A Program Logic for C11 Memory Fences

Marko Doko and Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

Abstract. We describe a simple, but powerful, program logic for reasoning about C11 relaxed accesses used in conjunction with release and acquire memory fences. Our logic, called fenced separation logic (FSL), extends relaxed separation logic with special modalities for describing state that has to be protected by memory fences. Like its precursor, FSL allows ownership transfer over synchronizations and can be used to verify the message-passing idiom and other similar programs. The soundness of FSL has been established in Coq.

1 Introduction

In order to achieve good performance, modern hardware provides rather weak guarantees on the semantics of concurrent memory accesses. Similarly, to enable as many compiler optimizations as possible, modern "low-level" programming languages (such as C or C++) provide very weak memory models. In this paper we will focus on the memory model defined in the C and C++ standards from 2011 [6, 7] (henceforth, *the C11 memory model*).

The C11 memory model successfully provides a concise abstraction over the various existing hardware implementations, encompassing all the behaviors of widely used hardware platforms. In order not to restrict the set of behaviors exhibited by hardware, the C11 model had to be weaker than any of the hardware models, which makes the guarantees it provides very weak. This pronounced lack of guarantees provided by the C11 model makes it difficult to reason about the model, but those difficulties can be overcome, as evidenced by *Relaxed Separation Logic (RSL)* [16] and *GPS* [14].

Both of these program logics provide a framework for reasoning about the main novel synchronization mechanism provided by the C11 model, namely release and acquire atomic accesses. However, release and acquire accesses are not the only synchronization mechanism that C11 provides. A more advanced mechanism are memory fences, which are not supported by any existing program logics.

In this paper, our goal is to provide simple proof rules for memory fences that can give users of the logic insight and intuition about the behavior of memory fences within the C11 memory model. In order to achieve this goal, we are going to design *Fenced Separation Logic (FSL)* as an extension of RSL. We are choosing to build on top of RSL, because we want our rules for fences to be in the spirit of the RSL rules – clean, simple and intuitive.

We will show that in spite of memory fences having a very global effect on the semantics of C11 programs, we can specify them cleanly by local rules in the style of separation logic.

$$\begin{tikzcd}
\mathrm{a}\colon \mathrm{W}_{\mathrm{rel}} \arrow[r, "\mathsf{rf}"] \arrow[r, dashed, bend left, "\mathsf{sw}"] & \mathrm{b}\colon \mathrm{R}_{\mathrm{acq}}
\end{tikzcd}$$

$\mathsf{isRelease}\,(a) \wedge \mathsf{isAcquire}\,(b) \wedge$
$\mathsf{rf}(a, b)$

$$\begin{tikzcd}
\mathrm{a}\colon \mathrm{W}_{\mathrm{rel}} \arrow[r, "\mathsf{rf}"] \arrow[dr, dashed, "\mathsf{sw}"'] & c \arrow[d, "\mathsf{sb}^{+}"] \\
 & \mathrm{b}\colon \mathrm{F}_{\mathrm{acq}}
\end{tikzcd}$$

$\mathsf{isRelease}\,(a) \wedge \mathsf{isAcqFence}\,(b) \wedge$
$\exists c.\ \mathsf{rf}(a, c) \wedge \mathsf{sb}^{+}(c, b)$

$$\begin{tikzcd}
\mathrm{a}\colon \mathrm{F}_{\mathrm{rel}} \arrow[d, "\mathsf{sb}^{+}"'] \arrow[dr, dashed, "\mathsf{sw}"] & \\
c \arrow[r, "\mathsf{rf}"] & \mathrm{b}\colon \mathrm{R}_{\mathrm{acq}}
\end{tikzcd}$$

$\mathsf{isRelFence}\,(a) \wedge \mathsf{isAcquire}\,(b) \wedge$
$\exists c.\ \mathsf{sb}^{+}(a, c) \wedge \mathsf{rf}(c, b)$

$$\begin{tikzcd}
\mathrm{a}\colon \mathrm{F}_{\mathrm{rel}} \arrow[d, "\mathsf{sb}^{+}"'] \arrow[dr, dashed, "\mathsf{sw}"] & d \arrow[d, "\mathsf{sb}^{+}"] \\
c \arrow[ur, "\mathsf{rf}"] & \mathrm{b}\colon \mathrm{F}_{\mathrm{acq}}
\end{tikzcd}$$

$\mathsf{isRelFence}\,(a) \wedge \mathsf{isAcqFence}\,(b) \wedge$
$\exists c, d.\ \mathsf{sb}^{+}(a, c) \wedge \mathsf{rf}(c, d) \wedge \mathsf{sb}^{+}(d, b)$

Fig. 1. Definition of the synchronizes-with relation. For all actions a and b, a synchronizes with b, written $\mathsf{sw}(a, b)$, if one of the cases above holds.

The remainder of the paper recalls the C11 memory model and RSL (§2), presents the rules of our logic (§3), applies them to a few illustrative examples (§4), and outlines its semantics (§5) and the proof of soundness (§6). The full soundness proof was mechanized in Coq and can be found at `http://plv.mpi-sws.org/fsl/`.

2 Background

2.1 The C11 Memory Model

For the purposes of this paper, we will consider the relevant subset of the C11 memory model. Fuller formalizations of C11 can be found in [3, 15, 16].

The C11 memory model defines the semantics of a program as a set of executions consistent with a certain set of axioms. A program execution is a directed graph whose nodes represent actions (such as memory accesses or fences) and whose edges represent various relations between the actions.

The types of edges that will be of interest in order to prove soundness of FSL are *sequenced-before* and *reads-from* edges.

– *Sequenced-before* (sb) edges describe the program flow. We have an sb edge from a to b if a immediately precedes b in the thread's control flow. Fork actions are considered to immediately precede all of the initial actions of the forked threads, while join actions are considered to be immediate successors of all the joined threads' last actions.
– There is a *reads-from* (rf) edge from a to b if a is the write action that wrote the value read by the read action b.

In the restricted model we are considering, each memory access can be either *non-atomic* or *atomic*, where atomic accesses are further subdivided into *relaxed*, *release*, and *acquire* access types.

Non-atomic accesses should be used as regular accesses throughout the program, whenever we are not implementing a synchronization mechanism. The C11 memory model does not allow data races on non-atomic accesses, and programs with such data races are considered to have undefined semantics.

The intended use of the atomic accesses is to implement synchronization mechanisms. As can be seen from Fig. 1, release writes synchronize with acquire reads, whereas in order to achieve synchronization with relaxed accesses, we need some help from acquire and release fences.

Two additional relations between events are derived from the two relations mentioned above: *synchronizes-with* and *happens-before.*

- *Synchronizes-with* (sw) relates a release event with an acquire event, according to the rules summarized in Fig. 1. Intuitively, synchronization happens when an acquire event can "see" a release event through an rf edge.
- *Happens-before* (hb) is a partial order representing the intuitive notion that one action completed before the other one started. For the subset of the model we are considering, $\mathsf{hb} = (\mathsf{sb} \cup \mathsf{sw})^+$.

2.2 Relaxed Separation Logic

Relaxed Separation Logic (RSL) allows reasoning about the release-acquire fragment of the C11 model. More precisely, using RSL we can reason about ownership transfer that happens when an acquire read reads from a release write. The two most important inference rules in RSL that enable this kind of reasoning are the release write rule (RSL-W-REL) and the acquire read rule (RSL-R-ACQ).

Besides having rules for read and write accesses, RSL also deals with compare-and-swap (CAS) accesses. In this paper, we want to concentrate on dealing with memory fences; so, for the sake of simplicity, we will not model CAS accesses.

Before explaining (RSL-W-REL) and (RSL-R-ACQ), we need to talk about RSL's view of memory locations. Since the C11 model provides us with two classes of accesses (atomic and non-atomic), RSL classifies each location as either atomic or non-atomic, depending on the access mode that will be used throughout the program. For this reason, RSL provides two allocation rules. Rule (A-NA) gives us an uninitialized non-atomic location, while (A-AT) allocates an atomic location by providing us with corresponding release and acquire permissions.

$$\{\mathsf{emp}\}\ \mathbf{alloc}()\ \{\ell.\ \mathsf{Uninit}(\ell)\} \tag{A-NA}$$

$$\{\mathsf{emp}\}\ \mathbf{alloc}()\ \{\ell.\ \mathsf{Rel}(\ell, \mathcal{Q}) * \mathsf{Acq}(\ell, \mathcal{Q})\} \tag{A-AT}$$

The assertions $\mathsf{Rel}(\ell, \mathcal{Q})$ and $\mathsf{Acq}(\ell, \mathcal{Q})$ "attach" a mapping $\mathcal{Q}$ from values to assertions to location ℓ. This mapping should be used to describe the manner in which we intend to use the location ℓ. We can roughly consider $\mathcal{Q}$ to be an invariant stating: "if location ℓ holds value v, then the assertion $\mathcal{Q}(v)$ is true."

We can now proceed with presenting RSL's rules for atomic accesses in which we will further clarify how release and acquire permissions are used. For non-atomic accesses, RSL imports the standard separation logic rules.

Release write RSL's release write rule

$$\big\{\mathsf{Rel}(\ell,\mathcal{Q}) * \mathcal{Q}(v)\big\}\ [\ell]_{\mathrm{rel}} := v\ \big\{\mathsf{Rel}(\ell,\mathcal{Q}) * \mathsf{Init}(\ell)\big\} \tag{RSL-W-REL}$$

says that in order to do a release write of value v to location ℓ, we need to have a permission to do so, $\mathsf{Rel}(\ell,\mathcal{Q})$, and we have to satisfy the invariant specified by that permission, namely $\mathcal{Q}(v)$. After the write is done, we no longer own the resources specified by the invariant (so that readers can obtain them), and we gain the knowledge that the location ℓ has been initialized.

Acquire read The acquire read rule

$$\frac{\forall x.\ \mathsf{precise}(\mathcal{Q}(x))}{\big\{\mathsf{Init}(\ell) * \mathsf{Acq}(\ell,\mathcal{Q})\big\}\ [\ell]_{\mathrm{acq}}\ \big\{v.\ \mathsf{Acq}(\ell,\mathcal{Q}[v{:=}\mathsf{emp}]) * \mathcal{Q}(v)\big\}} \tag{RSL-R-ACQ}$$

complements the release write rule. Here we are able to execute an acquire read of location ℓ if we know that it is initialized, and we have an acquire permission for ℓ. Just as in a release permission, an acquire permission carries a mapping $\mathcal{Q}$ from values to assertions. In case of an acquire permission this mapping describes what resource will be acquired by reading a certain value, so if the value v is read, resource $\mathcal{Q}(v)$ is acquired.

This rule is slightly complicated by two technical details. First, we have to lose the permission to acquire the same ownership by reading v again. Hence, the acquire permission becomes $\mathcal{Q}[v{:=}\mathsf{emp}] \overset{\mathrm{def}}{=} \lambda y.\ \textbf{if}\ y{=}v\ \textbf{then}\ \mathsf{emp}\ \textbf{else}\ \mathcal{Q}(y)$ in the postcondition. Second, we require all the assertions $\mathcal{Q}(x)$ to be precise [10]. This technical detail arises from the nature of the semantics of RSL triples. For details, see [16].

With these two rules, various message passing idioms can be proven correct, as long as we refrain from using relaxed atomic accesses.

Relaxed atomic accesses RSL also provides simple and intuitive rules for dealing with relaxed atomic reads and writes. Since relaxed accesses do not synchronize, we are going to allow relaxed writes only when our permission states that we are releasing an empty resource. Similarly when doing an acquire read, we have to know that the location is initialized, but we will not be acquiring any ownership. The only thing we assert about the value read is that it is one of those that, given the permission structure, could have been written.

$$\frac{\mathcal{Q}(v) = \mathsf{emp}}{\big\{\mathsf{Rel}(\ell,\mathcal{Q})\big\}\ [\ell]_{\mathrm{rlx}} := v\ \big\{\mathsf{Init}(\ell)\big\}} \tag{RSL-W-RLX}$$

$$\big\{\mathsf{Init}(\ell) * \mathsf{Acq}(\ell,\mathcal{Q})\big\}\ [\ell]_{\mathrm{rlx}}\ \big\{v.\ \mathsf{Acq}(\ell,\mathcal{Q}) \wedge (\mathcal{Q}(v) \neq \mathsf{false})\big\} \tag{RSL-R-RLX}$$

Unfortunately, no matter how reasonable the rules seem, both (RSL-W-RLX) and (RSL-R-RLX) are unsound under the standard C11 model, due to the well known problem of the so called thin-air reads [2, 16]. Since this behavior makes the standard C11 relaxed atomics unusable, when dealing with relaxed accesses RSL assumes a certain strengthening of the C11 model. Namely, we consider the union of the happens-before and reads-from relations to be acyclic. Since this strengthening was necessary even for the simple rules above, we consider it justifiable to work under the same strengthened model when developing a logic that will enable us to do ownership transfer using relaxed accesses together with memory fences.

3 Fenced Separation Logic (FSL)

Our goal is to enhance the rules from the previous section to allow ownership transfer through relaxed accesses. Ideally, we would like the rules for release writes and acquire reads to remain unchanged, so that all valid RSL proofs remain valid FSL proofs. We will therefore need to change the rules for dealing with relaxed accesses, and we will need to come up with rules for memory fences.

Looking back at Fig. 1, we can see how the rules (RSL-W-REL) and (RSL-R-ACQ) describe the simple situation in the top left corner. Imagine how the resource released by the read travels along the rf-edge in order to appear as acquired resource after the acquire read has been done.

Note also how the other three situations look like decompositions of the simplest one. The release write has been split into a release fence after which comes a relaxed write, and the acquire read has been split into a relaxed read after which comes an acquire fence. This is more than just a good intuition: release writes and acquire reads can, in fact, be implemented in this fashion.

Now, the question is, can we extend our mental image of a released resource "traveling" over the rf-edge to the point of it being acquired? Well, in the simple case, we were lucky, since the rf-edge coincided with the synchronization arrow, but in other cases the situation is a bit more complicated. We would like for the resource to disappear from the release point, and reappear at the acquire point, but the problem arises from the fact that the only point of transfer possible is the rf-edge.

A solution to the above mentioned conundrum is, in the presence of fences and relaxed accesses, to think of resource transfer in the following way. If we are releasing a resource by a combination of a release fence and a relaxed write, at the fence we should decide what is going to be released, and not touch that resource until we finally send it away by doing the write. Conversely, when acquiring a resource using a pair of a relaxed read and an acquire fence, once we do the read, we know which resources we are going to get, but we will not be able to use those resources until we reach a synchronization point marked by the acquire fence.

In order to formally represent our intuition, we will introduce two modalities into RSL's language of assertions. We will mark the resources that have been prepared to be released by $\triangle$, while $\triangledown$ will mark resources that are waiting for synchronization. Therefore, we define FSL assertions by the following grammar:

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

FSL judgments follow the form of RSL judgments $\{P\}\, E\, \{y.Q\}$, where y binds the return value of the expression E.

FSL supports all the standard rules of Hoare and separation logic, just as RSL does. Non-atomics and allocation are also treated exactly as in RSL. Furthermore, FSL support RSL's program relaxation rule. Namely, if we define relaxation of a program E to be a program E' identical to E except that E' can have weaker atomic accesses according to the following partial order: rlx $\sqsubseteq$ rel $\sqsubseteq$ sc, rlx $\sqsubseteq$ acq $\sqsubseteq$ sc; the following rule is sound:

$$\frac{\{P\}\, E'\, \{y.\ Q\} \qquad E' \sqsubseteq E}{\{P\}\, E\, \{y.\ Q\}}\,. \tag{RELAX}$$

Which RSL rules saw changes happen to them? As expected, rules for atomic accesses.

Atomic writes First we take a look at the rule for release writes. It got a small cosmetic change:

$$\frac{\mathsf{normalizable}(\mathcal{Q}(v))}{\{\mathsf{Rel}(\ell, \mathcal{Q}) * \mathcal{Q}(v)\}\, [\ell]_{\text{rel}} := v\, \{\mathsf{Rel}(\ell, \mathcal{Q}) * \mathsf{Init}(\ell)\}}\,. \tag{W-REL}$$

We will define the notion of normalizability in Section 5.1. For now, the important thing to note is that any RSL assertion (i.e., any FSL formula in which modalities $\triangle$ and $\triangledown$ do not appear) is normalizable. For this reason, our stated goal to keep RSL proofs to be valid FSL proofs has not been compromised.

The rule for relaxed writes is almost exactly the same as (W-REL).

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

All the main ingredients are the same: we have to have a release permission, and we have to own the resource specified by the permission. The only additional requirement is that the resource has to be under the modality stating that it can be released by a relaxed write. As we will later see this ensures that our write is placed after a release fence.

Atomic reads The acquire read rule got changed in the same vein as the release write rule, by adding a normalizability requirement:

$$\frac{\forall x.\ \mathsf{precise}(\mathcal{Q}(x)) \land \mathsf{normalizable}(\mathcal{Q}(x))}{\{\mathsf{Init}(\ell) * \mathsf{Acq}(\ell, \mathcal{Q})\}\, [\ell]_{\text{acq}}\, \{v.\ \mathsf{Acq}(\ell, \mathcal{Q}[v{:=}\mathsf{emp}]) * \mathcal{Q}(v)\}}\,. \tag{R-ACQ}$$

Just as it was the case for writes, the rule for relaxed reads differs only in a single modality appearance:

$$\frac{\forall x.\ \mathsf{precise}(\mathcal{Q}(x)) \wedge \mathsf{normalizable}(\mathcal{Q}(x))}{\big\{\mathsf{Init}(\ell) * \mathsf{Acq}(\ell,\mathcal{Q})\big\}\ [\ell]_{\mathrm{rlx}}\ \big\{v.\ \mathsf{Acq}(\ell,\mathcal{Q}[v{:=}\mathsf{emp}]) * \triangledown\mathcal{Q}(v)\big\}}\ . \qquad \text{(R-RLX)}$$

In order to do an atomic read, we have to have an acquire permission, and to know that the location has been initialized. After an acquire read, we gain ownership of the resource described by the permission. In the case of a relaxed read, we get the same resource, but under the $\triangledown$ modality. This, as we will see below, makes the resource unusable before we reach an acquire fence.

Fences Finally, let us turn our attention to the two rules describing the actions of memory fences. These are by far the simplest rules, as the only job of fences is to manage the two modalities. This is achieved by the following two rules:

$$\frac{\mathsf{normalizable}(P)}{\big\{P\big\}\ \mathrm{FENCE}_{\mathrm{rel}}\ \big\{\triangle P\big\}} \quad \text{(F-REL)} \qquad\qquad \big\{\triangledown P\big\}\ \mathrm{FENCE}_{\mathrm{acq}}\ \big\{P\big\} \quad \text{(F-ACQ)}$$

Release fences protect resources that are to be released by putting them under the $\triangle$ modality, while acquire fences clear the $\triangledown$ modality making resources under it usable.

A note about normalizability Even though the normalizability requirement that appears in several places in atomic accesses and fence rules looks pretty cumbersome, we can easily see that is not the case. The requirement is formally necessary, but a user of the logic, when doing proofs, will never even notice it. In practice, the only resources we want to transfer are the ones described by formulas containing no modalities, which are always normalizable. Because of this, the only situation where we are forced to think of normalizability is when proving the inference rules sound, and not when proving programs correct.

4 Examples

Let us first take a look at the message passing example in Fig. 2. In this example, the middle thread initializes two non-atomic variables (a and b), and then uses two atomic variables (x and y) to signal to consumer threads that the resources are ready. The consumer threads then proceed to increment the variables a and b.

The middle thread uses only one release fence in order to transfer ownership using two different relaxed writes. In order to be able to verify such idioms, it is necessary for our modalities to distribute over separating conjunction (see Section 5.1).

The most interesting part of the proof in the consumer threads is the waiting loop. Let us have a more detailed look at how we derive the triple

$$\big\{\mathsf{Acq}(x,\mathcal{P}) * \mathsf{Init}(x)\big\}\ \mathbf{while}([x]_{\mathrm{rlx}} == 0);\ \big\{\mathsf{true} * \triangledown a \mapsto 42\big\}\ .$$

$$\textbf{Let } \mathcal{P} \overset{\text{def}}{=} \lambda v.\ \textbf{if } v = 0 \textbf{ then } \mathsf{emp} \textbf{ else } a \mapsto 42\,,$$
$$\textbf{and } \mathcal{Q} \overset{\text{def}}{=} \lambda v.\ \textbf{if } v = 0 \textbf{ then } \mathsf{emp} \textbf{ else } b \mapsto 7\,.$$

$$\begin{array}{c}
\{\mathsf{emp}\} \\
a := \textbf{alloc}(); b := \textbf{alloc}(); \\
\{\mathsf{Uninit}(a) * \mathsf{Uninit}(b)\} \\
x := \textbf{alloc}(); y := \textbf{alloc}(); \\
\{\mathsf{Uninit}(a) * \mathsf{Uninit}(b) * \mathsf{Rel}(x, \mathcal{P}) * \mathsf{Acq}(x, \mathcal{P}) * \mathsf{Rel}(y, \mathcal{Q}) * \mathsf{Acq}(y, \mathcal{Q})\} \\
[x]_{\text{rlx}} := 0; [x]_{\text{rlx}} := 0 \\
\{\mathsf{Uninit}(a) * \mathsf{Uninit}(b) * \mathsf{Rel}(x, \mathcal{P}) * \mathsf{Acq}(x, \mathcal{P}) * \mathsf{Rel}(y, \mathcal{Q}) * \mathsf{Acq}(y, \mathcal{Q}) * \mathsf{Init}(x) * \mathsf{Init}(y)\}
\end{array}$$

$$\begin{array}{l||l||l}
 & \{\mathsf{Uninit}(a) * \mathsf{Uninit}(b) * \mathsf{Rel}(x, \mathcal{P}) * \mathsf{Rel}(y, \mathcal{Q})\} & \\
 & [a]_{\text{na}} := 42; [b]_{\text{na}} := 7; & \\
\{\mathsf{Acq}(x, \mathcal{P}) * \mathsf{Init}(x)\} & \{a \mapsto 42 * b \mapsto 7 * \mathsf{Rel}(x, \mathcal{P}) * \mathsf{Rel}(y, \mathcal{Q})\} & \{\mathsf{Acq}(y, \mathcal{Q}) * \mathsf{Init}(y)\} \\
\textbf{while}([x]_{\text{rlx}} == 0); & \text{FENCE}_{\text{rel}}; & \textbf{while}([y]_{\text{rlx}} == 0); \\
\{\mathsf{true} * \nabla a \mapsto 42\} & \{\triangle(a \mapsto 42 * b \mapsto 7) * \mathsf{Rel}(x, \mathcal{P}) * \mathsf{Rel}(y, \mathcal{Q})\} & \{\mathsf{true} * \nabla b \mapsto 7\} \\
\text{FENCE}_{\text{acq}}; & \{\triangle a \mapsto 42 * \triangle b \mapsto 7 * \mathsf{Rel}(x, \mathcal{P}) * \mathsf{Rel}(y, \mathcal{Q})\} & \text{FENCE}_{\text{acq}}; \\
\{\mathsf{true} * a \mapsto 42\} & [x]_{\text{rlx}} := 1; & \{\mathsf{true} * b \mapsto 7\} \\
[a]_{\text{na}} := [a]_{\text{na}} + 1; & \{\mathsf{Init}(x) * \triangle b \mapsto 7 * \mathsf{Rel}(y, \mathcal{Q})\} & [b]_{\text{na}} := [b]_{\text{na}} + 1; \\
\{\mathsf{true} * a \mapsto 43\} & [y]_{\text{rlx}} := 1; & \{\mathsf{true} * b \mapsto 8\} \\
 & \{\mathsf{Init}(x) * \mathsf{Init}(y)\} &
\end{array}$$

$$\{a \mapsto 43 * b \mapsto 8 * \mathit{true}\}$$

Fig. 2. Double message passing example.

We first use the equivalence $\mathsf{Init}(x) \iff \mathsf{Init}(x) * \mathsf{Init}(x)$, and apply (R-RLX) (together with the frame rule) inside the loop:

$$\{\mathsf{Acq}(x, \mathcal{P}) * \mathsf{Init}(x) * \mathsf{Init}(x)\}\ [x]_{\text{rlx}}\ \{v.\ \mathsf{Acq}(x, \mathcal{P}[v{:=}\mathsf{emp}]) * \nabla\mathcal{P}(v) * \mathsf{Init}(x)\}\ .$$

It is important to note that $\mathcal{P}(0) = \mathsf{emp}$, and consequently $\mathcal{P}[0{:=}\mathsf{emp}] = \mathcal{P}$. Therefore, as long as we stay in the loop (i.e., the value being read is 0), our postcondition reads $\{\mathsf{Acq}(x, \mathcal{P}) * \nabla\mathsf{emp} * \mathsf{Init}(x)\}$. Since we have $\nabla\mathsf{emp} \iff \mathsf{emp}$, this is equivalent to $\{\mathsf{Acq}(x, \mathcal{P}) * \mathsf{Init}(x) * \mathsf{Init}(x)\}$. With this, the loop invariant has been established.

Once we are out of the loop, we know that the value we read from x is not 0. Therefore, we have

$$\begin{array}{c}
\{\mathsf{Acq}(x, \mathcal{P}) * \mathsf{Init}(x)\} \\
\textbf{while}([x]_{\text{rlx}} == 0); \\
\{v.\ v \neq 0 \land \mathsf{Acq}(x, \mathcal{P}[v{:=}\mathsf{emp}]) * \nabla\mathcal{P}(v) * \mathsf{Init}(x)\}\,.
\end{array}$$

We can transform the postcondition into $\{v.\ v \neq 0 \land \mathsf{true} * \nabla\mathcal{P}(v)\}$, using the consequence rule. Expanding the definition of $\mathcal{P}$, and using the fact that $v \neq 0$, the postcondition becomes

$$\{v.\ v \neq 0 \land \mathsf{true} * \nabla a \mapsto 42\}\ .$$

We can now use the consequence rule to transform the loop postcondition into

$$\{\mathsf{true} * \nabla a \mapsto 42\}\ .$$

$$\textbf{Let } \mathcal{Q} \overset{\text{def}}{=} \lambda v.\ \textbf{if } v = 0 \textbf{ then } \mathsf{emp} \textbf{ else } a \mapsto 42\,.$$

$$\begin{array}{l||l||l}
\{a \mapsto 0 * \mathsf{Rel}(x, \mathcal{Q})\} & & \{\mathsf{Acq}(y, \mathcal{Q}) * \mathsf{Init}(y)\} \\
[a]_{\text{na}} := 42; & \{\mathsf{Acq}(x, \mathcal{Q}) * \mathsf{Init}(x)\} & \textbf{while}([y]_{\text{rlx}} == 0); \\
\{a \mapsto 42 * \mathsf{Rel}(x, \mathcal{Q})\} & \textbf{while}([x]_{\text{rlx}} == 0); & \{\Diamond a \mapsto 42 * \mathsf{true}\} \\
\text{FENCE}_{\text{rel}}; & \{\Diamond a \mapsto 42 * \mathsf{true}\} & \text{FENCE}_{\text{acq}}; \\
\{\Diamond a \mapsto 42 * \mathsf{Rel}(x, \mathcal{Q})\} & [y]_{\text{rlx}} := 1; & \{a \mapsto 42 * \mathsf{true}\} \\
[x]_{\text{rlx}} := 1; & \{\mathsf{true}\} & [a]_{\text{na}} := [a]_{\text{na}} + 1; \\
\{\mathsf{true}\} & & \{a \mapsto 43 * \mathsf{true}\}
\end{array}$$

Fig. 3. Example showing that merging the two modalities into one is unsound.

Note how the consumer threads encounter a fence only once, when the resource they have been waiting for has been made ready. On architectures such as PowerPC and ARM, this way of implementing waiting loops gives us performance benefit over doing an acquire read in the loop. The difference comes from the fact that those architectures require placing a hardware fence instruction in order to implement acquire reads, while relaxed reads can be implemented by plain read instructions. This shows that the ability to reason about memory fences enables verification of an important class of programs.

Our next example, given in Fig. 3, shows that we could not have designed our logic to have only one modality. In the example, we assume that all our inference rules use only one modality $\Diamond$ in all the places where $\triangle$ or $\triangledown$ are used. Before the fork, x and y have been allocated as atomic variables, and a has been allocated as non-atomic. All variables have been initialized to 0. As we can see, using the rules with only one modality, we can verify the ownership transfer from the left thread to the right thread via the middle thread.

The problem here is that, since there is no rf communication between the threads on the left and right, no synchronization can happen between them, which means that the two accesses of the non-atomic location a are racing. According to the C11 model, this program has undefined semantics, and we should not have been able to verify it.

The problem lies in the middle thread. After we use the (R-RLX) rule to get the protected resource $\Diamond a \mapsto 42$, we can now immediately use the (W-RLX) rule to send that resource away to be acquired in the right thread, and subsequently used in a racy manner. In order to avoid this behavior, we have to make resources produced by the (R-RLX) rule unusable by the (W-RLX) rule, and that is exactly what has been done by introducing two different modalities.

5 Semantics

5.1 Semantics of Assertions

We first briefly describe the interpretation of the Rel and Acq assertions. In order to model release and acquire permissions, we store equivalence classes of assertions (Assn), modulo an equivalence relation $\sim$ (e.g., treating conjunctions up to commutativity and associativity). For our purposes, we can take the relation used in RSL, extending it with the requirement that for any two assertions P and Q, if $P \sim Q$ holds, then $\triangle P \sim \triangle Q$ and $\triangledown P \sim \triangledown Q$ also have to hold.

The greatest challenge in defining the semantics of FSL assertions is interpreting the modalities, especially when applied to the Acq and Rel assertions. The obvious idea is to add a single label to each location, and use that label to track its "protection status" (is it under a modality and which one). This solution works perfectly for dealing with locations that are being accessed non-atomically. For atomic locations, the situation is more complicated because of the three splitting rules inherited from RSL, which we also have to support in FSL.

$$\begin{aligned}
\mathsf{Init}(\ell) &\iff \mathsf{Init}(\ell) * \mathsf{Init}(\ell) && (\textsc{init-split})\\
\mathsf{Rel}(\ell, \mathcal{Q}_1) * \mathsf{Rel}(\ell, \mathcal{Q}_2) &\iff \mathsf{Rel}(\ell, \lambda v.\ \mathcal{Q}_1(v) \vee \mathcal{Q}_2(v)) && (\textsc{rel-split})\\
\mathsf{Acq}(\ell, \mathcal{Q}_1) * \mathsf{Acq}(\ell, \mathcal{Q}_2) &\iff \mathsf{Acq}(\ell, \lambda v.\ \mathcal{Q}_1(v) * \mathcal{Q}_2(v)) && (\textsc{acq-split})
\end{aligned}$$

When thinking of modalities, we are thinking about them protecting locations from being tampered with after being prepared for being released by a release fence, or from being prematurely accessed while waiting on a synchronizing acquire fence. We can think of release permissions as giving us the right to write to a location and initializations as giving us the right to read the location. Therefore, for each atomic location we should keep separate labels for release permissions and initialization. Keeping labels for acquire permissions is not necessary, because acquire permissions are always used in conjunction with initializations.

The model of heaps is as follows:

$$\begin{aligned}
\mathsf{Lab} &\stackrel{\text{def}}{=} \{\circ, \triangle, \triangledown\},\\
\mathcal{M} &\stackrel{\text{def}}{=} \mathsf{Val} \to \mathsf{Assn}/\!\sim,\\
\mathsf{Heap}_{\text{spec}} &\stackrel{\text{def}}{=} \mathsf{Loc} \rightharpoonup \mathsf{NA}[(\mathbb{U} + \mathsf{Val}) \times \mathsf{Lab}] + \mathsf{Atom}[\mathcal{M} \times \mathcal{M} \times \mathbb{P}(\mathsf{Lab}) \times \mathbb{P}(\mathsf{Lab})].
\end{aligned}$$

Locations can be either non-atomic or atomic. Non-atomic locations have a label and are either uninitialized (denoted by the symbol $\mathbb{U}$) or contain a value. Atomic locations contain two permission mappings, representing the release and acquire permissions, and two sets of labels, for keeping track of protection status of the release permission and of the location's initialization. We need to keep sets of labels in order to give meaning to assertions such as $\mathsf{Init}(\ell) * \triangle\mathsf{Init}(\ell)$. Heap composition $\oplus$ is defined as follows:

$$h_1 \oplus' h_2 \stackrel{\text{def}}{=} \lambda\ell. \begin{cases} h_1(\ell) & \text{if } \ell \in \mathsf{dom}(h_1) \setminus \mathsf{dom}(h_2)\\ h_2(\ell) & \text{if } \ell \in \mathsf{dom}(h_2) \setminus \mathsf{dom}(h_1)\\ \mathsf{Atom}[\lambda v.\ \mathcal{P}_1(v) \vee \mathcal{P}_2(v), & \\ \quad \lambda v.\ \mathcal{Q}_1(v) * \mathcal{Q}_2(v), \Lambda_1 \cup \Lambda_2, \Gamma_1 \cup \Gamma_2] & \text{if } h_i(\ell) = \mathsf{Atom}[\mathcal{P}_i, \mathcal{Q}_i, \Lambda_i, \Gamma_i] \\ & \quad \text{for } i = 1, 2\\ \text{undef} & \text{otherwise} \end{cases}$$

$$h_1 \oplus h_2 \stackrel{\text{def}}{=} \begin{cases} h_1 \oplus' h_2 & \text{if } \mathsf{dom}(h_1 \oplus' h_2) = \mathsf{dom}(h_1) \cup \mathsf{dom}(h_2)\\ \text{undef} & \text{otherwise} \end{cases}$$

In order to define the semantics of assertions, we need to define two more notions.

Definition 1 (Heap similarity). *Heaps* h_1 *and* h_2 *are* similar *(*$h_1 \approx h_2$*) if* $\mathsf{dom}(h_1) = \mathsf{dom}(h_2)$ *and for all locations* ℓ*,* $h_1(\ell) \simeq h_2(\ell)$*, where* $\simeq$ *is defined as follows:*

$$\begin{aligned}
\mathsf{NA}[x, \lambda] \simeq \mathsf{NA}[x', \lambda'] &\overset{\text{def}}{\iff} x = x'\,, \\
\mathsf{Atom}[\mathcal{P}, \mathcal{Q}, \Lambda, \Gamma] \simeq \mathsf{Atom}[\mathcal{P}', \mathcal{Q}', \Lambda', \Gamma'] &\overset{\text{def}}{\iff} \begin{array}{l} \mathcal{P} = \mathcal{P}' \wedge \mathcal{Q} = \mathcal{Q}' \wedge \\ (\Lambda = \emptyset \iff \Lambda' = \emptyset) \\ (\Gamma = \emptyset \iff \Gamma' = \emptyset)\,. \end{array}
\end{aligned}$$

Definition 2 (Exact label). *Heap* h *is* exactly labeled *by* $\lambda \in \mathsf{Lab}$ *(notation:* $\text{labeled}(h, \lambda)$*) if, for all locations* ℓ*,* $h(\ell) = \mathsf{NA}[x, \gamma] \Rightarrow \gamma = \lambda$*, and* $h(\ell) = \mathsf{Atom}[\mathcal{P}, \mathcal{Q}, \Lambda, \Gamma] \Rightarrow \Lambda \setminus \{\lambda\} = \Gamma \setminus \{\lambda\} = \emptyset$*.*

In short, two heaps are similar when they only differ in labels which appear in the heap, and a heap is exactly labeled by a label if that's the only label appearing in the heap.

We are now ready to define semantics of FSL assertions.

Definition 3 (Assertion semantics). *Let* $[\![-]\!] : \mathsf{Assn} \to \mathbb{P}(\mathsf{Heap}_{\text{spec}})$ *be:*

$$\begin{array}{rcl rcl}
[\![\mathsf{false}]\!] &\overset{\text{def}}{=}& \emptyset & [\![P * Q]\!] &\overset{\text{def}}{=}& \{h_1 \oplus h_2 \mid h_1 \in [\![P]\!] \wedge h_2 \in [\![Q]\!]\} \\
[\![\mathsf{emp}]\!] &\overset{\text{def}}{=}& \{\emptyset\} & [\![P \to Q]\!] &\overset{\text{def}}{=}& \{h \mid h \in [\![P]\!] \implies h \in [\![Q]\!]\} \\
[\![\forall x.\ P]\!] &\overset{\text{def}}{=}& \{h \mid \forall v.\ h \in [\![P[v/x]]\!]\} & [\![\mathsf{Init}(\ell)]\!] &\overset{\text{def}}{=}& \{\{\ell \mapsto \mathsf{Atom}[\mathsf{False}, \mathsf{Emp}, \emptyset, \{\circ\}]\}\} \\
[\![\mathsf{Uninit}(\ell)]\!] &\overset{\text{def}}{=}& \{\{\ell \mapsto \mathsf{NA}[\mathbb{U}, \circ]\}\} & [\![\mathsf{Rel}(\ell, \mathcal{Q})]\!] &\overset{\text{def}}{=}& \{\{\ell \mapsto \mathsf{Atom}[\mathcal{Q}, \mathsf{Emp}, \{\circ\}, \emptyset]\}\} \\
[\![\ell \mapsto v]\!] &\overset{\text{def}}{=}& \{\{\ell \mapsto \mathsf{NA}[v, \circ]\}\} & [\![\mathsf{Acq}(\ell, \mathcal{Q})]\!] &\overset{\text{def}}{=}& \{\{\ell \mapsto \mathsf{Atom}[\mathsf{False}, \mathcal{Q}, \emptyset, \emptyset]\}\}
\end{array}$$

$$\begin{array}{rcl}
[\![\triangle P]\!] &\overset{\text{def}}{=}& \{h \mid \text{labeled}(h, \triangle) \wedge \exists h' \in [\![P]\!].\ h \approx h' \wedge \text{labeled}(h', \circ)\} \\
[\![\triangledown P]\!] &\overset{\text{def}}{=}& \{h \mid \text{labeled}(h, \triangledown) \wedge \exists h' \in [\![P]\!].\ h \approx h' \wedge \text{labeled}(h', \circ)\}
\end{array}$$

With this definition, we can prove the splitting rules sound and that the modalities distribute over conjunction, disjunction, and separating conjunction.

Lemma 1. *The properties* (INIT-SPLIT)*,* (REL-SPLIT)*,* (ACQ-SPLIT) *hold universally, as well as the following equivalences:*

$$\begin{array}{cc}
\triangle(P \wedge Q) \iff \triangle P \wedge \triangle Q & \triangledown(P \wedge Q) \iff \triangledown P \wedge \triangledown Q \\
\triangle(P \vee Q) \iff \triangle P \vee \triangle Q & \triangledown(P \vee Q) \iff \triangledown P \vee \triangledown Q \\
\triangle(P * Q) \iff \triangle P * \triangle Q & \triangledown(P * Q) \iff \triangledown P * \triangledown Q
\end{array}$$

As conditions in several inference rules presented in Section 3, we encountered precision and normalizability. The definition of precision is standard [10], and normalizability means that if an assertion is satisfied by some heap, then it is also satisfied by some subheap exactly labeled by $\circ$.

Definition 4 (Normalizability). *An assertion* P *is* normalizable *if for all* $h \in [\![P]\!]$*, there exist* $h_\circ$ *and* h' *such that* $h = h_\circ \oplus h'$*,* $h_\circ \in [\![P]\!]$*, and* $\text{labeled}(h_\circ, \circ)$*.*

5.2 The Semantics of Triples

The semantics of FSL triples closely follows that of RSL triples. To define the semantics of a triple $\{P\}\ E\ \{y.Q\}$, we annotate edges of executions of E, put into an arbitrary context, with the restriction that each execution of E should have a unique incoming sb-edge and a unique outgoing sb-edge from/to its context. These edges will be responsible for carrying heaps satisfying precondition P and postcondition Q.

Triple semantics is defined in terms of *annotation validity*. In short, validity states that the sum of heaps on all incoming edges of a node is equal to the sum of heaps on all outgoing edges, modulo the effect of the node's action (e.g., allocation will produce a new heap cell).

Figure 4 showcases the most important parts of FSL's validity definition, namely the conditions for nodes corresponding to atomic accesses and fences. In the figure, *hmap* is the function that annotates edges of the execution with heaps, and $\mathsf{SB}_{\mathsf{in}}(a)$, $\mathsf{SB}_{\mathsf{out}}(a)$, $\mathsf{RF}_{\mathsf{in}}(a)$, and $\mathsf{RF}_{\mathsf{out}}(a)$ denote sets of incoming sb-edges, outgoing sb-edges, incoming rf-edges, and outgoing rf-edges of node a, respectively. We also extend each execution by adding a special *sink node*. Each node a of the execution is connected to the sink node by an edge which we denote by $\mathsf{Sink}(a)$.

The main idea of the validity conditions is to have heaps satisfying preconditions of inference rules on the incoming sb-edges, while heaps satisfying postconditions go on the outgoing sb-edges. If there is some ownership transfer, we will put the resources being transferred on rf-edges.

Let us now take a closer look at the validity conditions presented in Fig. 4.

The first line of the validity condition for the $W_Z(\ell, v)$ action establishes that the outgoing sb-edge contains everything there was on the incoming edge, except h_r, the resources released by the write. In addition, the outgoing sb-edge can contain the knowledge that the location ℓ has been initialized. The second line says that the outgoing sb-edge (and by the first line, also the incoming sb-edge) contains release permission $\mathsf{Rel}(\ell, \mathcal{Q})$. Note that the release permission label contains $\circ$. We need this to ensure that only resources labeled by $\circ$ are being accessed. We often refer to resources labeled by $\circ$ as *normal resources*. For the same reason, we force the initialization label on the outgoing sb-edge to contain $\circ$.

The next two lines ensure that the resource being released (h_r) is, in fact, described by $\mathcal{Q}(v)$, as stated by the release permission. The idea is for the resources that have been acquired by some read to go on the corresponding rf-edges, and those that have been released, but not (yet) acquired to be annotated on the sink edge. Note how we require resources that have been released to be labeled only by $\triangledown$. This is to mark them as unusable until a synchronization point is reached. The last line states that only release writes can release normal resources. Relaxed writes can only release resources marked by $\vartriangle$, which means that those resources have been protected by a release fence.

The first line of the validity condition for $R_Z(\ell, v)$ states that the incoming sb edge has to contain initialization information for ℓ (labeled with $\circ$), together with the acquire permission $\mathsf{Acq}(\ell, \mathcal{Q})$. We lose the permission to acquire more

– If $a = W_Z(\ell, v)$ and $Z \neq \mathrm{na}$, then there exist h_r, h_F, $\mathcal{Q}$ such that

$$\begin{array}{l}
hmap(\mathsf{SB}_{\mathsf{in}}(a)) \oplus \{\ell \mapsto \mathsf{Atom}[\mathsf{False}, \mathsf{Emp}, \emptyset, \{\circ\}]\} = hmap(\mathsf{SB}_{\mathsf{out}}(a)) \oplus h_r \ \wedge \\
hmap(\mathsf{SB}_{\mathsf{out}}(a)) = \{\ell \mapsto \mathsf{Atom}[\mathcal{Q}, \mathsf{Emp}, \{\circ\}, \{\circ\}]\} \oplus h_F \ \wedge \\
h_r \approx hmap(\mathsf{RF}_{\mathsf{out}}(a)) \oplus hmap(\mathsf{Sink}(a)) \ \wedge \\
hmap(\mathsf{RF}_{\mathsf{out}}(a)) \oplus hmap(\mathsf{Sink}(a)) \in [\![\triangledown \mathcal{Q}(v)]\!] \ \wedge \\
(\mathrm{labeled}(h_r, \triangle) \vee (Z \neq \mathrm{rlx} \wedge \mathrm{labeled}(h_r, \circ)))\,,
\end{array}$$

– If $a = R_Z(\ell, v)$ and $Z \neq \mathrm{na}$, then there exist h_r, h_F, $\mathcal{Q}$ such that

$$\begin{array}{l}
hmap(\mathsf{SB}_{\mathsf{in}}(a)) = \{\ell \mapsto \mathsf{Atom}[\mathsf{False}, \mathcal{Q}, \emptyset, \{\circ\}]\} \oplus h_F \ \wedge \\
hmap(\mathsf{Sink}(a)) = \{\ell \mapsto \mathsf{Atom}[\mathsf{False}, \mathsf{Emp}[v := \mathcal{Q}(v)], \emptyset, \emptyset]\} \ \wedge \\
hmap(\mathsf{RF}_{\mathsf{in}}(a)) \in [\![\triangledown \mathcal{Q}(v)]\!] \wedge h_r \approx hmap(\mathsf{RF}_{\mathsf{in}}(a)) \in [\![\triangledown \mathcal{Q}(v)]\!] \ \wedge \\
hmap(\mathsf{SB}_{\mathsf{out}}(a)) = \{\ell \mapsto \mathsf{Atom}[\mathsf{False}, \mathcal{Q}[v := \mathsf{emp}], \emptyset, \{\circ\}]\} \oplus h_r \oplus h_F \ \wedge \\
\mathsf{precise}(\mathcal{Q}(v)) \wedge \mathsf{normalizable}(\mathcal{Q}(v)) \ \wedge \\
(\mathrm{labeled}(h_r, \triangledown) \vee (Z \neq \mathrm{rlx} \wedge \mathrm{labeled}(h_r, \circ)))
\end{array}$$

– If $a = \textsc{fence}_Z$, then there exist h_{rel}, h'_{rel}, h_{acq}, h'_{acq}, h_F, such that

$$\begin{array}{l}
hmap(\mathsf{SB}_{\mathsf{in}}(a) = h_{\mathrm{rel}} \oplus h_{\mathrm{acq}} \oplus h_F \wedge hmap(\mathsf{SB}_{\mathsf{out}}(a) = h'_{\mathrm{rel}} \oplus h'_{\mathrm{acq}} \oplus h_F \ \wedge \\
h_{\mathrm{rel}} \approx h'_{\mathrm{rel}} \wedge \mathrm{labeled}(h_{\mathrm{rel}}, \circ) \wedge \mathrm{labeled}(h'_{\mathrm{rel}}, \triangle) \ \wedge \\
h_{\mathrm{acq}} \approx h'_{\mathrm{acq}} \wedge \mathrm{labeled}(h_{\mathrm{acq}}, \triangledown) \wedge \mathrm{labeled}(h'_{\mathrm{acq}}, \circ) \ \wedge \\
Z = \mathrm{rel} \rightarrow h_{\mathrm{acq}} = \emptyset \wedge Z = \mathrm{acq} \rightarrow h_{\mathrm{rel}} = \emptyset\,.
\end{array}$$

where $\mathsf{False} \overset{\mathrm{def}}{=} \lambda v.\, \mathsf{false}$, $\quad \mathsf{Emp} \overset{\mathrm{def}}{=} \lambda v.\, \mathsf{emp}$,
and $f[s := t] \overset{\mathrm{def}}{=} \lambda x.\, \textbf{if}\ x = s\ \textbf{then}\ t\ \textbf{else}\ f(x)$.

Fig. 4. Validity conditions for atomic accesses and fences.

ownership by reading the same value, and that lost permission gets placed on the sink edge (line 2). Line 3 states that the resources acquired via the rf-edge are exactly those described by the acquire permission. The contents of the incoming sb-edge (without the lost permission), together with the resources acquired via the rf-edge, are to be placed on the sb-edge (line 4). Line 5 states the technical requirements of precision and normalizability.

The last line serves a purpose analogous to the last line in the validity of writes. Only acquire reads can make acquired resources immediately usable (by changing their label from $\triangledown$ to $\circ$). Acquire reads can do this because they serve as synchronization points. Relaxed reads have to leave the $\triangledown$ label on the acquired resources, which will force us to wait for a synchronization point provided by an acquire fence before we will be able to use those resources.

Note 1 (Sink edges). Using sink edges to keep track of lost permissions and resources that have been released but that nobody has acquired may seem like an unnecessary complication. Why do we not just forget about them? The reason for introducing sink edges is pragmatism. Keeping track of those "lost" annotations greatly simplifies the soundness proofs of (R-ACQ) and (R-RLX) rules because it makes Lemma 2 in Section 6 more widely applicable.

The validity conditions for fences are fairly straightforward. Release fences take some normal resource and protect it by setting its labels to $\triangle$, while acquire fences make resources usable by changing labels from $\triangledown$ to $\circ$.

Now that we have defined annotation validity, we can proceed to discuss the semantics of triples.

Somewhat simplified, in RSL the triple $\{P\}\ E\ \{y.Q\}$ holds if all executions of E satisfying the precondition P (i.e. the unique incoming sb-edge to the execution of E is annotated by a heap satisfying $P * R$, where R is some frame) can be validly annotated such that the annotation of the unique outgoing sb-edge from the execution of E satisfies $Q * R$.

In FSL, the triple semantics differs from the one in RSL in one minor detail. Whereas in RSL the heap annotating the outgoing sb-edge should satisfy $Q * R$, here we allow it to be bigger: it suffices that the heap can be split into the sum of two heaps $h \oplus h'$ such that $h \in [\![Q * R]\!]$, while h' can be arbitrary. The reason for this change will be discussed in the next section.

6 Soundness

In this section, we illustrate some key points in the soundness proof of FSL. Of particular interest are the places where the proof structure deviates from that of RSL. The full soundess proof can be found at `http://plv.mpi-sws.org/fsl/`.

When talking about the soundness of a program logic like FSL, it is, of course, necessary to prove the inference rules valid according to the semantics of triples, but we also want to say that if $\{P\}\ E\ \{Q\}$ holds, then the program E satisfies some useful properties. The properties of interest here are race-freedom, memory safety and absence of reads of uninitialized locations. In the definitions below, we list formal statements of these three properties.

Definition 5 (Conflicting accesses). *Two actions are* conflicting *if both of them are accesses (i.e. reads or writes) of the same location, at least one of them is a write, and at least one of them is non-atomic.*

Definition 6 (Race-freedom). *Execution $\mathcal{X}$ is* race-free *if for every two conflicting actions a and b in $\mathcal{X}$, we have* $\mathsf{hb}(a, b)$ *or* $\mathsf{hb}(b, a)$.

Definition 7 (Memory safety). *Execution $\mathcal{X}$ is* memory safe *if for every access action b in $\mathcal{X}$ there is an allocation action a in $\mathcal{X}$ such that a allocates the location accessed by b, and* $\mathsf{hb}(a, b)$.

Definition 8 (Initialized reads). *We say that in execution $\mathcal{X}$ all* reads are initialized *if for every read action r in $\mathcal{X}$ there is a write action w in $\mathcal{X}$ accessing the same location such that* $\mathsf{hb}(w, r)$.

Recall that for $\{P\}\ E\ \{Q\}$ to hold, there has to be a way to validly annotate every execution of E. Therefore, to establish the properties we are interested in, it suffices to prove the following theorem.

Theorem 1. *If $\mathcal{X}$ is a validly annotated execution, then $\mathcal{X}$ is memory safe and race-free, and all reads in $\mathcal{X}$ are initialized.*

Let us first concentrate on proving race-freedom for a validly annotated execution, $\mathcal{X}$. We start with two conflicting accesses a and b, and we first want to show is that there is a path in $\mathcal{X}$ connecting a and b. For this, we need the *heap compatibility* lemma.

Definition 9 (Independent edges). *In an execution, $\mathcal{X}$, a set of edges, $\mathcal{T}$, is* pairwise independent *if for all $(a, a'), (b, b') \in \mathcal{T}$, we have $\neg(\mathsf{sb} \cup \mathsf{rf})^*(a', b)$.*

Lemma 2 (Independent heap compatibility). *For every consistent execution $\mathcal{X}$, validly annotated by hmap, and pairwise independent set of edges $\mathcal{T}$, $\bigoplus_{e \in \mathcal{T}} hmap(e)$ is defined.*

Since this is exactly the same lemma that appears in the soundness proof of RSL, details of its proof can be found in [16].

Now, since our accesses a and b access the same location, and at least one of them is non-atomic, validity conditions guarantee that $hmap(\mathsf{SB_{in}}(a)) \oplus hmap(\mathsf{SB_{in}}(b))$ is undefined. Therefore, according to independent heap compatibility, execution $\mathcal{X}$ has to contain a path between a and b. Without loss of generality we can assume that the path goes from a to b.

Here we hit the main difference between RSL and FSL. In RSL existence of a path from a to b immediately implies $\mathsf{hb}(a, b)$, but in FSL we need to do some more work before getting there.

What we are going to do is take a closer look at the location ℓ that is being accessed by a and b. Denote the immediate sb predecessor of b by b'. We want to obtain a path π from a to b' such that for any edge $e \in \pi$, $\ell \in \mathsf{dom}(hmap(e))$. The building blocks for showing the existence of path π are the validity conditions and the independent heap compatibility lemma. Using these two, we can start at $\mathsf{SB_{in}}(a)$ and inductively build our path π, eventually reaching the node b'.

If we can show that the endpoints of π are related by hb, our work is done. In order to do that, let us look at the labels assigned to location ℓ by the edges along the path $\mathsf{SB_{in}}(a); \pi; \mathsf{SB_{in}}(b)$. (If $hmap(e)(\ell) = \mathsf{NA}[\_, \lambda]$, we say that edge e assigns label λ to location ℓ.) If we write out these labels, we get a string described by the regular expression $(\circ^+ \triangle^* \triangledown^+ \circ^+)^+$.

Sequences of “normal labels” $\circ$ are of no concern, since validity conditions mandate that $\circ$ appears only on sb-edges, which are part of hb relation. Therefore we turn our attention to the parts of the path where labeling of location ℓ is described by $\circ \triangle^* \triangledown^+ \circ$. Luckily, our validity conditions are designed in such a way as to reflect the definition sw relation (Fig. 1), which means that the node at which the $\circ$ label disappears is always sw-related with the node at which the $\circ$ label reappears.

This concludes the race-freedom part of the proof.

Next, we turn our attention to memory safety. If a is an action that accesses location ℓ, we follow the location ℓ starting from the $\mathsf{SB_{in}}(a)$ edge backwards in the execution until we reach the node that allocates ℓ. Along the way we make note of the labels appearing alongside ℓ, and similarly to the race-freedom proof, use the validity conditions to establish happens-before relation between the allocation and the access of ℓ.

The analysis here is a bit more complicated since now we have to also deal with atomic locations, which have a more complicated labeling structure, while in the case of race-freedom it sufficed to consider only non-atomic locations.

Finally, the fact that all reads are initialized is proven analogously to memory safety. We start at the read and in case the location that has been read is non-atomic, we follow it backwards until we reach a write to that location that happened before the read. When dealing with an atomic location, we have to be more careful. For atomic locations, we follow its initialization label until we find a write that happened before our read.

The only thing left to do is to prove all the inference rules valid according to the semantics of triples. Since all the proofs are analogous to the validity proofs of RSL's rules, we will concentrate on explaining the normalizability condition and the reason for the change in the definition of the triple semantics (see Section 5.2). We will do this by taking a look at the two rules that are new to FSL, (F-REL) and (F-ACQ).

Let us first turn our attention to the one that does not use the normalizability condition, namely (F-ACQ). To prove the rule valid, we need to validly annotate a very simple execution. It consists of a single node a, representing the acquire fence, one incoming and one outgoing sb-edge. Incoming edge ($\mathsf{SB_{in}}(a)$), is annotated by a heap satisfying precondition $\triangledown P$, plus some heap satisfying the frame R. In short, $\mathit{hmap}(\mathsf{SB_{in}}(a)) = h_P \oplus h_R$, where $h_P \in [\![\triangledown P]\!]$ and $h_R \in [\![R]\!]$. Our job is to annotate outgoing edge ($\mathsf{SB_{out}}(a)$) satisfying both the validity condition and triple semantics.

Since $h_P \in [\![\triangledown P]\!]$, from the assertion semantics we know that labeled$(h_P, \triangledown)$ holds, and that there is a heap $h_{\circ P} \in [\![P]\!]$ such that $h_P \approx h_{\circ P}$ and labeled$(h_{\circ P}, \circ)$. We set $\mathit{hmap}(\mathsf{SB_{out}}(a)) = h_{\circ P} \oplus h_R$. This satisfies the triple semantics, because $h_{\circ P} \oplus h_R \in [\![P * R]\!]$. The validity conditions are also satisfied by selecting $h_{\text{rel}} = h'_{\text{rel}} = \emptyset$, $h_{\text{acq}} = h_P$, $h'_{\text{acq}} = h_{\circ P}$, and $h_F = h_R$.

Let us now see what happens in the proof of validity of the (F-REL) rule. Here we start in a very similar situation with node a representing the release fence, one incoming and one outgoing sb-edge. The initial annotation is $\mathit{hmap}(\mathsf{SB_{in}}(a)) = h_P \oplus h_R$, where $h_R \in [\![R]\!]$, $h_P \in [\![P]\!]$, and $\mathsf{normalizable}(P)$. Before even trying to select a proper annotation for $\mathsf{SB_{out}}(a)$, we can see that there is a problem when trying to satisfy the validity condition. Namely, what should we choose for h_{rel}? There is no obvious heap labeled exactly by $\circ$, and validity requires labeled$(h_{\text{rel}}, \circ)$. This is where normalizability saves the day.

Normalizability of P gives us a decomposition $h_P = h_{\circ P} \oplus h'$, where $h_{\circ P} \in [\![P]\!]$, and labeled$(h_{\circ P}, \circ)$. We can now set $\mathit{hmap}(\mathsf{SB_{out}}(a)) = h_{\triangle P} \oplus h' \oplus h_R$, where $h_{\triangle P}$ is obtained from $h_{\circ P}$ by replacing all the labels appearing in $h_{\circ P}$ with $\triangle$. Validity is now satisfied by selecting $h_{\text{rel}} = h_{\circ P}$, $h'_{\text{rel}} = h_{\triangle P}$, $h_{\text{acq}} = h'_{\text{acq}} = \emptyset$, and $h_F = h' \oplus h_R$.

The RSL-style triple semantics is not satisfied by setting $hmap(\mathsf{SB}_{\mathsf{out}}(a)) = h_{\triangle P} \oplus h' \oplus h_R$, since we cannot guarantee $h_{\triangle P} \oplus h' \oplus h_R \in [\![\triangle P * R]\!]$, but FSL allows us to "forget" about h', and since $h_{\triangle P} \oplus h_R \in [\![\triangle P * R]\!]$, we satisfied our new triple semantics.

7 Related Work and Conclusion

We have presented FSL, an extension of RSL [16] for handling C11 memory fences. FSL is the first program logic that can handle C11 fences, as both existing program logics for C11, namely RSL [16] and GPS [14], do not support reasoning about these programming language features.

In this paper, our focus was on exploring modalities we introduced in order to specify the behavior of memory fences within the C11 model. We therefore chose to base our logic on the simpler logic (RSL) instead of the more powerful one, GPS. The simpler structure of RSL enabled us to give very simple specifications to fences, and retain simple rules for atomic accesses.

GPS is a noticeably more powerful logic than RSL. Its strength stems from the more flexible way in which GPS handles ownership transfer. Instead of relying on release and acquire permissions, GPS offers protocols, ghost resources and escrows, with which it is possible to verify a wider range of programs, such as an implementation of the RCU synchronization mechanism [13].

The soundness proof of GPS closely follows the structure of the soundness proof of RSL. Because of this, we feel confident that lessons learned in building FSL on top of RSL can be used in order to enrich GPS with FSL-style modalities, giving rise to a much more useful logic for reasoning about memory fences.

There have been other logics dealing with weak memory, mainly focusing on the TSO memory model. Notable examples include a rely-guarantee logic for x86-TSO by Ridge [11], and iCAP-TSO [12] which embeds separation logic inside a logic that deals with TSO concurrency. For the release-acquire model, there is also a recent Owicki-Gries logic called OGRA [8]. All of these logics assume stronger memory models than we have done in this paper.

Aside from program logics, there are model checking tools for C11 programs. Worth noting is CDSChecker [9] which includes support for memory fences.

An alternative approach to reasoning about weak memory behaviors is to restore sequential consistency. This can be done by placing fences in order to eliminate weak behavior [1], or by proving robustness theorems [4,5] stating conditions under which programs have no observable weak behaviors. So far, none of these techniques have been used to specifically target the C11 memory model.

Acknowledgments. We would like to thank Rayna Dimitrova, Ori Lahav, and the anonymous VMCAI'16 reviewers for their feedback. This work was supported by EC FET project ADVENT (308830).

References

1. 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)
2. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The problem of programming language concurrency semantics. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 283–307. Springer (2015)
3. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL 2011. pp. 55–66. ACM (2011)
4. 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)
5. 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)
6. ISO/IEC 14882:2011: Programming language C++ (2011)
7. ISO/IEC 9899:2011: Programming language C (2011)
8. 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)
9. 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)
10. O'Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. ACM Trans. Program. Lang. Syst. 31(3) (2009)
11. 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)
12. 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)
13. 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)
14. 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)
15. Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R., Zappa Nardelli, F.: Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In: Rajamani, S.K., Walker, D. (eds.) POPL 2015. pp. 209–220. ACM (2015)
16. 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)