow_{\text{choosing } s:=0} \mathcal{Q}^{EFC}(r-1,v)$$

where we used that the existentially quantified variable s in T must be 1 in this case otherwise $\mathcal{P}_{\mathsf{send}}(r, v) * T \Rightarrow \mathsf{false}$.

If $v_1 + 1 \neq n$, then we have $\mathcal{P}_{\mathsf{send}}(r, v) \Leftrightarrow \boxed{\eta : (1, 0)^+}$, hence

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(r, v) * T \Rightarrow & \exists s \in \{0, 1\}.\, 0 \leq r - 1 \wedge r - 1 \leq v \wedge (v < n \Rightarrow s = 0) \wedge \\ & \boxed{\eta : (r - 1, s)^-} * tok.val \overset{1-s}{\mapsto} \_ \\ \Rightarrow & \mathcal{Q}^{EFC}(r, v)\end{aligned}$$

This concludes the proof for case 3.

Case 4: $r > 0, v > n$**.** In this case define

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(r, v) &:= (v_1 + 1 = n \ ? \ \boxed{\eta : (1, 1)^+} : \boxed{\eta : (1, 0)^+}) \\ \mathcal{P}_{\mathsf{keep}}(r, v) &:= (v_1 + 1 = n \ ? \ tok.val \overset{1}{\mapsto} \_ : (v_1 + 1 < n \ ? \ \mathsf{emp} : \mathsf{true}))\end{aligned}$$

it holds that $P \Leftrightarrow \mathcal{P}_{\mathsf{send}}(r, v) * \mathcal{P}_{\mathsf{keep}}(r, v)$. We have

$$\mathcal{Q}^{EFC}(r, v) \Leftrightarrow \exists s \in \{0, 1\}.\, 0 \leq r \wedge r \leq v \wedge \boxed{\eta : (r, s)^-}$$

Choose

$$\begin{aligned}T &:= \mathcal{Q}^{EFC}(r, v) \\ A &:= \mathsf{emp}\end{aligned}$$

If $v_1 + 1 = n$ then $\mathcal{P}_{\mathsf{send}}(r, v) \Leftrightarrow \boxed{\eta : (1, 1)^+}$, hence

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(r, v) * T &\Rightarrow 0 \leq r - 1 \wedge r - 1 \leq v \wedge \boxed{\eta : (r - 1, 0)^-} \\ &\Rightarrow_{\text{choosing } s:=0} \mathcal{Q}^{EFC}(r - 1, v)\end{aligned}$$

where we used that the existentially quantified variable s in T must be 1 otherwise $\mathcal{P}_{\mathsf{send}}(r, v) * T \Rightarrow \mathsf{false}$.

If $v_1 + 1 \neq n$ then $\mathcal{P}_{\mathsf{send}}(r, v) \Leftrightarrow \boxed{\eta : (1, 0)^+}$, hence

$$\begin{aligned}\mathcal{P}_{\mathsf{send}}(r, v) * T \Rightarrow & \exists s \in \{0, 1\}.\, 0 \leq r - 1 \wedge r - 1 \leq v \wedge \boxed{\eta : (r - 1, s)^-} \\ \Rightarrow & \mathcal{Q}^{EFC}(r - 1, v)\end{aligned}$$

This concludes the proof for case 4.

6.3 Rust atomic reference counter

ARC was introduced and proved in Section 3.3. In this section we give a different proof using the *EFC* permission structure.

6.3.1 Location invariant

The location invariant using the *EFC* permission structure for the atomic location `a.count` is given by:

$$\begin{aligned}\mathcal{Q}^{EFC}_{a,v}(c) := \ & \textbf{if } c = 0 \textbf{ then } \boxed{\omega : (0,0)^-} \\ & \textbf{else } \exists f \in \mathbb{Q} \cap [0,1].\, \texttt{a.data} \overset{f}{\mapsto} v * \boxed{\omega : (c, 1-f)^-}\end{aligned}$$

where ω is a ghost location governed by the *EFC* permission structure. The idea of the location invariant is that if $c > 0$ then there are c ARC resources held by threads which in total account for $1 - f$ permission to `a.data` (f is an existentially quantified variable). Furthermore, the remaining permission to `a.data` is in the location invariant.

By using the *EFC* permission structure we can express this simply by asserting $\boxed{\omega : (c, 1-f)^-}$ inside the location invariant for $c > 0$. If $c = 0$ then all the entities should be have been collected. We can express this by asserting $\boxed{\omega : (0,0)^-}$. This, in our opinion, is more direct than the location invariant in Section 3.3.2. We reduced the number of ghost locations from two to one. While the ghost location we use has more components, it makes it easier to express the complete situation. In particular, we do not need any correlations between ghost locations as is the case for the location invariant in Section 3.3.2.

6.3.2 Specification

We prove the same specification as in Section 3.3.3 except that we define the ARC resource using $\mathsf{ARC}^{EFC}_{\omega}(a, v)$:

$$\begin{aligned}\mathsf{ARC}^{EFC}_{\omega}(a, v) := \ & \mathsf{U}(\texttt{a.count}, \mathcal{Q}^{EFC}_{a,v}) * \\ & \exists q \in \mathbb{Q} \cap (0,1].\, \texttt{a.data} \overset{q}{\mapsto} v * \boxed{\omega : (1,q)^+}\end{aligned}$$

We only need one ghost location here that consists of one entity with the same permission amount as is held to `a.data`.

6.3.3 Proof of function `new`

The proof outline for the function `new` is given in Figure 16. In the proof we use that ghost state can be introduced at any time, hence we introduce full permission $\boxed{\omega : (0,0)^-}$. Furthermore, we use that ghost state is agnostic towards modalities and that

$$\mathsf{emp} \Leftrightarrow \triangle \mathsf{emp}$$

$$\mathsf{emp} \Leftrightarrow \texttt{a.data} \overset{0}{\mapsto} v$$

```
ref new(v) {
  {emp}
  a = alloc();
  {RMWAcq(a.count, Q^EFC_{a,v}) * Rel(a.count, Q^EFC_{a,v}) * a.data ↦^1 _ * ⌜ω : (0,0)^-⌝}
  [a.data]_na := v;
  {RMWAcq(a.count, Q^EFC_{a,v}) * Rel(a.count, Q^EFC_{a,v}) * a.data ↦^1 v * ⌜ω : (0,0)^-⌝}
  { RMWAcq(a.count, Q^EFC_{a,v}) * Rel(a.count, Q^EFC_{a,v}) *
    a.data ↦^1 v * a.data ↦^0 v * ⌜ω : (1,1)^-⌝ * ⌜ω : (1,1)^+⌝ }
  { RMWAcq(a.count, Q^EFC_{a,v}) * Rel(a.count, Q^EFC_{a,v}) *
    a.data ↦^1 v * ⌜ω : (1,1)^+⌝ * △(a.data ↦^0 v * ⌜ω : (1,1)^-⌝) }
  [a.count]_rlx := 1
  {U(a.count, Q^EFC_{a,v}) * a.data ↦^1 v * ⌜ω : (1,1)^+⌝}
  {ARC^EFC_ω(a, v)}
  return a;
}
```

Figure 16: Proof outline for function **new** *with respect to the new location invariant*

which is needed to be able to perform the relaxed write where we also use that

$$\begin{aligned}&\texttt{a.data} \overset{0}{\mapsto} v * \boxed{\omega : (1,1)^-} \Rightarrow_{\text{choosing } f:=0}\\ &\exists f \in \mathbb{Q} \cap [0,1].\, \texttt{a.data} \overset{f}{\mapsto} v * \boxed{\omega : (1,1-f)^-} \Rightarrow\\ &\mathcal{Q}^{EFC}_{a,v}(1)\end{aligned}$$

Finally, we use that

$$\begin{aligned}&\texttt{a.data} \overset{1}{\mapsto} v * \boxed{\omega : (1,1)^+} \Rightarrow_{\text{choosing } q:=1}\\ &\exists q \in \mathbb{Q} \cap (0,1].\, \texttt{a.data} \overset{q}{\mapsto} v * \boxed{\omega : (1,q)^+}\end{aligned}$$

6.3.4 Proof of function `clone`

We need to show

$$\left\{\mathsf{ARC}^{EFC}_{\omega}(a,v)\right\} \texttt{x} := \texttt{fetch\_and\_add}_{\texttt{rlx}}(\texttt{a.count}, 1) \left\{ \begin{array}{l} \mathsf{ARC}^{EFC}_{\omega}(a,v) * \\ \mathsf{ARC}^{EFC}_{\omega}(a,v) \end{array} \right\}$$

using the fetch-and-add rule. We have

$$P \equiv \exists q \in \mathbb{Q} \cap (0,1].\, \texttt{a.data} \overset{q}{\mapsto} v * \boxed{\omega : (1,q)^+}$$

and we choose

$$\mathcal{P}_{\mathsf{send}}(t) := (t = 0 \ ? \ P : \mathsf{emp})$$
$$\mathcal{P}_{\mathsf{keep}}(t) := (t = 0 \ ? \ \mathsf{emp} : P)$$

Clearly it holds that $P \Leftrightarrow \mathcal{P}_{\mathsf{send}}(t) * \mathcal{P}_{\mathsf{keep}}(t)$ for every t. We need to verify the CAS triple in the premise of the fetch-and-add rule for every possible value t that can be read. First, we show that $t = 0$ cannot be read by verifying the CAS triple using the CAS$-\bot$ rule.

Suppose $t = 0$. We have

$$\mathcal{P}_{\mathsf{send}}(0) \Leftrightarrow \exists q \in \mathbb{Q} \cap (0, 1]. \, \mathtt{a.data} \overset{q}{\mapsto} v * \boxed{\omega : (1, q)^+}$$

Furthermore, we have

$$\mathcal{Q}^{EFC}_{a,v}(0) \Leftrightarrow \boxed{\omega : (0, 0)^-}$$

hence $\mathcal{P}_{\mathsf{send}}(0) * \mathcal{Q}^{EFC}_{a,v}(0) \Rightarrow \mathsf{false}$ because $(0, 0)^- \oplus (1, q)^+$ is undefined since $0 - 1 < 0$. Therefore we can use the CAS$-\bot$ rule to conclude that $t = 0$ cannot be read.

Next, suppose $t > 0$. We have

$$\begin{aligned}
&\mathcal{P}_{\mathsf{send}}(0) \Leftrightarrow \mathsf{emp} \\
&\mathcal{Q}^{EFC}_{a,v}(t) \Leftrightarrow \exists f \in \mathbb{Q} \cap [0, 1]. \, \mathtt{a.data} \overset{f}{\mapsto} v * \boxed{\omega : (t, 1 - f)^-} \\
&\qquad\qquad \Leftrightarrow \exists f \in \mathbb{Q} \cap [0, 1]. \, \mathtt{a.data} \overset{f}{\mapsto} v * \boxed{\omega : (t + 1, 1 - f)^-} * \boxed{\omega : (1, 0)^+}
\end{aligned}$$

We choose

$$T := \exists f \in \mathbb{Q} \cap [0, 1]. \, \mathtt{a.data} \overset{f}{\mapsto} v * \boxed{\omega : (t + 1, 1 - f)^-}$$
$$A := \boxed{\omega : (1, 0)^+}$$

and it holds that $\mathcal{Q}^{EFC}_{a,v}(t) \Rightarrow A * T$. Furthermore, it holds that $T \Rightarrow \mathcal{Q}^{EFC}_{a,v}(t + 1)$. This concludes the proof of the CAS triple (where we used the CAS$-$*basic* rule). It remains to be shown that $\mathcal{P}_{\mathsf{keep}}(t) * A$ is enough to

satisfy the postcondition of `clone`. We have

$$\begin{aligned}
\mathcal{P}_{\mathsf{keep}}(t) * A &\Rightarrow \exists q \in \mathbb{Q} \cap (0,1].\, \texttt{a.data} \overset{q}{\mapsto} v * \boxed{\omega : (1,q)^+} * \boxed{\omega : (1,0)^+} \\
&\Leftrightarrow \exists q \in \mathbb{Q} \cap (0,1].\, \texttt{a.data} \overset{q}{\mapsto} v * \boxed{\omega : (2,q)^+} \\
&\Leftrightarrow \exists q \in \mathbb{Q} \cap (0,1].\, \texttt{a.data} \overset{\frac{q}{2}}{\mapsto} v * \boxed{\omega : (1,\tfrac{q}{2})^+} * \\
&\qquad \texttt{a.data} \overset{\frac{q}{2}}{\mapsto} v * \boxed{\omega : (1,\tfrac{q}{2})^+} \\
&\Rightarrow_{\text{choosing } q' := \frac{q}{2}, q'' := \frac{q}{2}} \exists q' \in \mathbb{Q} \cap (0,1].\, \texttt{a.data} \overset{q'}{\mapsto} v * \boxed{\omega : (1,q')^+} * \\
&\qquad \exists q'' \in \mathbb{Q} \cap (0,1].\, \texttt{a.data} \overset{q''}{\mapsto} v * \boxed{\omega : (1,q'')^+}
\end{aligned}$$

If we also carry along $\mathsf{U}(\texttt{a.count}, \mathcal{Q}^{EFC}_{a,v})$ (which is in the precondition and is duplicable) then we can show that

$$\mathcal{P}_{\mathsf{keep}}(t) * A \Rightarrow \mathsf{ARC}^{EFC}_{\omega}(a,v) * \mathsf{ARC}^{EFC}_{\omega}(a,v)$$

Since only ghost state is transferred we need not worry about any modalities. This concludes the proof.

This proof shows one good use of allowing elements of the form $(x,0)^+$ for some x in the *EFC* permission structure. If we did not allow such elements where we associate 0 permission amount with a positive number of entities then we would have to give $\boxed{\omega : (1,q)^+}$ to the location invariant and get back $\boxed{\omega : (2,q)^+}$.

6.3.5 Proof of function `drop`

The proof outline for `drop` with respect to location invariant $\mathcal{Q}^{EFC}_{a,v}$ is the same as the proof outline given with respect to the location invariant without the *EFC* permission structure given in Figure 12 except that the adjusted ARC predicate $\mathsf{ARC}^{EFC}_{\omega}(a,v)$ is used. The proof of the fetch-and-add operation is different and we show it now. We have

$$P \equiv \texttt{a.data} \overset{q}{\mapsto} v * \boxed{\omega : (1,q)^+}$$

for some $q \in (0,1]$. We choose

$$\begin{aligned}
\mathcal{P}_{\mathsf{send}}(t) &:= (t = 1\ ?\ \boxed{\omega : (1,q)^+} : P) \\
\mathcal{P}_{\mathsf{keep}}(t) &:= (t = 1\ ?\ \texttt{a.data} \overset{q}{\mapsto} v : \mathsf{emp})
\end{aligned}$$

It holds that $P \Leftrightarrow \mathcal{P}_{\mathsf{send}}(t) * \mathcal{P}_{\mathsf{keep}}(t)$ for all t. We need to prove the CAS triple in the premise of the fetch-and-add rule for every possible value t that could be read.

We can show that $t = 0$ cannot be read by verifying the CAS triple using the CAS$-\bot$ rule analogously to the proof for the `clone` function.

Suppose $t > 1$. We prove the CAS triple using the CAS$-$*basic* rule. We have

$$\begin{aligned}
&\mathcal{P}_{\mathsf{send}}(t) \Leftrightarrow P \\
&\mathcal{P}_{\mathsf{keep}}(t) \Leftrightarrow \mathsf{emp} \\
&\mathcal{Q}^{EFC}_{a,v}(t) \Leftrightarrow \exists f' \in \mathbb{Q} \cap [0,1].\, \texttt{a.data} \overset{f'}{\mapsto} v * \boxed{\omega : (t, 1-f')^-}
\end{aligned}$$

We choose

$$\begin{aligned}
T &:= \mathcal{Q}^{EFC}_{a,v}(t) \\
A &:= \mathsf{emp}
\end{aligned}$$

It holds that

$$\begin{aligned}
\mathcal{P}_{\mathsf{send}}(t) * T &\Rightarrow \exists f' \in \mathbb{Q} \cap [0,1].\, \texttt{a.data} \overset{f'+q}{\mapsto} v * \boxed{\omega : (t-1, 1-(f'+q))^-} \\
&\Rightarrow_{\text{choosing } f := f'+q} \mathcal{Q}^{EFC}_{a,v}(t-1)
\end{aligned}$$

and since the fetch-and-add has access type `rel` this proves the CAS triple. Furthermore, since in this case the thread needs to end up with `emp`, this case is proved.

Next, suppose $t = 1$. In this case we want the thread to end up with the full permission to `a.data`. We prove the CAS triple using the CAS$-$*param* rule. We have

$$\begin{aligned}
&\mathcal{P}_{\mathsf{send}}(1) \Leftrightarrow \boxed{\omega : (1, q)^+} \\
&\mathcal{P}_{\mathsf{keep}}(1) \Leftrightarrow \texttt{a.data} \overset{q}{\mapsto} v \\
&\mathcal{Q}^{EFC}_{a,v}(1) \Leftrightarrow \exists f \in \mathbb{Q} \cap [0,1].\, \texttt{a.data} \overset{f}{\mapsto} v * \boxed{\omega : (1, 1-f)^-}
\end{aligned}$$

Choose

$$\begin{aligned}
\mathcal{T}(z) &:= \boxed{\omega : (1, 1-z)^-} \\
\mathcal{A}(z) &:= \texttt{a.data} \overset{z}{\mapsto} v
\end{aligned}$$

It holds that $\mathcal{Q}^{EFC}_{a,v}(1) \Rightarrow \exists z.\, \mathcal{A}(z) * \mathcal{T}(z)$. Let z be arbitrary. We have that

$$\begin{aligned}
\mathcal{P}_{\mathsf{send}}(1) * T(z) &\Rightarrow \boxed{\omega : (1, q)^+} * \boxed{\omega : (1, 1-z)^-} \\
&\Rightarrow \boxed{\omega : (0, 1-(z+q))^-} \wedge (z + q = 1) \\
&\Rightarrow \boxed{\omega : (0, 0)^-} \wedge (z + q = 1) \\
&\Rightarrow \mathcal{Q}^{EFC}_{a,v}(0) \wedge (z + q = 1)
\end{aligned}$$

where we used that $(a_0, b_0)^- \oplus (a_0, b_1)^+$ is defined iff $b_0 - b_1 = 0$ (i.e. in our case $b_0 = 1 - z$, $b_1 = q$ and $b_0 - b_1 = 1 - (z + q)$). This is essentially the property that if the entity count is 0 then the entity sum must be 0 (as argued in Section 5 the converse does not hold in general).

Since additionally $z + q = 1$ is pure for every z we can instantiate $\varphi(z)$ in the CAS$-$*param* rule with $z + q = 1$. Because only ghost state is given to the location invariant we need not worry about modalities for those resources, but since the access type of the fetch-and-add is `rel` we must take modalities into account when extracting permission to `a.data` from the location invariant. The thread gets

$$\exists z.\, \bigtriangledown \mathcal{A}(z) \wedge (z + q = 1)$$

which implies

$$\bigtriangledown \mathcal{A}(1 - q)$$

hence after the fetch-and-add operation the thread is left with

$$\mathcal{P}_{\mathsf{keep}}(1) * \bigtriangledown \mathcal{A}(1 - q) \Leftrightarrow \texttt{a.data} \overset{q}{\mapsto} v * \bigtriangledown \texttt{a.data} \overset{1-q}{\mapsto} v$$

which after the fence acquire instruction results in the full permission to `a.data`. This concludes the proof of the fetch-and-add in the `drop` function.

6.4 An alternative expression of the *EFC* permission structure

Another way to express location invariants involving the *EFC* permission structure for some ghost location γ would be to introduce global predicates $count(\gamma)$ and $sum(\gamma)$, where $count(\gamma)$ gives the number of entities to γ that have been given away and $sum(\gamma)$ gives the sum of the permission values associated with those entities. One could then introduce proof rules to the logic to be able to directly work with these predicates. Of course one would have to make sure that the global predicates cannot be held by multiple threads

(similar to the source permission) and that they always remain consistent. This would make the source permission $(c, s)^-$ (where $c, s \in \mathbb{Q}_{\geq 0}$) obsolete.

For example, the presented location invariant using the *EFC* permission structure for *ARC* is given by

$$\begin{aligned}\mathcal{Q}^{EFC}_{a,v}(c) :=& \textbf{if } c = 0 \textbf{ then } \boxed{\omega : (0,0)^-} \\ & \textbf{else } \exists f \in \mathbb{Q} \cap [0,1].\, \texttt{a.data} \overset{f}{\mapsto} v * \boxed{\omega : (c, 1-f)^-}\end{aligned}$$

Using the global predicates this could be replaced by

$$\begin{aligned}\mathcal{Q}^{EFC}_{a,v}(c) :=& \textbf{if } c = 0 \textbf{ then } \mathit{count}(\omega) = 0 * \mathit{sum}(\omega) = 0 \\ & \textbf{else } \mathit{count}(\omega) = c * \texttt{a.data} \overset{1-\mathit{sum}(\omega)}{\mapsto} v\end{aligned}$$

One difference is that there is no need for an existentially quantified variable for the permission value associated with the sum of the entities given away, since we can express it using the *sum* predicate. These two location invariants express the same thing. The predicate version explains more directly what is going on. Re-expressing the *EFC* permission structure using the sum and count might enable thinking of an intuitive way for tweaking the permission structure such that the proofs can be made even more direct.

7 Conclusion and future work

In this section we conclude about what was achieved and sketch areas where the work could be continued.

7.1 Conclusion

We provided the first recorded proofs of correctness for simplified versions of the Folly reader-writer spinlock and the Folly barrier examples. In the presented proofs we noticed that the utilized location invariants are not as precise as we would like them to be. As a result, we introduced *generalized location invariants* along with *transition functions* which allow for more expressive location invariants. While this extension solved certain issues, there are still issues that remain in cases where the transition functions may depend on the state of the thread executing the CAS operation.

In [9] a FSL++ proof for the Rust ARC example is provided which uses a known permission structure for the ghost locations. We presented a slightly modified permission structure, the *entity fractional counting permission structure*, which can be used to express a more direct location invariant

for the Rust ARC example. It also enabled the reduction of the number of ghost locations for the other examples.

7.2 Future work

There is no soundness proof for the CAS rule using the generalized location invariants and transition functions presented in Section 4. Finalizing this area is important to be able to use this logic extension reliably in proofs.

As discussed, the transition functions in their current form do not get rid of all of the issues. One reason is that assertions can be framed away in separation logic. To have a more complete solution it seems useful to be able to talk about the complete set of resources held by the thread executing a read-modify-write operation. It would be interesting to investigate if such a notion is even possible without losing many of the advantages, such as modularity, inherent in separation logic. Alternatively, one could try to develop other extensions which deal with the presented issues in a more complete manner.

The *EFC* permission structure provides a nice way to express location invariants for the considered examples. Our proofs seem to justify that supporting this structure in a tool which encodes FSL++, such as Viper [11], would be useful. One challenge with encoding the *EFC* permission structure is automation. If one uses a location invariant with a ghost location governed by the *EFC* permission structure, then for a CAS one needs to figure out which entities must be transferred. It would be nice if a user of the encoding would have to only provide very few additional annotations to make this work.

While the *EFC* permission structure gives advantages, there may be a similar but different permission structure which simplifies the proofs even more. Investigation in this direction might be promising as well.

7.3 Acknowledgements

I would like to thank my supervisor Alexander J. Summers, who helped me develop many of the ideas presented in this report, for all the interesting discussions and for showing me how to view a problem in very elegant ways. Alex, thanks for your invaluable encouragement and guidance. Finally, I would like to thank Prof. Peter Müller for the chance to work on this interesting research project.

References

[1] ISO/IEC 14882:2011: Programming language C++, 2011.

[2] Rust atomic reference counter. `https://doc.rust-lang.org/std/sync/struct.Arc.html`, (accessed October 2, 2017).

[3] Facebook Folly reader-writer spinlock. `https://github.com/facebook/folly/blob/master/folly/RWSpinLock.h`, (accessed September 21, 2017).

[4] Facebook Folly barrier. `https://github.com/facebook/folly/blob/master/folly/futures/Barrier.cpp`, (accessed September 22, 2017).

[5] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2005, pages 259–270, New York, NY, USA, 2005. ACM.

[6] J. Boyland. Checking interference with fractional permissions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS 2003, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag.

[7] M. Doko. Personal communication, 2017.

[8] M. Doko and V. Vafeiadis. A program logic for C11 memory fences. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, pages 413–430, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[9] M. Doko and V. Vafeiadis. Tackling real-life relaxed concurrency with FSL++. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 448–475, 2017.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[11] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract*

Interpretation (VMCAI), volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[12] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL 2001, pages 1–19, London, UK, UK, 2001. Springer-Verlag.

[13] M. Parkinson and G. Bierman. Separation logic and abstraction. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2005, pages 247–258, New York, NY, USA, 2005. ACM.

[14] A. J. Summers and P. Müller. Automating deductive verification for weak-memory programs. Technical Report arXiv:1703.06368, 2017.

[15] V. Vafeiadis. *Program Verification Under Weak Memory Consistency Using Separation Logic*, pages 30–46. Springer International Publishing, Cham, 2017.

[16] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA 2013, pages 867–884, New York, NY, USA, 2013. ACM.

A Partial commutative monoid

Let $(S, \oplus)$ be an algebraic structure, where S is a set and $\oplus$ is a partial binary operation on S. Then $(S, \oplus)$ is a partial commutative monoid iff the following properties hold:

1. For all $x, y, z \in S$ it holds that $(x \oplus y) \oplus z \cong x \oplus (y \oplus z)$ *(associativity)*
2. There exists an element $\eta \in S$ such that for all $x \in S$ it holds that $x \oplus \eta = \eta \oplus x = x$ *(neutral element)*
3. For all $x, y \in S$ it holds that $x \oplus y \cong y \oplus x$ *(commutativity)*

where $t_1 \cong t_2$, for expressions t_1 and t_2, holds iff t_1 and t_2 are defined and have the same value or they are both undefined.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Applying and Extending the Weak-Memory Logic FSL++

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Parthasarathy	Gaurav

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Fislisbach, 19.10.2017	Gaurav Parthasarathy

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.