# Tackling Real-Life Relaxed Concurrency with FSL++

Marko Doko and Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

**Abstract.** We extend fenced separation logic (FSL), a program logic for reasoning about C11 relaxed access and memory fences. Our extensions to FSL allow us to handle concurrent algorithms appearing in practice. New features added to FSL allow for reasoning about concurrent non-atomic reads, atomic updates, ownership transfer via release sequences, and ghost state. As a demonstration of power of the extended FSL, we verify correctness of the atomic reference counter (ARC), a standard library of the Rust programing language, whose implementation relies heavily on advanced features of the C11 memory model. Soundness of FSL and its extensions, as well as the correctness proof of ARC have been established in Coq.

## 1 Introduction

Most formal verification work on multithreaded programs with concurrent accesses to shared memory assumes that programs follow the *sequentially consistent* model of execution [22]. In this model, the executions of a concurrent program consist of all possible interleavings of the actions of its threads.

Even though sequential consistency is a simple and intuitive concurrency model, it does not match the real world. In practice, no hardware provides us with a sequentially consistent execution environment. In order to improve performance or conserve energy, modern hardware implementations give us what is known as *weak memory* models; that is, models of concurrency providing weaker guarantees than sequential consistency. As a result, most of the verification techniques developed for sequential consistency are inapplicable to weak memory models.

In this paper, we will focus on the C11 weak memory model. This software-level model was introduced by the 2011 C and C++ standards [15, 16] as an abstraction over the various different hardware memory models, and provides various low-level primitives for developing efficient concurrent programs. These low-level primitives are slowly gaining adoption not only in C and C++, but are also being incorporated in other programming languages such as Java and Rust.

As the adoption of C11-style weak memory primitives grows, so does the importance of being able to verify correctness of algorithms that use them. Currently, the most successful logic for reasoning about the C11 memory model is GPS [34], which has, for instance, been used to verify an implementation

of the read-copy-update (RCU) algorithm [33], a synchronization mechanism used in the Linux kernel. GPS, however, has an important limitation: namely, it can reason only about the *release-acquire* fragment of the C11 memory model, which leaves programs that use *relaxed* operations (i.e., operations weaker than release-acquire ones) completely out of the reach of GPS. One such algorithm is the *atomic reference counter* (ARC) [1], which we will verify in this paper.

ARC is a part of the standard library of the Rust programming language [2] and provides an interface for concurrent access to a shared data structure. The shared structure can be read by multiple threads, but cannot be modified. ARC ensures that the shared data structure will be deallocated once no reader needs to access the data structure any more. Features present in ARC, which are unsupported by GPS, include *relaxed memory accesses* and *memory fences*.

There is a logic that can deal with both relaxed accesses and memory fences: *fenced separation logic* (FSL) [13]. Unfortunately, even though FSL supports relaxed accesses and memory fences, it lacks some key features which makes it inapplicable beyond simple "toy" examples.

In this work, we extend FSL to make it applicable to real world examples, using ARC as a demonstration of its abilities. Specifically, we extend FSL with three new features:

- partial read *permissions* for non-atomic accesses [8, 10],
- support for *compare-and-swap* (CAS) operations, and
- *ghost state* [12, 18, 23],

all of which are actually needed for proving ARC correct.

Among these three features, the most interesting is ghost state because it interacts with the other FSL features in novel and interesting ways. Ghost state represents supplementary logical resources not used by the program, but only by the user of the logic in order to establish program's correctness.

Ghost state interacts with FSL's ability to transfer ownership of resources between threads. For soundness purposes, transferring a resource from one thread to another cannot happen by simply writing or reading a shared variable; it requires some form of additional synchronization: either a memory fence or a special type of memory access, which essentially incorporates a fence.

A key observation that we made, however, is that ghost state may be soundly transferred between threads under weaker conditions than the other types of resources owned by threads. In particular, it may be transferred by simple non-synchronizing memory accesses! In essence, this is sound because unlike other resources such as $x \mapsto 5$, owning some ghost state does not provide additional power to a thread to perform an action; it only allows us to deduce that certain interference patterns between threads are not possible. As such, the soundness proof can impose slightly weaker conditions that allow two threads to occasionally own the same ghost state resource simultaneously.

At this point, it is worth noting that the soundness proof of FSL assumes a standard strengthening of the C11 model which disables some compiler optimizations (namely, read-write reordering). This strengthening of the C11 model—though standard and partly necessary for performing any kind of formal rea-

```
new(v){
  a = alloc();
  a.data = v;
  a.count_rlx = 1;
  return a;
}

read(a){
  return a.data;
}

drop(a){
  t = fetch_and_add_rel(a.count, -1);
  if(t == 1){
    fence_acq;
    free(a);
  }
}

clone(a){
  fetch_and_add_rlx(a.count, 1);
}
```

**Fig. 1.** Atomic reference counter implementation.

soning about the model—has interesting implications for the soundness of ghost state, which we will discuss in §5.2.

With FSL strengthened in this way, we are able to *formally verify an implementation of ARC* that uses the same pattern of atomic accesses and memory fences as the one that can be found in the standard library of Rust. Both the soundness proof of the new features of FSL and the formal correctness proof of ARC have been fully mechanized in Coq. The complete Coq development, together with our online appendix, is available at `http://plv.mpi-sws.org/fsl/`.

As a rough measure of the effort required to extend the FSL with the features mentioned above, we can look at the size of the Coq development. The size of the soundness proof for FSL is approximately 17.6 KLOC (thousand lines of code), while the soundness proof for FSL++ consists of around 22.7 KLOC representing an increase in size of about 30%. Another 2000 lines were required to complete the verification of ARC, out of which 800 belong to generic auxiliary lemmas, while the remaining 1200 closely follow the correctness proof outlined in §4.

## 2 Atomic Reference Counter

Before going into FSL and its extensions, let us first have a look at the ARC algorithm, as we will use its features to motivate our extensions of FSL.

### 2.1 The Algorithm

Our ARC implementation is given in Fig. 1 and consists of four functions: `new`, `read`, `drop`, and `clone`. To gain a basic understanding of the algorithm, we can ignore the **rel**, **acq**, and **rlx** annotations, as well as any `fence` instructions.

Function `new(v)` creates a new ARC object `a`, sets its `data` field to `v`, and the `count` field to `1`. The `data` field holds the value that can be accessed through the ARC object, and `count` counts the number of references to the ARC object.

Function `read(v)` simply returns the value stored in the ARC object.

Function `clone(a)` operationally just increments the reference counter by one using an atomic *fetch-and-add* instruction. Semantically, `clone` gives us another reference to the ARC object (hence the increment of the counter), which can now

$$
\begin{array}{l}
\{\mathsf{emp}\}\ \texttt{new}(v)\ \{a.\ \mathsf{ARC}(a,v)\} \\
\{\mathsf{ARC}(a,v)\}\ \texttt{read}(a)\ \{y.\ y = v \wedge \mathsf{ARC}(a,v)\} \\
\{\mathsf{ARC}(a,v)\}\texttt{clone}(a)\{\mathsf{ARC}(a,v) * \mathsf{ARC}(a,v)\} \\
\{\mathsf{ARC}(a,v)\}\ \texttt{drop}(a)\ \{\mathsf{emp}\}
\end{array}
$$

**Fig. 2.** ARC specification in separation logic.

also be used to access the value stored in the ARC object. After calling `clone` we can, for example, create a new thread, let it read from one ARC reference, and keep the other reference available for ourselves.

Function `drop(a)` disposes of a reference to the ARC object `a`. If there are still multiple references to the ARC object, `drop` only decreases the reference counter. On the other hand, if the counter gets decremented from one to zero (i.e., there are no more references to the ARC object), `drop` also deallocates the ARC object.

The intended use of the ARC library can be succinctly expressed in terms of separation logic in Fig. 2. In this specification, $\mathsf{ARC}(a, v)$ represents the permission to run functions that access the ARC object $a$. This permission is created by the function `new`, duplicated by `clone`, and destroyed by `drop`.

### 2.2 Why Is ARC Correct?

Let us now consider why ARC is correct. Before attempting to answer this question, we should first ask ourselves, what is the correctness criterion for this algorithm? In other words, what should its specification in Fig. 2 achieve?

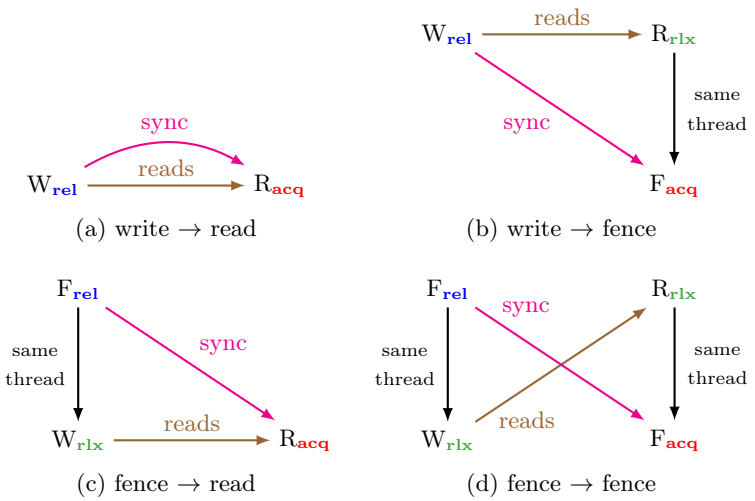
For the algorithm to operate correctly, we are primarily interested in memory safety. We have to ensure that the deallocation does not happen until all the threads are done with reading the value stored in the ARC object. More precisely, the read of the `data` field in the `read` function should not race with the deallocation that happens in the `drop` function.

Additionally, the deallocation should not be attempted twice. For this particular algorithm, it is quite easy to see that is not the case: deallocation happens only once, when the reference counter drops to zero.

In remainder of this section, we therefore focus on the first property.

**Sequential Consinstency** From the perspective of the *interleaving semantics* (a.k.a. *sequential consistency*), the situation is quite clear. Recall that the deallocation happens when `drop` decrements the reference counter to zero. This means that all the ARC objects that have been produced (by either `new` or `clone`) have also been disposed of by `drop`. Obviously, no call to `read` can be made any more, since we no longer have any ARC objects available.

**Weak Memory** When moving to weak memory models, such as C11, the reasoning becomes significantly more complex. In what follows, we are going to give a simplified presentation of the C11 model, focusing on the features used in the ARC algorithm. Complete presentations of the C11 model can be found in [6,35].




(a) write → read (b) write → fence

(c) fence → read (d) fence → fence

**Fig. 3.** Basic release-acquire synchronization.

The C11 model presents executions as graphs where nodes (also called *events*) represent memory accesses. Events (i.e., memory accesses and fences) can be either reads (R), writes (W), updates (U), or fences (F). Reads and writes can be of atomic or non-atomic kind, while updates represent atomic read-modify-write instructions, such as compare-and-swap or fetch-and-add, and can thus be only of atomic kind.

Having a data race on non-atomic accesses is considered to be a programming error, while racing on atomic access is allowed. Atomic accesses provide us with mechanisms to implement synchronization among different threads. How effective an atomic access is in enforcing synchronization depends on its type. Types of atomic accesses are: *relaxed* (**rlx**), which can be applied to any atomic access; *release* (**rel**), for writes and updates; *acquire* (**acq**), for reads and updates; and *acquire-release* (**acq_rel**) for updates only.

For us, the most important question about the C11 model is, *how do we know when one event precedes another in a given execution?*

Put simply, the C11 model specifies that the events in different threads are happening concurrently, and the only way to be sure that two events from different threads are happening in some definite order is to have one of them "see" the other through the process of *synchronization*. In other words, in order to show that an event $a$ happens before another event $b$, we have to be able to start at $a$, and eventually reach $b$ by following thread execution "downstream", and the only time we are allowed to move from one thread to another is at the synchronization points.

Some simple ways to achieve synchronization are depicted in Fig. 3. Synchronization always connects a *release event* (event of a **rel** or **acq_rel** type) with an *acquire event* (event of an **acq** or **acq_rel** type), and always happens

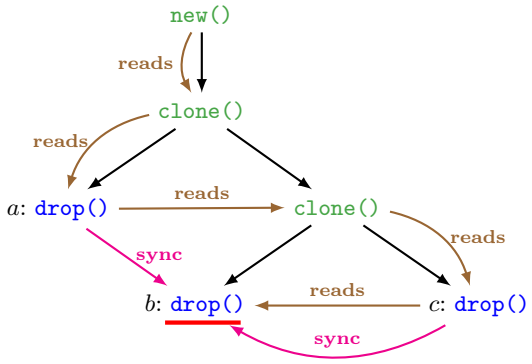



**Fig. 4.** An example execution of the ARC algorithm.

as consequence of a read. In Fig. 3a we see the simplest case of synchronization, which happens immediately when an acquire read reads from a release write. In the other three situations in Fig. 3, relaxed accesses are helped along by fences (which can be of a **rel**, **acq**, or **acq_rel** kind) in order to achieve synchronization. Note that in these three cases, synchronization does not occur immediately as the read happens, but is delayed until all the required fences come into play.

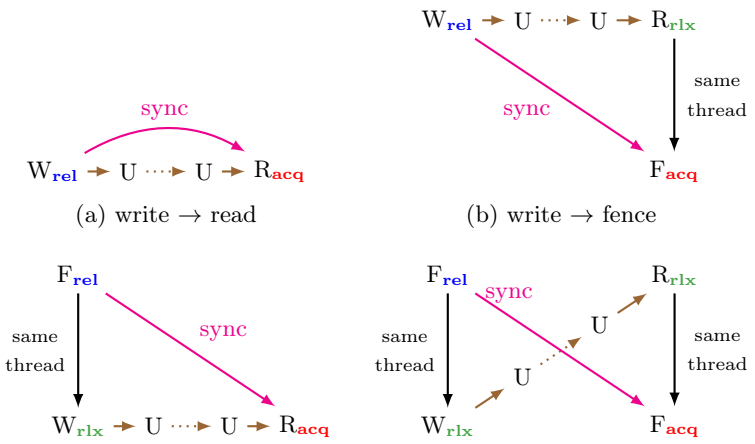
Looking back at the ARC algorithm in Fig. 1 we can see that it uses relaxed accesses in the `new` and `clone` functions, while the function `drop` features a release access and an acquire fence. Instead of being regular reads or writes, `fetch_and_add` instructions are *atomic update* events, which act as both reads and writes. A release update (such as the one inside `drop`) acts as a release write and a relaxed read, while relaxed updates are relaxed as both reads and writes.

In order to get an intuitive understanding of the synchronization strategy employed by the ARC algorithm, we will have a look at the example execution presented in Fig. 4. The underlined `drop` function is the one that does the deallocation. To ensure absence of data races, all other drop functions should synchronize with `drop`. This suffices to ensure the absence of races, because we know by the intended use of the ARC library that every `read` will be followed by some `drop`.

One of these synchronizations happens according to Fig. 3b, as the `drop` at node $b$ reads from the `drop` at node $c$. For the other synchronization between nodes $a$ and $b$, however, the mechanisms presented in Fig. 3 are just not enough.

The problem we are facing with achieving the other synchronization is that so far presented synchronization mechanisms allow an acquire construct to synchronize only with one other thread. What we need is some mechanism that will allow the single acquire fence in the whole ARC algorithm to synchronize with multiple release writes.

In order to synchronize all threads before deallocation, ARC exploits a more advanced synchronization technique provided by C11 called *release sequences*. Simply stated, to trigger synchronization between two threads it is not necessary for one to read directly from the other (as in Fig. 3), but there can be a reading chain (through atomic updates) from one thread to the other.




(a) write → read (b) write → fence

(c) fence → read (d) fence → fence

**Fig. 5.** Synchronization through release sequences.

Figure 5 depicts the four generalized versions of the cases in Fig. 3. We can now see that the synchronization mechanism shown in Fig. 5b explains the problematic synchronization from $a$ to $b$ in Fig. 4.

## 3 Extending FSL

In this section, we will first take an overview look at the existing features of FSL, after which we are going to turn our attention to the three extensions necessary for applying FSL to realistic examples such as ARC.

### 3.1 FSL Basics

Like its precursor, RSL [35], FSL divides memory locations into two categories: *atomic* and *non-atomic*.

Non-atomic locations are the ones that are used for "regular" accesses (i.e., we use non-atomic accesses whenever we are not implementing a synchronization mechanism). FSL ensures that there will be no data races on non-atomic accesses. For reasoning about non-atomic accesses, FSL provides the standard separation logic rules [26, 29].

Atomic accesses are the more interesting ones. As we have already seen in §2.2, atomic accesses come in four modes (**acq_rel**, **rel**, **acq**, and **rlx**), and are used to create synchronization between threads. In the rest of this subsection, we will focus our attention on FSL rules regarding atomic accesses.

From the perspective of FSL, atomic accesses are used to transfer ownership between threads. Threads can give up ownership of certain resources by writing to an atomic location, after which another thread can pick up that resource by

reading from the same location. Resources are transferred through write-read pairs, and the rules of the logic make sure that the transferred resources are not used until the threads in question synchronize.

In what follows, for the sake of clarity, we are going to present slightly simplified FSL rules. A complete presentation of FSL can be found in [13].

**FSL triples** FSL triples are of the form $\{P\}\, E\, \{v.\, Q\}$, where $P$ and $Q$ are assertions denoting the precondition and the postcondition of the expression $E$. In the postcondition, the variable $v$ binds the return value of $E$. In cases where the postcondition does not depend on the return value, the $v$ binder may be omitted.

**Release Writes** The easiest way to transfer away a resource is to do a release write. Since the release write is both the point of origin of ownership transfer, as well as the point of origin of synchronization (see Figs. 3a and 3b), we can simply transfer the resource we want without any further complications. This is summarized in the following rule.

$$\{\mathsf{Rel}(\ell, \mathcal{Q}) * \mathcal{Q}(v)\}\, [\ell]_{\mathbf{rel}} := v\, \{\mathsf{Rel}(\ell, \mathcal{Q})\} \qquad \text{(W-REL)}$$

In the precondition, the assertion $\mathsf{Rel}(\ell, \mathcal{Q})$ grants us permission to write to the atomic location $\ell$. $\mathcal{Q}$ is a mapping from values to assertions, specifying which resource we have to give up when writing which value. In particular, if we want to store the value $v$ into $\ell$, we have to give up the ownership of the resource $\mathcal{Q}(v)$. As we can see from the postcondition, once the write is done, we no longer have the access to the resource $\mathcal{Q}(v)$, which can now be obtained by readers.

**Relaxed Writes** Resources can also be sent away by doing a relaxed write, but only if the write is helped along by a release fence, as in Figs. 3c and 3d. Our ownership transfer strategy is somewhat more involved in this case. By doing a relaxed write, we can only transfer resources that have been "prepared" before the release fence took effect. In other words, the resources sent away by the relaxed write should not be accessed in between the fence and the write. The following two rules describe this situation.

$$\{P\}\, \texttt{fence}_{\mathbf{rel}}\, \{\triangle P\} \qquad \text{(F-REL)}$$

$$\{\mathsf{Rel}(\ell, \mathcal{Q}) * \triangle \mathcal{Q}(v)\}\, [\ell]_{\mathbf{rlx}} := v\, \{\mathsf{Rel}(\ell, \mathcal{Q})\} \qquad \text{(W-RLX)}$$

When executing a release fence, we can put any resource under the $\triangle$ modality. The assertion $\triangle P$ says, "$P$ has been made ready for transfer and it may not be accessed any more." The (W-RLX) rule differs from the (W-REL) rule only in the appearance of $\triangle$ in the precondition. Essentially, we execute a relaxed write the same way we do a release write, with one important difference: a resource transferred away by the relaxed write has to be under the $\triangle$ modality, ensuring that a release fence has been placed before the write.

**Acquire Reads** Acquire reads function as end points of both resource transfer and synchronization (see Figs. 3a and 3c). For this reason, resource acquisition by acquire reads is quite simple.

$$\{\mathsf{Acq}(\ell, \mathcal{Q})\}\ [\ell]_{\mathbf{acq}}\ \{v.\ \mathcal{Q}(v)\} \tag{R-ACQ}$$

The assertion $\mathsf{Acq}(\ell, \mathcal{Q})$ allows a thread to perform the acquire read. Again, $\mathcal{Q}$ is a mapping from values to assertions. From the perspective of a read, this mapping tells us which resource will be acquired when reading which value. In particular, if the value read is $v$, then the resource acquired is $\mathcal{Q}(v)$.

**Relaxed Reads** When acquiring ownership via relaxed read, we have to wait for a subsequent acquire fence to synchronize with the thread we are reading from (see Figs. 3b and 3d). Only after synchronization are we allowed to use the acquired resource. The following two rules represent this case.

$$\{\mathsf{Acq}(\ell, \mathcal{Q})\}\ [\ell]_{\mathbf{rlx}}\ \{v.\ \triangledown\mathcal{Q}(v)\} \tag{R-RLX}$$

$$\{\triangledown P\}\ \mathtt{fence}_{\mathbf{acq}}\ \{P\} \tag{F-ACQ}$$

The resource acquired in the (R-ACQ) rule is placed under the $\triangledown$ modality. The assertion $\triangledown P$ simply means "$P$ cannot be used before an acquire fence has been reached." The (F-ACQ) rule tells us that the acquire fence makes resources hidden behind the $\triangledown$ modality usable.

**Allocation of Atomics** The $\mathsf{Rel}$ and $\mathsf{Acq}$ permissions are generated when a new atomic variable is allocated. At the point of allocation, we can freely choose the mapping $\mathcal{Q}$ which governs the ownership transfer through the newly allocated variable.

$$\frac{\mathcal{Q}\colon \mathit{Values} \to \mathit{Assertions}}{\{\mathsf{emp}\}\ \mathbf{alloc}()\ \{\ell.\ \mathsf{Rel}(\ell, \mathcal{Q}) * \mathsf{Acq}(\ell, \mathcal{Q})\}} \tag{A-AT}$$

These are all the rules regarding ownership transfer through atomic accesses in FSL. Let us now turn our attention to the three extensions which will allow us to verify ARC.

### 3.2 Partial Permissions for Non-atomics

Basic FSL does not support reasoning about programs with concurrent read accesses to non-atomic locations. On the other hand, ARC is a library specifically used to allow concurrent reads of a shared resource. Therefore, this is the first gap that needs to be bridged in order to successfully verify programs like ARC.

To enable reasoning about concurrent non-atomic reads, we outfitted FSL with partial permissions [8, 10] for non-atomic locations. In order to execute a write, the full permission is needed, while reading is possible with a partial permission. The rules of the logic make sure that the full permission cannot

concurrently coexists with a partial one, nor can there exists more than one full permission at a time. As a result, there cannot be any read-write or write-write races on non-atomic locations.

Formally, permission structures are tuples $(M, \oplus, \varepsilon, \mathbb{1})$, where $(M, \oplus)$ forms a partial commutative monoid with $\varepsilon$ as the neutral element, and $\mathbb{1} \in M \setminus \{\varepsilon\}$ is a 'maximal' element of the monoid composable only with the neutral element, i.e., $\mathbb{1} \oplus q$ is undefined for every $q \in M \setminus \{\varepsilon\}$.

To write to a location $\ell$, one must have the full permission $\ell \overset{\mathbb{1}}{\mapsto} -$; while to read from $\ell$, having a permission $\ell \overset{q}{\mapsto} v$ for any $q \in M \setminus \{\varepsilon\}$ suffices. Assertion $\ell \overset{\varepsilon}{\mapsto} -$ is taken to be equivalent with the empty resource $\mathsf{emp}$. Separating conjunction respects the composition operation on the monoid:

$$\ell \overset{p}{\mapsto} v * \ell \overset{q}{\mapsto} v \iff \begin{cases} \ell \overset{p \oplus q}{\mapsto} v & \text{if } p \oplus q \text{ is defined} \\ \mathsf{false} & \text{otherwise.} \end{cases}$$

The most well known permission model, which is incidentally also the one used in the correctness proof of ARC, is the model of *fractional permissions* [10]. In this model, permissions are fractions in the interval $[0, 1]$, $\varepsilon = 0$, $\mathbb{1} = 1$, and composition is defined by

$$p \oplus q = \begin{cases} p + q & \text{if } p + q \in [0, 1] \\ \textbf{undefined} & \text{otherwise.} \end{cases}$$

Our proof of soundness is not dependent on fractional permissions, but is parametric in the permission structure for non-atomic accesses, which allows for greater flexibility when designing proofs that require partial permissions.

### 3.3 Compare-and-swap Rules

Another problem we are facing when verifying ARC is the presence of atomic update operations (`fetch_and_add` instructions), for which no support is provided in FSL. We provide the rules for compare-and-swap (CAS), a basic atomic update instruction, which can be used to implement other, more advanced ones, such as `fetch_and_add`.

Details of the implementation of `fetch_and_add` using CAS, and the corresponding FSL specification for `fetch_and_add` can be found in §4.3.

The CAS instruction $\mathsf{CAS}_{\boldsymbol{\tau}}(\ell, v, v')$ reads the location $\ell$, and if the value read is $v$ it updates it atomically to $v'$. If CAS reads some value other than $v$, then the update is not executed. In any case, CAS returns the value read. Parameter $\boldsymbol{\tau}$ tells us the type of update event generated by the successful CAS operation. The possible values of $\boldsymbol{\tau}$ are **rlx**, **rel**, **acq**, and **acq_rel**.

Recall that update actions act as both reads and writes. When reading, the update is treated as an acquire read action if it is of **acq** or **acq_rel** kind, and as a relaxed read otherwise. Acting as a writer, the update is treated as a release write if it is of **rel** or **acq_rel** kind, and as a relaxed write otherwise.

FSL [13] provides no CAS rules, but its predecessor RSL [35] does. The CAS rule provided by RSL only supports ownership transfer by **acq_rel** CASes, and does not allow any ownership transfer over release sequences. Ownership transfer using release sequences and multiple types of CASes is necessary to verify complex algorithms such as ARC. Therefore, it is necessary to augment FSL with stronger CAS rules than the one present in RSL.

In what follows, we will present the new rules regarding CAS instructions. Here, as in §3.1, we are presenting a simplified version of the rules. For full rules, we refer the reader to the appendix.

We will start the presentation of the CAS rules with a simplified version of the rule for the strongest type of CAS instruction, the **acq_rel** CAS.

$$\frac{\begin{array}{c}\mathcal{Q}(v) \;\Rightarrow A * T \\ P * T \Rightarrow \mathcal{Q}(v')\end{array}}{\left\{\mathsf{U}(\ell,\mathcal{Q}) * P\right\}\, \mathsf{CAS}_{\mathbf{acq\_rel}}(\ell, v, v') \left\{\begin{array}{l} a.\ (a = v \wedge A) \\ \vee\ (a \neq v \wedge \mathsf{U}(\ell,\mathcal{Q}) * P)\end{array}\right\}} \quad (\textsc{cas-ar}^*)$$

In the precondition we have assertion $\mathsf{U}(\ell, \mathcal{Q})$, which gives us the permission to execute CAS on the location $\ell$. As in $\mathsf{Rel}$ and $\mathsf{Acq}$ assertions, $\mathcal{Q}$ is a mapping from values to assertions, telling us what resource we can get by reading a value, and which resource we have to send away when writing a value. The remaining component in the precondition is $P$, the resource we want to transfer away upon a successful CAS operation.

If the CAS fails (i.e., the value read, $a$, is different from $v$), then no resource transfer happens, and in the postcondition we are left with the same resources we had in the precondition.

In the case of a successful CAS (i.e., the value read was $v$), we have at our disposal the resource $\mathcal{Q}(v)$. According to the first premise of the rule, we have to split $\mathcal{Q}(v)$ into two parts, $A$, and $T$. Resource $A$ is the part that we are going to acquire and keep it for ourselves in the postcondition. Resource $T$ will remain in the invariant $\mathcal{Q}$. The second premise requires that the resource $P$ (which we have in our precondition) together with the resource $T$ (which we left behind when acquiring ownership) are enough to satisfy $\mathcal{Q}(v')$, thus reestablishing the invariant for the newly written value.

The ($\textsc{cas-ar}^*$) is a useful rule as it stands, but can still be strengthened. The opportunity for strengthening lies in the second premise of the ($\textsc{cas-ar}^*$) rule. If, in addition to merely reestablishing the invariant, we manage to prove some additional facts, we can carry those facts into the postcondition. The strengthened rule is

$$\frac{\begin{array}{c}\mathcal{Q}(v) \Rightarrow \exists z.\ \mathcal{A}(z) * \mathcal{T}(z) \\ \forall z.\ (P * \mathcal{T}(z) \Rightarrow \mathcal{Q}(v') \wedge \varphi(z)) \\ \forall z.\ \mathsf{pure}(\varphi(z))\end{array}}{\left\{\mathsf{U}(\ell,\mathcal{Q}) * P\right\}\, \mathsf{CAS}_{\mathbf{acq\_rel}}(\ell, v, v') \left\{\begin{array}{l} a.\ (a = v \wedge \exists z.\, \mathcal{A}(z) \wedge \varphi(z)) \\ \vee\ (a \neq v \wedge \mathsf{U}(\ell,\mathcal{Q}) * P)\end{array}\right\}}. \quad (\textsc{cas-ar})$$

Instead of assertions $A$ and $T$, the rule now features mappings $\mathcal{A}$ and $\mathcal{T}$ from values to assertions. The first premise asks us to split $\mathcal{Q}(v)$ into $\mathcal{A}(z)$ and

$\mathcal{T}(z)$, for some value $z$. The second premise requires that from $P * \mathcal{T}(z)$ we prove not only $\mathcal{Q}(v')$, but also some fact about $z$, which then gets carried over to the postcondition. Lastly, it is required for $\varphi(z)$ to be *pure*, meaning that the assertion $\varphi(z)$ is a logical fact about $z$, and is not saying anything about the ownership of resources or the state of the heap.

Rules for the other types of CAS accesses are all a slight modification of the (CAS-AR) rule. Modifications are in the same vein as the ones that get us from (R-ACQ) and (W-REL) to (R-RLX) and (W-RLX). Namely, where the access type gets relaxed, $\triangle$ and $\triangledown$ modalities take over in order to ensure that proper fences have been placed.

Since the premises in (CAS-REL), (CAS-ACQ), and (CAS-RLX) are the same as in (CAS-AR), we will avoid repeating them.

$$\{\mathsf{U}(\ell, \mathcal{Q}) * P\}\mathsf{CAS}_{\mathbf{rel}}(\ell, v, v') \begin{Bmatrix} a.\ (a = v \land \exists z.\ \triangledown\mathcal{A}(z) \land \varphi(z)) \\ \lor\ (a \neq v \land \mathsf{U}(\ell, \mathcal{Q}) * P) \end{Bmatrix} \quad \text{(CAS-REL)}$$

$$\{\mathsf{U}(\ell, \mathcal{Q}) * \triangle P\}\mathsf{CAS}_{\mathbf{acq}}(\ell, v, v') \begin{Bmatrix} a.\ (a = v \land \exists z.\ \mathcal{A}(z) \land \varphi(z)) \\ \lor\ (a \neq v \land \mathsf{U}(\ell, \mathcal{Q}) * \triangle P) \end{Bmatrix} \quad \text{(CAS-ACQ)}$$

$$\{\mathsf{U}(\ell, \mathcal{Q}) * \triangle P\}\mathsf{CAS}_{\mathbf{rlx}}(\ell, v, v') \begin{Bmatrix} a.\ (a = v \land \exists z.\ \triangledown\mathcal{A}(z) \land \varphi(z)) \\ \lor\ (a \neq v \land \mathsf{U}(\ell, \mathcal{Q}) * \triangle P) \end{Bmatrix} \quad \text{(CAS-RLX)}$$

Release CAS is treated as a release write and a relaxed read. Therefore, in (CAS-REL) we can send away $P$ without any problems, but the acquired resource has to be placed under the $\triangledown$ modality, requiring us to use an acquire fence before accessing the resource.

Acquire CAS is a relaxed write and an acquire read. Because of this, in (CAS-ACQ) the resource we are trying to transfer away is under the $\triangle$ modality, requiring a release fence before the CAS. On the other hand, the resource we acquire is immediately usable.

Relaxed CAS is relaxed as both read and write. This is reflected in the (CAS-RLX) rule by having both modalities in play.

Note that simple CAS rules in the style of (CAS-AR*) can be derived from the more general ones for any type of CAS. We simply need to choose $\mathcal{A}$ and $\mathcal{T}$ such that they do not depend on $z$, and set $\varphi(z)$ to always be $\mathsf{true}$.

*Remark 1 (About the CAS rule strengthening).* The strengthening was motivated by the ARC proof. The ARC algorithm can be proven correct using just the simple CAS rules that do not contain the "$z$ parametrization". The proof using the simple CAS rules requires the use of additional ghost state (see §3.4), and is in general more complicated compared to the proof presented in §4.

*Remark 2 (About the soundness of the CAS rules).* The soundness of FSL++'s CAS rules (even the simple ones) depends heavily on release sequences (Fig. 5). Specifically, the rules allow us to split the invariant of the value read $\mathcal{Q}(v)$ into two parts and take out only the $\mathcal{A}(z)$ part, while using the $\mathcal{T}(z)$ part to reestablish the invariant for the new value written. In essence, the $\mathcal{T}(z)$ part of $\mathcal{Q}(v)$ is being sent down the chain of updates reading from each other, and can be picked up at any later point.

It is interesting to note that as long as we are working within the release-acquire fragment of the C11 model (i.e., all writes are of **rel** type, all reads are of **acq** type, and all updates are of **acq_rel** type), the soundness of the split does not depend on release sequences, because every act of reading causes synchronization to happen.

On the other hand, in the presence of the relaxed accesses, release sequences are required to establish the soundness of the split even for the (CAS-AR) rule.

*Remark 3 (Soundness of the RSL-style CAS rule).* A variant of the RSL's CAS rule is admissible in FSL++. The difference is that we would now require the release permission to be present in the precondition, unlike in RSL, where it could be a part of the acquired resource. This is not an important restriction, because (due to the duplicability of release permissions) any RSL proof that uses the CAS rule can be modified to include the release permission in the precondition.

The last CAS rule (CAS-⊥) allows us to quickly conclude that a successful CAS cannot happen in the situation where we own a resource which is incompatible with the resources which would be acquired by a successful CAS operation.

$$\frac{\mathcal{Q}(v) * P \Rightarrow \mathsf{false} \qquad \boldsymbol{\tau} \in \{\mathbf{rlx}, \mathbf{rel}, \mathbf{acq}, \mathbf{acq\_rel}\}}{\left\{\mathsf{U}(\ell, \mathcal{Q}) * P\right\} \mathsf{CAS}_{\boldsymbol{\tau}}(\ell, v, v') \left\{\, a.\, a \neq v \wedge \mathsf{U}(\ell, \mathcal{Q}) * P \,\right\}} \quad \text{(CAS-}\bot\text{)}$$

The U permission is obtained upon allocation in a similar fashion as the Rel and Acq permissions.

$$\frac{\mathcal{Q}\colon \mathit{Values} \to \mathit{Assertions}}{\left\{\mathsf{emp}\right\} \mathbf{alloc}() \left\{\ell.\ \mathsf{U}(\ell, \mathcal{Q})\right\}} \quad \text{(A-AT-U)}$$

Finally, we would like to bring your attention to several useful properties of the update permission U. It is duplicable, and it interacts with the Rel and Acq permissions, allowing us to perform not only updates, but also reads and writes, when holding an update permission.

$$\mathsf{U}(\ell, \mathcal{Q}) \iff \mathsf{U}(\ell, \mathcal{Q}) * \mathsf{U}(\ell, \mathcal{Q}) \quad \text{(U-SPLIT)}$$

$$\mathsf{U}(\ell, \mathcal{Q}) \iff \mathsf{U}(\ell, \mathcal{Q}) * \mathsf{Rel}(\ell, \mathcal{Q}) \quad \text{(U-REL-SPLIT)}$$

$$\mathsf{U}(\ell, \mathcal{Q}) \iff \mathsf{U}(\ell, \mathcal{Q}) * \mathsf{Acq}(\ell, \lambda v.\mathsf{emp}) \quad \text{(U-ACQ-SPLIT)}$$

According to (U-REL-SPLIT), when holding the $\mathsf{U}(\ell, \mathcal{Q})$, we also have $\mathsf{Rel}(\ell, \mathcal{Q})$, allowing us to write to $\ell$ using the appropriate atomic write rule. On the other hand, (U-ACQ-SPLIT) tells us that we are allowed to read when holding the $\mathsf{U}(\ell, \mathcal{Q})$ permission, but we cannot gain any ownership (more precisely, no matter the value read, the acquired resource will always be the empty resource emp).

### 3.4 Ghost State

Even though we are now able to reason about both concurrent non-atomic reads, and atomic update operations, we still do not have sufficient reasoning power to verify the correctness of ARC.

To see what are we lacking, we will turn our attention to the `clone` function (see Fig. 1). Our desired specification from Fig. 2 tells us that starting with one $\mathsf{ARC}(a, v)$ resource, after executing `clone(`$a$`)`, we will have that permission duplicated.

The only thing `clone` does is to increment the reference counter by one. The obvious way to get the additional ARC permission would be to acquire it from the invariant governing the reference counter, via the (CAS-RLX) rule. Unfortunately, any resource acquired that way would be protected by the $\triangledown$ modality, and there is no acquire fence to make the resource usable. In short, `clone` function cannot acquire any ownership, since it does not synchronize with any other thread.

So, if we cannot acquire any ownership when executing `clone`, what can we do? One possibility is to somehow duplicate the $\mathsf{ARC}(a, v)$ permission we already have. This would not require us to acquire any ownership, but it also makes the act of incrementing the counter superfluous. If we can simply duplicate the $\mathsf{ARC}(a, v)$ permission, what is the point in having the `clone` function at all?

If we want to verify ARC, we have to be able to remember the fact that `clone` produced another instance of the $\mathsf{ARC}(a, v)$ resource (i.e., the reference counter was incremented), without the `clone` function acquiring any additional resources. To achieve this reasoning, we employ *ghost state* [12, 18, 23, 34], a very useful feature of program logics that is often used for logical "accounting" without changing the program state.

The way to think of the ghost state is as if we have at our disposal locations that are never accessed by our program. Those locations carry *ghost resources*, which cannot influence the behavior of the program, since they are never accessed by the program, but can help us in reasoning.

In a proof, ghosts can be simply introduced whenever the need for them arises using the (GHOST-INTRO) rule.

$$\frac{\{P\}\, C\, \{Q\}}{\{P\}\, C\, \left\{Q * \exists \gamma.\, \boxed{\gamma : g}\right\}} \qquad \text{(GHOST-INTRO)}$$

The assertion $\boxed{\gamma : g}$ means that the ghost location $\gamma$ carries the ghost resource $g$. Ghost resources (on a single location) have to form a *partial commutative monoid* (PCM). The composition operation ($\oplus$) of the PCM connects the ghost resources to the separating conjunction of FSL.

$$\boxed{\gamma : g} * \boxed{\gamma : g'} \iff \begin{cases} \boxed{\gamma : g \oplus g'} & \text{if } g \oplus g' \text{ is defined,} \\ \mathsf{false} & \text{otherwise.} \end{cases} \qquad \text{(GHOST-}*\text{)}$$

The most important feature of ghost state from the perspective of the verification of ARC is ability to transfer ownership of ghosts without the need for synchronization. This is achieved by having the ghost state be agnostic with respect to the $\triangle$ and $\triangledown$ modalities.

$$\boxed{\gamma : \varepsilon} \iff \triangle \boxed{\gamma : \varepsilon} \iff \triangledown \boxed{\gamma : \varepsilon} \qquad \text{(GHOST-MOD)}$$

Intuitively, it is not a problem to define the ghost state in such a way to have the (GHOST-MOD) equivalences hold, because the ghost state is not accessed by the program. The principal duty of the $\triangle$ and $\triangledown$ modalities is to ensure proper placement of fences in order to avoid any data races on non-atomic accesses. Since the ghost state is never accessed, it cannot be involved in any data races, and is therefore free to ignore modalities.

## 4 Verification of ARC

In this section, we will use FSL to verify the ARC algorithm from Fig. 1. Since FSL does not have support for deallocation, we treat the call to the `free` function as a no-operation. For further discussion about handling deallocation see §5.3.

The following theorem contains the formal correctness statement for ARC.

**Theorem 1 (Correctness of ARC).** *There exists a predicate* $\mathsf{ARC}_{\gamma,\delta}$*, parametrized by two ghost locations* $\gamma$ *and* $\delta$*, such that the following holds*

$$\begin{array}{rcl}
\{\mathsf{emp}\} & \texttt{new}(v) & \{a.\, \exists \gamma, \delta.\, \mathsf{ARC}_{\gamma,\delta}(a, v)\} \\
\{\mathsf{ARC}_{\gamma,\delta}(a, v)\} & \texttt{read}(a) & \{y.\, y = v \land \mathsf{ARC}_{\gamma,\delta}(a, v)\} \\
\{\mathsf{ARC}_{\gamma,\delta}(a, v)\} & \texttt{clone}(a) & \{y.\, y \neq 0 \land \mathsf{ARC}_{\gamma,\delta}(a, v) * \mathsf{ARC}_{\gamma,\delta}(a, v)\} \\
\{\mathsf{ARC}_{\gamma,\delta}(a, v)\} & \texttt{drop}(a) & \left\{y.\, (y > 1 \land \mathsf{emp}) \lor (y = 1 \land a.\texttt{data} \overset{1}{\mapsto} v)\right\},
\end{array}$$

*where the fractional permission structure is used for the non-atomic locations.*

The return value of the `clone` and `drop` functions is considered to be the value returned by the `fetch_and_add` instruction within those functions. (Function `fetch_and_add` returns the value before the increment.) In other words, return value $y$ for `clone` means that it incremented the reference counter from $y$ to $y+1$, and for `drop` it means that the counter was decremented from $y$ to $y-1$.

Note that the specification of `drop` tells us that in the case where the reference counter was decremented from 1 to 0, we have the full permission on $a.\texttt{data}$. When modeling deallocation, having the full permission for a location would be enough to deallocate it.

An additional thing of note is that we prove that the return value of the `clone` and `drop` functions can never be 0. This means that `clone` and `drop` never try to access the ARC object after all the references to it have been dropped.

The rest of this section is devoted to the proof of Theorem 1.

The theorem already states the permission model used for non-atomic locations. We are left with choosing a PCM for the ghost state. Our chosen structure is described in the following lemma.

**Lemma 1 (Ghost Monoid).** *The structure* $(\mathbb{Q}_{\geqslant 0} \times \{+,-\}, \oplus)$*, with the partial binary operation* $\oplus$ *defined as*

$$
\begin{aligned}
f^+ \oplus q^+ &:= (f+q)^+ \\
f^- \oplus q^- &:= \textbf{undefined} \\
f^+ \oplus q^- := q^- \oplus f^+ &:= \begin{cases} (q-f)^- & \textit{if } q - f \geqslant 0 \\ \textbf{undefined} & \text{otherwise} \end{cases}
\end{aligned}
$$

*is a partial commutative monoid, with the neutral element* $0^+$*.*

Think of a "positive" ghost assertion $\boxed{\gamma : q^+}$ as having a $q$ amount of some resource, while the "negative" ghost assertion $\boxed{\gamma : q^-}$ counts how much of that resource exists at any given time.

It is important to note that there can exist only one negative ghost assertion at a single point in time, since (according to (GHOST-$*$)) having more than one would lead to a contradiction.

We can now define the invariant that will govern updates to ARC's reference counting field.

**Definition 1 (ARC invariant).** *For location* $x$*, value* $v$*, and ghost locations* $\gamma$ *and* $\delta$*, we define the mapping from values to assertions*

$$
\begin{aligned}
\mathcal{Q}_{\gamma,\delta,v,x} \stackrel{\text{def}}{=} \lambda c.\, &\textbf{if } c = 0 \textbf{ then } \boxed{\gamma : 0^-} * \boxed{\delta : 0^-} \\
&\textbf{else } \exists f \in [0,1].\, x \overset{f}{\mapsto} v * \boxed{\gamma : (c-1+f)^-} * \boxed{\delta : (1-f)^-}.
\end{aligned}
$$

The way to think about the invariant is "if the value of the resource counter is $c$, then $\mathcal{Q}_{\gamma,\delta,v,x}(c)$ holds." There are two main parts to the $\mathcal{Q}_{\gamma,\delta,v,x}$ invariant.

1. Permissions to access the location $x$ that have been dropped by various threads are collected into the assertion $x \overset{f}{\mapsto} v$.
2. The assertion $\boxed{\gamma : (c-1+f)^-}$ counts the number of still active ARC objects created by the `clone` function (this number is $c-1$), while at the same taking note of the amount of read permissions to $x$ that have been dropped so far (this is represented by $f$).

The interplay between these two parts is what will enable us to reconstitute the full permission after all the ARC objects have been dropped. How this happens will become clear in §4.5.

Lastly, the least complicated part of the invariant, the ghost state attached to the ghost location $\delta$, counts how much of the access permission to $x$ is shared by the still active ARC objects. This will be used in §4.4 and §4.5 in order to establish that `clone` and `drop` never read 0 as the value of the reference counter.

We are now finally at the point where we can define the ARC predicate.

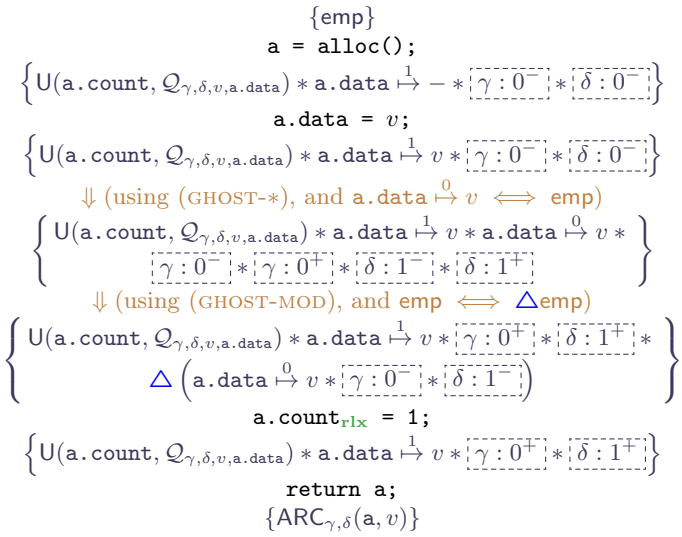



**Fig. 6.** Function `new`: proof sketch.

**Definition 2 (ARC Predicate).** *For ghost locations $\gamma$ and $\delta$, we define*

$$\mathsf{ARC}_{\gamma,\delta}(a, v) \stackrel{\text{def}}{=} \mathsf{U}(a.\texttt{count}, \mathcal{Q}_{\gamma,\delta,v,a.\texttt{data}}) * \exists q \in \langle 0, 1]. a.\texttt{data} \overset{q}{\mapsto} v * \boxed{\gamma : (1-q)^+} * \boxed{\delta : q^+}.$$

The ARC predicate consists of four parts.

1. A permission to execute atomic updates on $a.\texttt{count}$, as long as we respect the $\mathcal{Q}_{\gamma,\delta,v,a.\texttt{data}}$ invariant.
2. Some fraction of the access permission to $a.\texttt{data}$, allowing us to read from it.
3. A ghost $\boxed{\gamma : (1-q)^+}$, designed to help the ARC invariant in keeping track of the number of outstanding ARC objects, and the amount of read permissions to `a.data` shared among them.
4. A ghost $\boxed{\delta : q^+}$, designed to make the $\mathsf{ARC}_{\gamma,\delta}(a, v)$ assertion incompatible with the $\mathcal{Q}_{\gamma,\delta,v,a.\texttt{data}}(0)$ assertion $(q > 0 \wedge \boxed{\delta : q^+} * \boxed{\delta : 0^-} \Rightarrow \mathsf{false})$, therefore making sure we cannot read 0 from $a.\texttt{count}$.

In what follows, we are going to discuss main points of the proof for each of the functions from the ARC algorithm. Full formal proofs are available in the Coq formalization.

### 4.1 Function `new`

In Fig. 6 you can see a simplified version of the proof for the function `new`.

At the beginning, we have to introduce two ghosts ($\gamma$ and $\delta$) using the (GHOST-INTRO) rule, as well as allocate a non-atomic location `a.data`, and an atomic location `a.count`. We are allocating `a.count` using the (A-AT-U) rule. Naturally, we will choose the mapping defined in Definition 1 as the invariant governing the `a.count` location.

The most interesting part of the proof happens when we are executing the relaxed write instruction $\texttt{a.count}_{\texttt{rlx}}\ \texttt{= 1}$. The resources we own as we are about to execute the relaxed write are

$$\mathsf{U}(\texttt{a.count}, \mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}) * \texttt{a.data} \overset{1}{\mapsto} v * \boxed{\gamma : 0^-} * \boxed{\delta : 0^-},$$

and according to (U-REL-SPLIT) and (W-RLX), in order to execute our relaxed write, we have to send away a resource given by

$$\triangle \mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}(1) = \triangle \left( \exists f \in [0,1].\, \texttt{a.data} \overset{f}{\mapsto} v * \boxed{\gamma : f^-} * \boxed{\delta : (1-f)^-} \right).$$

Since we have not executed a release fence, we can only send away resources that are invariant under the $\triangle$ modality. The only non-ghost resource invariant under $\triangle$ is the empty resource. Therefore, we have to choose $f$ to be 0, in order to exploit the equivalence $\texttt{a.data} \overset{0}{\mapsto} v \iff \texttt{emp} \iff \triangle\texttt{emp}$.

Setting $f$ to 0 dealt with the $\texttt{a.data} \overset{f}{\mapsto} v$ part of the invariant. We now have to produce the rest of the invariant: the ghosts $\boxed{\gamma : 0^-}$ and $\boxed{\delta : 1^-}$. The $\gamma$ ghost we already have, and the $\delta$ one can be produced using the $\boxed{\delta : 0^-} \iff \boxed{\delta : 1^-} * \boxed{\delta : 1^+}$ equivalence.

Before releasing $\texttt{a.data} \overset{0}{\mapsto} v * \boxed{\gamma : 0^-} * \boxed{\delta : 1^-}$, we will exploit the $\boxed{\gamma : 0^-} \iff \boxed{\gamma : 0^-} * \boxed{\gamma : 0^+}$ equivalence in order to keep the $\boxed{\gamma : 0^+}$ ghost for ourselves.

We can now finally release the required resource, and what we are left with is $\texttt{a.data} \overset{1}{\mapsto} v * \boxed{\gamma : 0^+} * \boxed{\delta : 1^+}$, which is exactly the ARC predicate from Definition 2, with the existentially quantified $q$ set to be 1.

### 4.2 Function `read`

Verifying `read` is trivial. The ARC predicate from Definition 2 tells us that we have some positive fraction $q$ of the access permission for `a.data`, which allows us to execute the non-atomic read and return the value stored in `a.data`.

### 4.3 Implementing `fetch_and_add`

Before continuing with the proofs of `clone` and `drop`, let us take a step back and look at the `fetch_and_add` instruction used in those two functions. As mentioned in §3.3, `fetch_and_add` can be implemented using CAS instructions. The implementation of `fetch_and_add` using CAS is given in Fig. 7, together with the specification that will be used in the next two subsections.

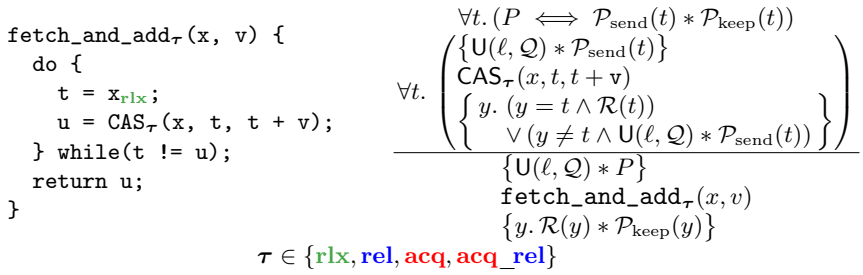



**Fig. 7.** Fetch and add implemented using CAS.

Proving the specification of `fetch_and_add` correct is simple, and we will not be going into details of it here. On the other hand, the specification looks quite daunting and deserves a closer look.

In the precondition, we are given the update permission $\mathsf{U}(\ell, \mathcal{Q})$ and some resource $P$.

The first premise of the specification allows us to decide how to split the resource $P$ *depending on the value that we will end up updating*. If the value modified is $v$, we want to keep the resource $\mathcal{P}_{\text{keep}}(v)$, while sending $\mathcal{P}_{\text{send}}(v)$ away.

The second premise deals with the atomic update of the location $\ell$ from $t$ to $t + v$. We need to prove that upon successful update we can send away $\mathcal{P}_{\text{send}}(t)$, and acquire $\mathcal{R}(t)$.

After executing the `fetch_and_add` instruction, in the postcondition we get $\mathcal{R}(y) * \mathcal{P}_{\text{keep}}(y)$, with $y$ being the value stored at the location $\ell$ prior to the update taking place. $\mathcal{R}(y)$ is what we acquired by updating $\ell$, while $\mathcal{P}_{\text{keep}}(y)$ is the part we kept from the original resource $P$ we had in the precondition.

Using the `fetch_and_add` specification boils down to deciding how we want to split the the resource we have for each particular value, and then applying appropriate CAS rules to satisfy the second precondition of the rule.

### 4.4 Function `clone`

For the `clone` function, we are required to prove two things: (1) executing `clone` produces an additional ARC resource, and (2) `clone` never increments the value of the reference counter from 0 to 1.

First, let us assume that the value read by the `fetch_and_add` is 0. In that case (in accordance with the rule from Fig. 7) we decide to put $\boxed{\delta : q^+}$ into $\mathcal{P}_{\texttt{keep}}$. Since $q > 0$, assertions $\boxed{\delta : q^+}$ and $\mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}(0) = \boxed{\gamma : 0^-} * \boxed{\delta : 0^-}$ are incompatible ($q > 0 \land \boxed{\delta : q^+} * \boxed{\delta : 0^-} \Rightarrow \mathsf{false}$), and we can use the (CAS-$\bot$) rule to conclude that the value 0 could not have been read.

Now that we know that the value read is not 0, we need, in cases where we read some positive value of the reference counter, to somehow produce an additional ARC resource.

When executing `fetch_and_add`, we are going to keep all the resources we have to ourselves, which means that we have to satisfy the invariant for the incremented value using only what is already there in the invariant for the original value. Fortunately, our invariant is designed in such a way that for any $c > 0$, the equivalence $\mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}(c) \iff \mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}(c+1) * \boxed{\gamma : 1^+}$ holds. Using this equivalence, when incrementing the reference counter from $c$ to $c+1$, we obtain the ownership of the ghost assertion $\boxed{\gamma : 1^+}$.

Adding the newly acquired ghost resource to the ARC resource we already have allows us to "produce" an additional ARC resource. In order to do that, we have to use the following three equivalences: $\texttt{a.data} \overset{q}{\mapsto} v \iff \texttt{a.data} \overset{\frac{q}{2}}{\mapsto} v * \texttt{a.data} \overset{\frac{q}{2}}{\mapsto} v$, $\boxed{\gamma : (1-q)^+} * \boxed{\gamma : 1^+} \iff \boxed{\gamma : \left(1-\frac{q}{2}\right)^+} * \boxed{\gamma : \left(1-\frac{q}{2}\right)^+}$, and $\boxed{\delta : q^+} \iff \boxed{\delta : \frac{q}{2}^+} * \boxed{\delta : \frac{q}{2}^+}$. Using those equivalences, it is easy to see that the implication $\mathsf{ARC}_{\gamma,\delta}(\texttt{a.data}, v) * \boxed{\gamma : 1^+} \Rightarrow \mathsf{ARC}_{\gamma,\delta}(\texttt{a.data}, v) * \mathsf{ARC}_{\gamma,\delta}(\texttt{a.data}, v)$ holds.

Please note the importance of the fact that the only ownership we obtained when updating the counter was of a ghost state. Since we are executing an update of the relaxed kind, any non-ghost resources acquired would be burdened by the $\triangledown$ modality, and thus unusable.

### 4.5 Function `drop`

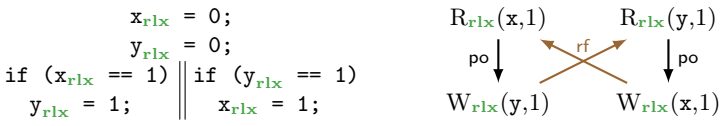
When verifying the `drop` function, we can establish that the value of the reference counter is not 0 in exactly the same way we have done it for the `clone` function in §4.4. We are now left with two distinct cases.

First case is when the decrementing the counter does not bring the counter down to zero, i.e., the value of the counter is being decremented from some value $c > 1$. In this case, we are going to release all the resources held by the ARC predicate, and push them into the invariant. It is easy to see that $\mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}(c) * \texttt{a.data} \overset{q}{\mapsto} v * \boxed{\gamma : (1-q)^+} * \boxed{\delta : q^+} \Rightarrow \mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}(c-1)$ holds for any $q \in \langle 0, 1]$ and $c > 1$, which reestablishes the invariant for the decremented value, and leaves us with the empty resource.

Note the importance of the `fetch_and_add` being of the release kind, which (trough the (CAS-REL) rule) enables us to release all the resources we have.

In the second case, the decrement brings the reference count down to 0. Since the value read from the counter is 1, we know that the resource being held by the invariant is $\mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}(1) = \texttt{a.data} \overset{f}{\mapsto} v * \boxed{\gamma : f^-} * \boxed{\delta : (1-f)^-}$, for some fraction $f \in [0, 1]$. We are going to take the read permission to the `data` field out of the invariant, and we are going to release the ghost resources held by the ARC predicate back into the invariant.

The ghost resource held by the ARC predicate is $\boxed{\gamma : (1-q)^+} * \boxed{\delta : q^+}$, for some $q \in \langle 0, 1]$. In order for this assertion to be compatible with $\boxed{\gamma : f^-} * \boxed{\delta : (1-f)^-}$, the resource that is already inside the invariant, it is necessary




(a) C11 model allows $\mathtt{x} = \mathtt{y} = 1$. (b) Problematic cyclic execution.

**Fig. 8.** Out-of-thin-air behavior due to a cycle in the $\mathsf{po} \cup \mathsf{rf}$ relation.

to have $q + f = 1$, and in that case we have $\boxed{\gamma : (1-q)^+} * \boxed{\delta : q^+} * \boxed{\gamma : f^-} * \boxed{\delta : (1-f)^-} \Rightarrow \boxed{\gamma : 0^-} * \boxed{\delta : 0^-}$, establishing the $\mathcal{Q}_{\gamma,\delta,v,\mathtt{a.data}}(0)$ invariant.

While establishing the $\mathcal{Q}_{\gamma,\delta,v,\mathtt{a.data}}(0)$ invariant, we were also able to prove $q + f = 1$, which is a pure assertion. According to the (CAS-REL) rule, we can use this fact in the postcondition.

After executing the decrement, we have $\mathtt{a.data} \overset{q}{\mapsto} v * \triangledown \mathtt{a.data} \overset{f}{\mapsto} v$ in the postcondition. The $f$ fraction of the access permission, which we obtained from the invariant, is under $\triangledown$, because the `fetch_and_add` was of the release kind, and we still have to wait for the acquire fence in order to use any resources taken from the invariant. Since we are in the case where the original value of the reference counter was 1, the very next instruction is exactly the acquire fence.

After the fence clears the $\triangledown$ modality (F-ACQ), the resource we own is transformed into $\mathtt{a.data} \overset{q}{\mapsto} v * \mathtt{a.data} \overset{f}{\mapsto} v \iff \mathtt{a.data} \overset{q+f}{\mapsto} v \iff \mathtt{a.data} \overset{1}{\mapsto} v$. These equivalences hold because we know $q + f = 1$, as proven earlier.

With this, the proof of Theorem 1 is concluded.

## 5 Discussion

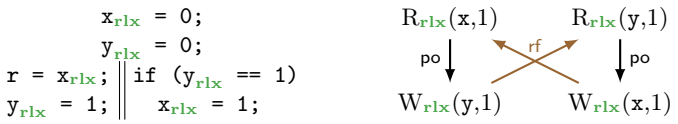
In this section, we are going to discuss the strengthening of the C11 memory model which is assumed by the FSL soundness proof and how it affects the ARC verification (§5.1). Further, in §5.2, we will discuss the necessity of this assumption showing that the logic is unsound in its absence. Finally, in §5.3 we will talk about a possible way to extend FSL with the support for deallocation.

### 5.1 The Additional Acyclity Assumption

As mentioned in the introduction, FSL is proven sound with respect to a strengthening of the C11 model. The strengthening is put in place in order to prevent the so called *out-of-thin-air* reads that are allowed by the original C11 model.

The problem arises because C11 is very lenient in what kind of cycles are allowed to be formed by the *program order* and *reads from* relations.

– The program order ($\mathsf{po}$) tells us about the ordering of the events within each execution thread. More precisely, $\mathsf{po}(a, b)$ means that the events $a$ and $b$ belong to the same thread, and $a$ precedes $b$.




(a) We can observe $\mathtt{r} = 1$! (b) Cyclic execution explaining $\mathtt{r} = 1$.

**Fig. 9.** Load buffering (allowed on Power and ARM).

– The reads from relation ($\mathsf{rf}$) relates writes and reads that read from those writes: $\mathsf{rf}(w, r)$ says that the read event $r$ reads the value written by the write event $w$.

Figure 8a shows a program with an undesirable behavior resulting from a cycle in $\mathsf{po} \cup \mathsf{rf}$. The C11 model allows the program to set both $\mathtt{x}$ and $\mathtt{y}$ to 1, due to the allowed "cyclic" execution shown in Fig. 8b.

As noted in [5, 35], this kind of behavior inhibits even the simplest forms of thread-local reasoning for relaxed accesses.

The simplest way to rectify the problem of out-of-thin-air behaviors is to forbid cycles in the $\mathsf{po} \cup \mathsf{rf}$ relation altogether. Forbidding these cycles requires the smallest possible intervention in the C11 model, namely adding just one axiom requiring acyclicity of $\mathsf{po} \cup \mathsf{rf}$. This is the solution employed by the soundness proofs of both RSL [35], and FSL [13] in order to restore sane reasoning principles for relaxed accesses under the C11 memory model. Apart from being used in RSL and FSL, this "patch" is also advocated by Boehm and Demsky [7].

Requiring $\mathsf{po} \cup \mathsf{rf}$ to be acyclic, however, does come with some implementation cost. First, it invalidates some compiler optimizations (namely, the reordering of a relaxed store above a relaxed load), and requires a slightly more expensive compilation scheme to the Power and ARM architectures. The problem is that these hardware architectures allow some executions with $\mathsf{po} \cup \mathsf{rf}$ cycles. Consider, for example, *load buffering*, shown in Fig. 9a. The weak behavior, returning $\mathtt{r} = 1$ is forbidden by the strengthened C11 model, but allowed by Power and ARM if the relaxed accesses are compiled to plain loads and stores. Intuitively, the behavior may arise if the hardware reorders the read from $\mathtt{x}$ and the write to $\mathtt{y}$ in the left thread, which do not depend on each other.

Note that the execution in Fig. 9b, which explains the load buffering behavior, is exactly the same as the execution we deemed undesirable in Fig. 8b. The difference between these two examples is the possibility of reordering two independent instructions in Fig. 9a, while in Fig. 8a the writes depend on the reads, and these dependencies should render any reorderings invalid. The C11 model does not model the dependencies between memory accesses, which makes it unable to differentiate between executions in Figs. 8 and 9.

As noted by Boehm and Demsky in [7], in order to obtain acyclic $\mathsf{po} \cup \mathsf{rf}$, it is enough to forbid load-to-store reordering. On x86-TSO acyclicity of $\mathsf{po} \cup \mathsf{rf}$ comes at no additional cost, since the architecture does not allow reordering of loads and the subsequent stores. On Power and ARM, load-to-store reordering

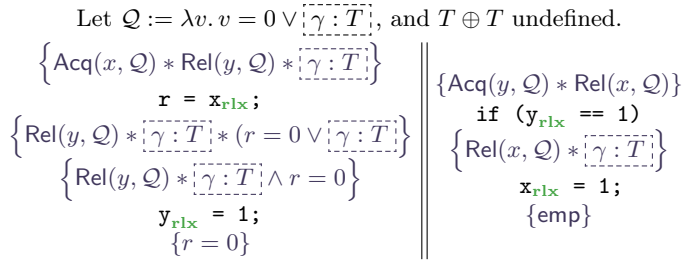



**Fig. 10.** Using ghosts we can establish absence of load buffering.

can be avoided by placing a *false dependency* (i.e., a conditional branch to the next instruction) between every relaxed load and subsequent relaxed stores.

**Acyclic po ∪ rf and ARC** It is interesting to note that with algorithms like ARC, which predominantly use atomic updates, and do not have many atomic reads, ensuring the acyclicity of po ∪ rf on Power and ARM comes for free.

The reason for this comes from the way atomic update instructions are implemented on Power and ARM [31]. When compiling atomic updates, a conditional branch is placed after the load instruction, which induces a dependency between the load and any subsequent stores. This means that the false dependencies are not necessary when compiling atomic updates.

In the case of ARC, a false dependency needs to be placed after the relaxed read in the implementation of `fetch_and_add` in Fig. 7. If `fetch_and_add` is implemented as a primitive, as it actually is in practice, then it comes without the burden of false dependencies. Therefore, there is no additional implementation cost for ensuring that ARC runs under the strengthened C11 model.

### 5.2 Without the Acyclicity Assumption Ghosts Are Too Strong

Ruling out po∪rf cycles is the simplest but not the only way of ruling out "out-of-thin-air" behaviors. In fact, during the last year, we saw the emergence of several new memory models [17, 19, 28] aimed at eliminating out-of-thin-air behaviors without completely forbidding cycles within the po∪rf relation. All these models allow the weak behavior of the load buffering program, while forbidding the weak behavior of the version with dependencies in both threads.

We will now show that our extension of FSL with ghost state is unsound with respect to these models. As can be seen in Fig. 10, FSL outfitted with ghost state is strong enough to prove that the weak behavior of the load buffering program does not happen, which in turn means that FSL is not sound for any of the new models which allow that behavior.

The proof uses a single ghost location $\gamma$ holding a non-duplicable token $T$. We then use the $\mathcal{Q}(v)$ resource invariant to say that either $v = 0$ or the location owns the token. Since the token is non-duplicable, we thus encode the invariant saying that at most one of `x` and `y` can have a non-zero value. Initially, both

locations store the value 0, so the ghost token is given to the left thread. Using the token, the first thread can thus assert that $r = 0$, and then use it to write 1 to `y`. The right thread can conversely gain the token by reading $\texttt{y} = 1$ and then use it to write 1 to `x`.

An interesting thing of note is that all the examples (that we are aware of) showing unsoundness of FSL under these new models rely on the use of ghosts, and in the ability to transfer them without any synchronization. In a sense, being able to fully transfer the ownership of the ghost state without any synchronization exposes the acyclicity of the $\mathsf{po} \cup \mathsf{rf}$ relation.

There are thus two main open questions regarding the connection of FSL, and the memory models that do not rely on the acyclic $\mathsf{po} \cup \mathsf{rf}$ assumption.

1. *Is FSL without ghosts sound under any of the models that do not require* $\mathsf{po} \cup \mathsf{rf}$ *to be acyclic?* We strongly suspect that FSL without ghosts is sound under the recent promising model of Kang et al. [19], but proving that this is indeed the case is a highly non-trivial task.
2. In the case of the affirmative answer to the first question, *can we come up with the rules for the ghost state which would allow us to verify algorithms like ARC?* A possibility would be to somehow restrict the (GHOST-MOD) rule so that it may be used only in conjuction with a release write. Such a restriction would preserve the proof of ARC, while ruling out the proof of load buffering. Its soundness with respect to models such as [17, 19, 28], however, is unclear.

### 5.3 Deallocation

The proof of soundness of FSL already ensures that if a thread owns the full permission to access a non-atomic location, then there are no other threads that concurrently hold an access permission to the same location. Using this fact, proving that it is safe to deallocate a non-atomic location when holding the full access permission to it is a purely technical matter.

In order to enable deallocation of the atomic locations, we would have to outfit atomic locations with permissions, and show that (for a single location) the full permission cannot coexist concurrently with any other permission. This result should follow from the same line of reasoning as the corresponding result for the non-atomic locations.

In the context of our correctness proof of ARC, the necessary permission for deallocating the atomic variable `a.count` could be obtained in exactly the same way as we obtained the full permission of `a.data` (see §4.5).

## 6 Related Work

In this section we would like to call attention to some related work that was not already discussed in §5. We divide our discussion in two parts: in §6.1 we discuss other program logics for reasoning about weak memory, and in §6.2 we turn our attention to some other approaches for establishing program correctness under weak memory.

### 6.1 Program Logics

Apart from FSL's predecessor, RSL [35], the only other separation logic for the C11 memory model is GPS [34]. Even though GPS handles the ownership transfer in a more flexible way than FSL (using protocols and escrows), GPS is unable to reason about programs that use relaxed memory accesses, such as ARC. The reason for this limitation of GPS is the fact that GPS works under the release-acquire fragment of the C11 memory model.

He et al. [14] have proposed an extension of GPS with FSL-style modalities, to give it support for relaxed accesses and memory fences. As the original FSL, this extension of GPS does not have support for atomic updates, which makes it inapplicable to programs like ARC. Additionally, unlike FSL, this extension of GPS lacks a soundness proof.

It would be interesting to explore adapting GPS-style protocols to FSL, in order to make FSL applicable to an even wider range of programs that require more sophisticated forms of reasoning.

Apart from the separation logics, there is an Owicki-Gries-based logic called OGRA [21] for reasoning about the C11 memory model, but it also handles only the release-acquire fragment of the C11 model. Other program logics for weak memory [30, 32] have been focused on the x86-TSO memory model, which is stronger than the one assumed by FSL.

### 6.2 Other Approaches

Aside from program logics, there are model checking tools for programs with C11-style atomics. Worth noting is CDSChecker [25] which includes support for relaxed accesses and memory fences. CDSChecker is designed to conduct unit tests on concurrent programs, and cannot be used to verify correctness.

An alternative approach to reasoning about weak memory behaviors is to restore sequential consistency. This can be done by placing fences or stronger atomic accesses in order to eliminate weak behaviors [4, 24], or by proving robustness theorems [9, 11, 20] stating conditions under which programs have no observable weak behaviors. These approaches are not applicable to performance-critical algorithms such as ARC, which are exploiting weak memory consistency. Placing additional fences or using stronger memory accesses to restore sequential consistency would go against the basic design principles of these algorithms.

Recently, Alglave proposed an invariance method for proving program correctness under weak memory [3]. This approach is parametric with the respect to the memory model, and so could be applied to the C11 memory model. It is, however, non-compositional, which makes using it to obtain a correctness proof for the ARC algorithm difficult.

**Acknowledgments** We would like to thank Soham Chakraborty, Rayna Dimitrova, Jeehoon Kang, Ori Lahav, Alex Summers, and the ESOP'17 reviewers for their feedback.

## References


1. Atomic reference counter (ARC) documentation. `https://doc.rust-lang.org/std/sync/struct.Arc.html`
2. The Rust programming language. `https://www.rust-lang.org/`
3. Alglave, J.: Simulation and invariance for weak consistency. In: Rival, X. (ed.) SAS 2016. pp. 3–22. Springer (2016)
4. Alglave, J., Kroening, D., Nimal, V., Poetzl, D.: Don't sit on the fence - A static analysis approach to automatic fence insertion. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 508–524. Springer (2014)
5. Batty, M., Dodds, M., Gotsman, A.: Library abstraction for C/C++ concurrency. In: POPL 2013. pp. 235–248. ACM (2013)
6. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL 2011. pp. 55–66. ACM (2011)
7. Boehm, H., Demsky, B.: Outlawing ghosts: avoiding out-of-thin-air results. In: Singer, J., Kulkarni, M., Harris, T. (eds.) MSPC 2014. pp. 7:1–7:6. ACM (2014)
8. Bornat, R., Calcagno, C., O'Hearn, P.W., Parkinson, M.J.: Permission accounting in separation logic. In: POPL 2005. pp. 259–270. ACM (2005)
9. Bouajjani, A., Meyer, R., Möhlmann, E.: Deciding robustness against total store ordering. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 428–440. Springer (2011)
10. Boyland, J.: Checking interference with fractional permissions. In: SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer (2003)
11. Derevenetc, E., Meyer, R.: Robustness against Power is PSpace-complete. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) ICALP 2014, Part II. LNCS, vol. 8573, pp. 158–170. Springer (2014)
12. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M.J., Yang, H.: Views: compositional reasoning for concurrent programs. In: Giacobazzi, R., Cousot, R. (eds.) POPL 2013. pp. 287–300. ACM (2013)
13. Doko, M., Vafeiadis, V.: A program logic for C11 memory fences. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. pp. 413–430. Springer (2016)
14. He, M., Vafeiadis, V., Qin, S., Ferreira, J.F.: Reasoning about fences and relaxed atomics. In: PDP 2016. pp. 520–527. IEEE Computer Society (2016)
15. ISO/IEC 14882:2011: Programming language C++ (2011)
16. ISO/IEC 9899:2011: Programming language C (2011)
17. Jeffrey, A., Riely, J.: On thin air reads towards an event structures model of relaxed memory. In: LICS 2016. pp. 759–767. ACM (2016)
18. Jensen, J.B., Birkedal, L.: Fictional separation logic. In: Seidl, H. (ed.) ESOP 2012. pp. 377–396 (2012)
19. Kang, J., Hur, C.K., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: POPL 2017. pp. 175–189. ACM (2017)
20. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: Bodík, R., Majumdar, R. (eds.) POPL 2016. pp. 649–662. ACM (2016)
21. Lahav, O., Vafeiadis, V.: Owicki-Gries reasoning for weak memory models. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015, Part II. LNCS, vol. 9135, pp. 311–323. Springer (2015)
22. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers 28(9), 690–691 (1979)
23. Ley-Wild, R., Nanevski, A.: Subjective auxiliary state for coarse-grained concurrency. In: Giacobazzi, R., Cousot, R. (eds.) POPL 2013. pp. 561–574. ACM (2013)

24. Meshman, Y., Rinetzky, N., Yahav, E.: Pattern-based synthesis of synchronization for the C++ memory model. In: Kaivola, R., Wahl, T. (eds.) FMCAD 2015. pp. 120–127. IEEE (2015)
25. Norris, B., Demsky, B.: CDSChecker: Checking concurrent data structures written with C/C++ atomics. In: Hosking, A.L., Eugster, P.T., Lopes, C.V. (eds.) OOPSLA 2013. pp. 131–150. ACM (2013)
26. O'Hearn, P.W.: Resources, concurrency and local reasoning. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. pp. 49–67. Springer (2004)
27. O'Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. ACM Trans. Program. Lang. Syst. 31(3) (2009)
28. Pichon-Pharabod, J., Sewell, P.: A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In: Bodík, R., Majumdar, R. (eds.) POPL 2016. pp. 622–633. ACM (2016)
29. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS 2002. pp. 55–74. IEEE Computer Society (2002)
30. Ridge, T.: A rely-guarantee proof system for x86-TSO. In: Leavens, G.T., O'Hearn, P.W., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 55–70. Springer (2010)
31. Sarkar, S., Memarian, K., Owens, S., Batty, M., Sewell, P., Maranget, L., Alglave, J., Williams, D.: Synchronising C/C++ and POWER. In: PLDI 2012. pp. 311–322. ACM (2012)
32. Sieczkowski, F., Svendsen, K., Birkedal, L., Pichon-Pharabod, J.: A separation logic for fictional sequential consistency. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 736–761. Springer (2015)
33. Tassarotti, J., Dreyer, D., Vafeiadis, V.: Verifying read-copy-update in a logic for weak memory. In: Grove, D., Blackburn, S. (eds.) PLDI 2015. pp. 110–120. ACM (2015)
34. Turon, A., Vafeiadis, V., Dreyer, D.: GPS: Navigating weak-memory with ghosts, protocols, and separation. In: Black, A.P., Millstein, T.D. (eds.) OOPSLA 2014. pp. 691–707. ACM (2014)
35. Vafeiadis, V., Narayan, C.: Relaxed separation logic: A program logic for C11 concurrency. In: Hosking, A.L., Eugster, P.T., Lopes, C.V. (eds.) OOPSLA 2013. pp. 867–884. ACM (2013)

## Appendix A Complete FSL++ Syntax and Rules

Here we present the full syntax and rules of FSL, and briefly discuss the technical differences between the actual FSL rules and the simplified version of the rules presented in §3.

Definitions 3 and 4 present the concurrent programing language used by FSL++, and the syntax of FSL++ assertions. Fig. 11 contains standard inference rules inherited from concurrent separation logic. Remaining FSL++ rules can be divided into rules regarding non-atomic accesses (Fig. 12), atomic accesses (Fig. 15), fences (Fig. 13), and ghosts (Fig. 14).

**Definition 3 (Programming language).** *FSL++ uses the programming language specified by the following grammar*

$$\begin{aligned} v \in \mathsf{Val} &::= \ell \mid n && \text{where } \ell \in \mathsf{Loc}, n \in \mathbb{N} \\ e \in \mathsf{AExp} &::= x \mid v && \text{where } x \in \mathsf{Var} \\ E \in \mathsf{Exp} &::= e \mid \mathbf{let}\ x = E\ \mathbf{in}\ E' \mid \mathbf{if}\ e\ \mathbf{then}\ E\ \mathbf{else}\ E' \\ &\quad \mid \mathbf{repeat}\ E\ \mathbf{end} \mid E_1 \| E_2 \mid \mathbf{alloc}() \\ &\quad \mid [e]_\xi \mid [e]_\zeta := e' \mid \mathsf{CAS}_{\tau,\sigma}(e, e', e'')), \\ \text{where } \xi &\in \{\mathbf{acq}, \mathbf{rlx}, \mathbf{na}\}, \zeta \in \{\mathbf{rel}, \mathbf{rlx}, \mathbf{na}\}, \\ \tau &\in \{\mathbf{acq\_rel}, \mathbf{acq}, \mathbf{rel}, \mathbf{rlx}\}, \sigma \in \{\mathbf{acq}, \mathbf{rlx}\}. \end{aligned}$$

Atomic expressions, $e \in \mathsf{AExp}$, consist of variables and values (locations and numbers). Program expressions, $E \in \mathsf{Exp}$, consist of atomic expressions, let-bound computations, conditionals, loops, parallel composition, memory allocation, loads, stores, and atomic compare-and-swap (CAS) instructions.

As in C, in conditional expressions we treat zero as false and non-zero values as true. The construct **repeat** $E$ **end** executes $E$ repeatedly until it returns a non-zero value.

The CAS instructions are parametrized by two access modes. In case of a successful CAS, an atomic update event of the type $\tau$ will be generated, and a failed CAS will generate an atomic read event of type $\sigma$. The simplified CAS instruction from §3.3, which is parametrized by only one access mode, can be obtained by setting $\sigma = \mathbf{rlx}$.

**Definition 4 (FSL++ assertions).** *Let* $(\mathbf{P}, +, \varepsilon, \mathbb{1})$ *be a permission structure*[1]*, and* $(\mathbf{G}, \oplus)$ *a partial commutative monoid. FSL++ assertions are defined by the grammar*

$$\begin{aligned} P, Q ::= {} & \mathsf{false} \mid P \to Q \mid P * Q \mid \forall x.\, P \\ & \mid \mathsf{emp} \mid e \overset{p}{\mapsto} e' \mid \mathsf{Uninit}(e) \\ & \mid \mathsf{Rel}(e, \mathcal{Q}) \mid \mathsf{Acq}(e, \mathcal{Q}) \mid \mathsf{RMWAcq}(e, \mathcal{Q}) \mid \mathsf{Init}(e) \\ & \mid \triangle P \mid \triangledown P \mid \boxed{e : g}, \end{aligned}$$

*where $e$ is an arithmetic expression, $p \in \mathbf{P}$, and $g \in \mathbf{G}$.*

[1] Permission structure has been defined in §3.2.

Note that the full FSL++ syntax contains three assertions not mentioned in §3 ($\mathsf{Uninit}(e)$, $\mathsf{Init}(e)$, $\mathsf{RMWAcq}(e, \mathcal{Q})$), and it is missing $\mathsf{U}(e, \mathcal{Q})$, which was presented in §3.3.

The $\mathsf{Uninit}(\ell)$ assertion represents ownership of an uninitialized non-atomic location $\ell$. This is the ownership we get after allocating a non-atomic location.

The $\mathsf{Init}(\ell)$ assertion represents knowledge that the atomic location $\ell$ has been initialized. This assertion is required by atomic reads and updates, and it is produced by atomic writes.

The $\mathsf{RMWAcq}(\ell, \mathcal{Q})$ assertion, together with the $\mathsf{Rel}(\ell, \mathcal{Q}')$ assertion gives us the permission to execute atomic updates. The update permission $\mathsf{U}$, from §3.3, can be defined as $\mathsf{U}(\ell, \mathcal{Q}) := \mathsf{Rel}(\ell, \mathcal{Q}) * \mathsf{RMWAcq}(\ell, \mathcal{Q}) * \mathsf{Init}(\ell)$. Properties (U-SPLIT), (U-REL-SPLIT), and (U-ACQ-SPLIT) follow from Lemma 2.

**Lemma 2 (Basic properties of FSL++ assertions).** *The following equivalences universally hold.*

$$\begin{aligned}
\mathsf{Init}(\ell) &\iff \mathsf{Init}(\ell) * \mathsf{Init}(\ell) \\
\mathsf{Rel}(\ell, \mathcal{Q}) &\iff \mathsf{Rel}(\ell, \mathcal{Q}) * \mathsf{Rel}(\ell, \mathcal{Q}) \\
\mathsf{RMWAcq}(\ell, \mathcal{Q}) &\iff \mathsf{RMWAcq}(\ell, \mathcal{Q}) * \mathsf{RMWAcq}(\ell, \mathcal{Q}) \\
\mathsf{RMWAcq}(\ell, \mathcal{Q}) &\iff \mathsf{RMWAcq}(\ell, \mathcal{Q}) * \mathsf{Acq}(\ell, \lambda v.\mathsf{emp}) \\
\mathsf{Acq}(\ell, \mathcal{Q}_1) * \mathsf{Acq}(\ell, \mathcal{Q}_2) &\iff \mathsf{Acq}(\ell, \lambda v.\, \mathcal{Q}_1(v) * \mathcal{Q}_2(v))
\end{aligned}$$

Some rules in Figs. 13 and 15 require certain assertions to be *precise*, or *normalizable*. The definition of precision is standard [27], while the notion of normalizability has been introduced in [13].

The role of the normalizability conditions is to forbid ownership transfer of resources that appear under a modality. This condition poses no burden to the user of the logic, since Lemma 3 provides a simple syntactical test for normalizability.

**Lemma 3 (Sufficient condition for normalizability).** *If $P$ is a positive*[2] *FSL++ assertion that does not contain modalities, i.e., $\triangle$ and $\triangledown$ do not appear in $P$, then $P$ is normalizable.*

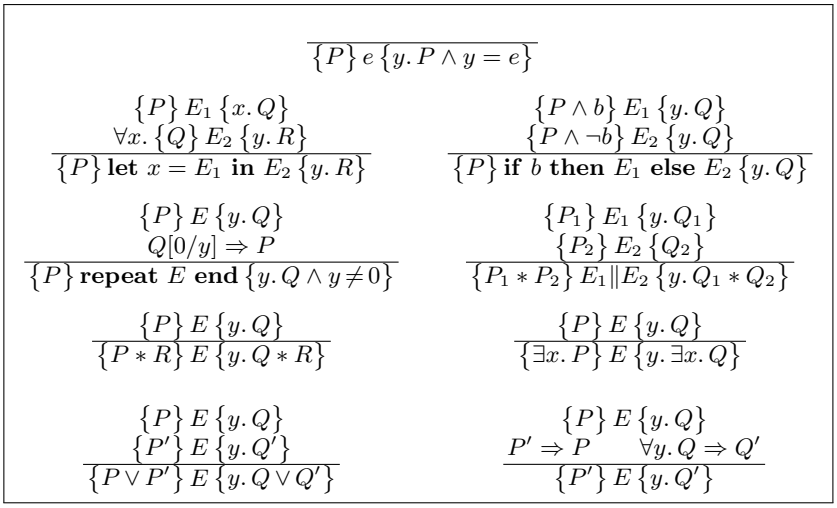
CAS rules in Fig. 15 acknowledge the fact that in case when the CAS fails, a read event is produced. Therefore, when CAS fails, atomic read rules take over. However, using Lemma 2 we can see that

$$\left\{\mathsf{RMWAcq}(\ell, \mathcal{Q}_{\mathsf{acq}})\right\}\, [\ell]_{\boldsymbol{\sigma}}\, \left\{\mathsf{RMWAcq}(\ell, \mathcal{Q}_{\mathsf{acq}})\right\}$$

holds for every $\boldsymbol{\sigma} \in \{\mathbf{rlx}, \mathbf{acq}\}$. With this in mind, it is straightforward to derive the simplified CAS rules presented in §3.3 from the general ones in Fig. 15.

It is worth noting that the verification of ARC requires only the simplified version of the CAS rules as presented in §3.3.

[2] An assertion is positive if the only logical connectives it contains are disjunction, conjunction, and separating conjunction.



$$\frac{}{\{P\}\, e\, \{y.\, P \wedge y = e\}}$$

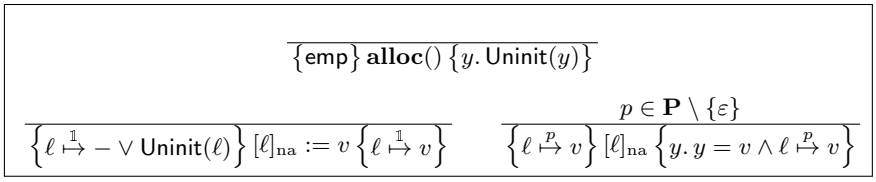
$$\frac{\{P\}\, E_1\, \{x.\, Q\} \quad \forall x.\, \{Q\}\, E_2\, \{y.\, R\}}{\{P\}\, \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2\, \{y.\, R\}} \qquad \frac{\{P \wedge b\}\, E_1\, \{y.\, Q\} \quad \{P \wedge \neg b\}\, E_2\, \{y.\, Q\}}{\{P\}\, \mathbf{if}\ b\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2\, \{y.\, Q\}}$$

$$\frac{\{P\}\, E\, \{y.\, Q\} \quad Q[0/y] \Rightarrow P}{\{P\}\, \mathbf{repeat}\ E\ \mathbf{end}\, \{y.\, Q \wedge y \neq 0\}} \qquad \frac{\{P_1\}\, E_1\, \{y.\, Q_1\} \quad \{P_2\}\, E_2\, \{Q_2\}}{\{P_1 * P_2\}\, E_1 \| E_2\, \{y.\, Q_1 * Q_2\}}$$

$$\frac{\{P\}\, E\, \{y.\, Q\}}{\{P * R\}\, E\, \{y.\, Q * R\}} \qquad \frac{\{P\}\, E\, \{y.\, Q\}}{\{\exists x.\, P\}\, E\, \{y.\, \exists x.\, Q\}}$$

$$\frac{\{P\}\, E\, \{y.\, Q\} \quad \{P'\}\, E\, \{y.\, Q'\}}{\{P \vee P'\}\, E\, \{y.\, Q \vee Q'\}} \qquad \frac{\{P\}\, E\, \{y.\, Q\} \quad P' \Rightarrow P \quad \forall y.\, Q \Rightarrow Q'}{\{P'\}\, E\, \{y.\, Q'\}}$$

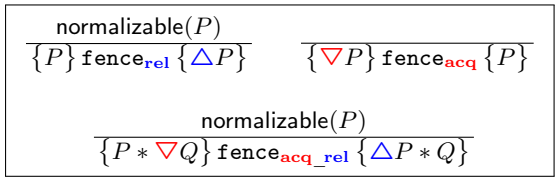
**Fig. 11.** Standard proof rules supported by FSL++.



$$\frac{}{\{\mathsf{emp}\}\, \mathbf{alloc}()\, \{y.\, \mathsf{Uninit}(y)\}}$$

$$\frac{}{\left\{\ell \overset{\mathbb{1}}{\mapsto} - \vee\, \mathsf{Uninit}(\ell)\right\}\, [\ell]_{\mathrm{na}} := v\, \left\{\ell \overset{\mathbb{1}}{\mapsto} v\right\}} \qquad \frac{p \in \mathbf{P} \setminus \{\varepsilon\}}{\left\{\ell \overset{p}{\mapsto} v\right\}\, [\ell]_{\mathrm{na}}\, \left\{y.\, y = v \wedge \ell \overset{p}{\mapsto} v\right\}}$$

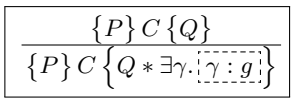
**Fig. 12.** FSL++ proof rules for non-atomic locations.



$$\frac{\mathsf{normalizable}(P)}{\{P\}\, \mathtt{fence}_{\mathbf{rel}}\, \{\triangle P\}} \qquad \frac{}{\{\triangledown P\}\, \mathtt{fence}_{\mathbf{acq}}\, \{P\}}$$

$$\frac{\mathsf{normalizable}(P)}{\{P * \triangledown Q\}\, \mathtt{fence}_{\mathbf{acq\_rel}}\, \{\triangle P * Q\}}$$

**Fig. 13.** FSL++ proof rules for memory fences.



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

**Fig. 14.** FSL++ ghost introduction rule.

$$\frac{\mathcal{Q}\colon \mathit{Values} \to \mathit{Assertions}}{\{\mathsf{emp}\}\ \mathbf{alloc}()\ \{\ell.\ \mathsf{Rel}(\ell,\mathcal{Q}) * \mathsf{Acq}(\ell,\mathcal{Q})\}}$$

$$\frac{\mathsf{normalizable}(\mathcal{Q}(v))}{\{\mathsf{Rel}(\ell,\mathcal{Q}) * \mathcal{Q}(v)\}\ [\ell]_{\mathbf{rel}} := v\ \{\mathsf{Init}(\ell)\}}$$

$$\frac{\forall v.\ \mathsf{precise}(\mathcal{Q}(v)) \land \mathsf{normalizable}(\mathcal{Q}(v))}{\begin{array}{c}\{\mathsf{Acq}(\ell,\mathcal{Q}) * \mathsf{Init}(\ell)\}\\ [\ell]_{\mathbf{acq}}\\ \{v.\ \mathsf{Acq}(\ell,\mathcal{Q}[v := \mathsf{emp}]) * \mathcal{Q}(v)\}\end{array}}$$

$$\frac{}{\{\mathsf{Rel}(\ell,\mathcal{Q}) * \triangle\mathcal{Q}(v)\}\ [\ell]_{\mathbf{rlx}} := v\ \{\mathsf{Init}(\ell)\}}$$

$$\frac{\forall v.\ \mathsf{precise}(\mathcal{Q}(v)) \land \mathsf{normalizable}(\mathcal{Q}(v))}{\begin{array}{c}\{\mathsf{Acq}(\ell,\mathcal{Q}) * \mathsf{Init}(\ell)\}\\ [\ell]_{\mathbf{rlx}}\\ \{v.\ \mathsf{Acq}(\ell,\mathcal{Q}[v := \mathsf{emp}]) * \triangledown\mathcal{Q}(v)\}\end{array}}$$

$$\frac{\mathcal{Q}\colon \mathit{Values} \to \mathit{Assertions}}{\{\mathsf{emp}\}\ \mathbf{alloc}()\ \{\ell.\ \mathsf{Rel}(\ell,\mathcal{Q}) * \mathsf{RMWAcq}(\ell,\mathcal{Q})\}}$$

**Let** $\mathsf{UPD}(\ell,\mathcal{Q}_{\mathtt{rel}},\mathcal{Q}_{\mathtt{acq}}) := \mathsf{Rel}(\ell,\mathcal{Q}_{\mathtt{rel}}) * \mathsf{RMWAcq}(\ell,\mathcal{Q}_{\mathtt{acq}}) * \mathsf{Init}(\ell)$ **in**

$$\frac{\begin{array}{c}\mathcal{Q}_{\mathtt{acq}}(v) \Rightarrow \exists z.\ \mathcal{A}(z) * \mathcal{T}(z)\\ \forall z.\ (P * \mathcal{T}(z) \Rightarrow \mathcal{Q}_{\mathtt{rel}}(v') \land \varphi(z))\\ \forall z.\ \mathsf{pure}(\varphi(z))\\ \mathsf{normalizable}(P)\\ \{\mathsf{UPD}(\ell,\mathcal{Q}_{\mathtt{rel}},\mathcal{Q}_{\mathtt{acq}}) * P\}\ [\ell]_{\boldsymbol{\sigma}}\ \{a.\ a \neq v \to R\}\\ \sigma \in \{\mathbf{acq},\mathbf{rlx}\}\end{array}}{\begin{array}{c}\{\mathsf{UPD}(\ell,\mathcal{Q}_{\mathtt{rel}},\mathcal{Q}_{\mathtt{acq}}) * P\}\\ \mathsf{CAS}_{\mathbf{acq\_rel},\boldsymbol{\sigma}}(\ell,v,v')\\ \left\{\begin{array}{l} a.\ (a = v \land \exists z.\ \mathcal{A}(z) \land \varphi(z))\\ \lor\ (a \neq v \land R)\end{array}\right\}\end{array}}$$

$$\frac{\begin{array}{c}\mathcal{Q}_{\mathtt{acq}}(v) \Rightarrow \exists z.\ \mathcal{A}(z) * \mathcal{T}(z)\\ \forall z.\ (P * \mathcal{T}(z) \Rightarrow \mathcal{Q}_{\mathtt{rel}}(v') \land \varphi(z))\\ \forall z.\ \mathsf{pure}(\varphi(z))\\ \mathsf{normalizable}(P)\\ \{\mathsf{UPD}(\ell,\mathcal{Q}_{\mathtt{rel}},\mathcal{Q}_{\mathtt{acq}}) * P\}\ [\ell]_{\boldsymbol{\sigma}}\ \{a.\ a \neq v \to R\}\\ \sigma \in \{\mathbf{acq},\mathbf{rlx}\}\end{array}}{\begin{array}{c}\{\mathsf{UPD}(\ell,\mathcal{Q}_{\mathtt{rel}},\mathcal{Q}_{\mathtt{acq}}) * P\}\\ \mathsf{CAS}_{\mathbf{rel},\boldsymbol{\sigma}}(\ell,v,v')\\ \left\{\begin{array}{l} a.\ (a = v \land \exists z.\ \triangledown\mathcal{A}(z) \land \varphi(z))\\ \lor\ (a \neq v \land R)\end{array}\right\}\end{array}}$$

$$\frac{\begin{array}{c}\mathcal{Q}_{\mathtt{acq}}(v) \Rightarrow \exists z.\ \mathcal{A}(z) * \mathcal{T}(z)\\ \forall z.\ (P * \mathcal{T}(z) \Rightarrow \mathcal{Q}_{\mathtt{rel}}(v') \land \varphi(z))\\ \forall z.\ \mathsf{pure}(\varphi(z))\\ \{\mathsf{UPD}(\ell,\mathcal{Q}_{\mathtt{rel}},\mathcal{Q}_{\mathtt{acq}}) * \triangle P\}\ [\ell]_{\boldsymbol{\sigma}}\ \{a.\ a \neq v \to R\}\\ \sigma \in \{\mathbf{acq},\mathbf{rlx}\}\end{array}}{\begin{array}{c}\{\mathsf{UPD}(\ell,\mathcal{Q}_{\mathtt{rel}},\mathcal{Q}_{\mathtt{acq}}) * \triangle P\}\\ \mathsf{CAS}_{\mathbf{acq},\boldsymbol{\sigma}}(\ell,v,v')\\ \left\{\begin{array}{l} a.\ (a = v \land \exists z.\ \mathcal{A}(z) \land \varphi(z))\\ \lor\ (a \neq v \land R)\end{array}\right\}\end{array}}$$

$$\frac{\begin{array}{c}\mathcal{Q}_{\mathtt{acq}}(v) \Rightarrow \exists z.\ \mathcal{A}(z) * \mathcal{T}(z)\\ \forall z.\ (P * \mathcal{T}(z) \Rightarrow \mathcal{Q}_{\mathtt{rel}}(v') \land \varphi(z))\\ \forall z.\ \mathsf{pure}(\varphi(z))\\ \{\mathsf{UPD}(\ell,\mathcal{Q}_{\mathtt{rel}},\mathcal{Q}_{\mathtt{acq}}) * \triangle P\}\ [\ell]_{\boldsymbol{\sigma}}\ \{a.\ a \neq v \to R\}\\ \sigma \in \{\mathbf{acq},\mathbf{rlx}\}\end{array}}{\begin{array}{c}\{\mathsf{UPD}(\ell,\mathcal{Q}_{\mathtt{rel}},\mathcal{Q}_{\mathtt{acq}}) * \triangle P\}\\ \mathsf{CAS}_{\mathbf{rlx},\boldsymbol{\sigma}}(\ell,v,v')\\ \left\{\begin{array}{l} a.\ (a = v \land \exists z.\ \triangledown\mathcal{A}(z) \land \varphi(z))\\ \lor\ (a \neq v \land R)\end{array}\right\}\end{array}}$$

$$\frac{\begin{array}{c}\mathcal{Q}(v) * P \Rightarrow \mathsf{false}\\ \{\mathsf{UPD}(\ell,\mathcal{Q}_{\mathtt{rel}},\mathcal{Q}_{\mathtt{acq}}) * P\}\ [\ell]_{\boldsymbol{\sigma}}\ \{a.\ a \neq v \to R\}\\ \boldsymbol{\tau} \in \{\mathbf{rlx},\mathbf{rel},\mathbf{acq},\mathbf{acq\_rel}\} \qquad \sigma \in \{\mathbf{acq},\mathbf{rlx}\}\end{array}}{\{\mathsf{UPD}(\ell,\mathcal{Q}_{\mathtt{rel}},\mathcal{Q}_{\mathtt{acq}}) * P\}\ \mathsf{CAS}_{\boldsymbol{\tau},\boldsymbol{\sigma}}(\ell,v,v')\ \{a.\ a \neq v \land R\}}$$

**Fig. 15.** FSL++ proof rules for atomic locations.