Program Logic for Weak Memory Concurrency

Thesis approved by
the Department of Computer Science
Technische Universität Kaiserslautern
for the award of the Doctoral Degree
Doctor of Natural Sciences (Dr. rer. nat.)

to

Marko Doko

Date of Defense: 7 December 2021

Dean: Prof. Dr. Jens Schmitt

Reviewer: Prof. Dr. Rupak Majumdar

Reviewer: Prof. Dr. Noam Rinetzky

Reviewer: Prof. Dr. Peter Sewell

DE-386

Contents

Summary v

Zusammenfassung vii

Acknowledgments ix

1. Introduction 1

I. Per-Execution Models 5

2. The C11 Memory Model 7
- 2.1. Programming Language . . . 7
- 2.2. Relational Notation . . . 8
- 2.3. The Model of Executions . . . 9
- 2.4. Examples . . . 19
  - 2.4.1. Store Buffering . . . 19
  - 2.4.2. Message Passing . . . 21
  - 2.4.3. CAS Atomicity . . . 23
- 2.5. The Out-of-Thin-Air Problem . . . 23

3. Fenced Separation Logic 29
- 3.1. Syntax . . . 29
  - 3.1.1. Standard Rules . . . 30
  - 3.1.2. Rules for Non-atomic Accesses . . . 31
  - 3.1.3. Basic Rules for Atomic Accesses . . . 32
  - 3.1.4. Basic Examples . . . 34
  - 3.1.5. CAS rules . . . 38
  - 3.1.6. Examples Using CAS Rules . . . 42
  - 3.1.7. Ghost State . . . 44
  - 3.1.8. Examples Using the Ghost State . . . 45
- 3.2. Semantics . . . 48
  - 3.2.1. Heaps as Models of Assertions . . . 48
  - 3.2.2. Semantics of Triples – Heap-annotated Executions . . . 57
- 3.3. Soundness . . . 68
  - 3.3.1. Properties of Validly Annotated Executions . . . 68
  - 3.3.2. The Adequacy Theorem . . . 75
  - 3.3.3. Soundness of the Inference Rules . . . 75

4. Case Study: Verification of Atomic Reference Counter 83
4.1. Algorithm . . . 83
4.1.1. Why Is ARC Correct? . . . 84
4.2. Verification of ARC . . . 85
4.2.1. Defining the ARC Predicate . . . 86
4.2.2. The ARC Correctness Theorem . . . 88
4.2.3. Dealing With Deallocation . . . 92

II. Multi-Execution Models 93

5. Promising Semantics 95
5.1. Memory and Messages . . . 95
5.2. Threads . . . 96
5.3. Constraining Promises . . . 97
5.4. Full Machine . . . 98

6. Weak Separation Logic 99
6.1. Syntax . . . 99
6.1.1. Programming Language . . . 99
6.1.2. The Assertions of the Logic . . . 100
6.1.3. The Rules for Relaxed Accesses . . . 101
6.1.4. Examples . . . 103
6.2. Semantics . . . 106
6.2.1. Semantics of Assertions . . . 106
6.2.2. Erasure and View Shift . . . 108
6.2.3. Semantics of Triples . . . 110
6.3. Soundness . . . 111
6.3.1. The Correctness Theorem . . . 113
6.4. An Overview of Full SLR . . . 114
6.4.1. Comparison with FSL . . . 114

III. Related Work and Discussion 117

7. Related Work 119
7.1. Program Logics . . . 119
7.1.1. Logics for Language-level Memory Models . . . 119
7.1.2. Logics for Hardware-level Memory Models . . . 121
7.2. Other Verification Topics . . . 121

8. Lessons Learned and Future Directions 123
8.1. Looking Back . . . 123
8.2. Looking Forward . . . 125

Bibliography 127

A. Coq Implementation of FSL 139

Curriculum Vitae 141

Summary

In order to improve performance or conserve energy, modern hardware implementations have adopted *weak memory* models; that is, models of concurrency that allow more outcomes than the classic *sequentially consistent* (SC) model of execution. Modern programming languages similarly provide their own language-level memory models, which strive to allow all the behaviors allowed by the various hardware-level memory models, as well as those that can occur as a result of desired compiler optimizations.

As these weak memory models are often rather intricate, it can be difficult for programmers to keep track of all the possible behaviors of their programs. It is therefore very useful to have an abstraction layer over the model that can be used to ensure program correctness without reasoning about the underlying memory model. Program logics are a way of constructing such an abstraction—one can use their syntactic rules to reason about programs, without needing to understand the messy details of the memory model for which the logic has been proven sound.

Unfortunately, most of the work on formal verification in general, and program logics in particular, has so far assumed the SC model of execution. This means that new logics for weak memory have to be developed.

This thesis presents two such logics—*fenced separation logic* (FSL) and *weak separation logic* (Weasel)—which are sound for reasoning under two different weak memory models.

FSL considers the C/C++ concurrency memory model, supporting several of its advanced features. The soundness of FSL depends crucially on a specific strengthening of the model which eliminates a certain class of undesired behaviors (so-called *out-of-thin-air behaviors*) that were inadvertently allowed by the original C/C++ model.

Weasel works under weaker assumptions than FSL, considering a model which takes a more fine-grained approach to the out-of-thin-air problem. Weasel's focus is on exploring the programming constructs directly related to out-of-thin-air behaviors, and is therefore significantly less feature-rich than FSL.

Using FSL and Weasel, the thesis explores the key challenges in reasoning under weak memory models, and what effect different solutions to the out-of-thin-air problem have on such reasoning. It explains which reasoning principles are preserved when moving from a stronger to a weaker model, and develops novel proof techniques to establish soundness of logics under weaker models.

Zusammenfassung

Moderne Hardware ist, um ihre Leistung zu verbessern oder Energie zu sparen, dazu übergegangen, sogenannte *weak memory*-Modelle (schwach konsistente Speichermodelle) zu verwenden; das sind Nebenläufigkeitsmodelle, die mehr Verhalten als klassische *sequentiell konsistente* (SC)-Ausführungsmodelle erlauben. Moderne Programmiersprachen haben entsprechend ihre eigenen sprachspezifischen Speichermodelle, die versuchen, alle Verhalten, die die verschiedenen Hardware-Modelle sowie solche, die aufgrund von Compiler-Optimierungen auftreten, zu unterstützen.

Da diese weak-memory-Modelle oft sehr kompliziert sind, kann es für Programmierer schwierig sein, alle Verhalten ihrer Programme im Blick zu behalten. Es ist deswegen sehr nützlich, eine Abstraktionsschicht über dem Modell zu haben, mit dem man Programmkorrektheit ohne Rückgriff auf das zugrundeliegenden Modell nachweisen kann. Programmlogiken sind ein Weg, eine solche Abstraktion zu konstruieren — man kann ihre syntaktischen Regeln nutzen, um über ein Programm zu argumentieren, ohne die verzwickten Details des Speichermodells, für das die Logik als korrekt bewiesen wurde, verstehen zu müssen.

Leider haben bisher die meisten Arbeiten über formale Verifikation im Allgemeinen und über Programmlogiken im Speziellen das SC-Ausführungsmodell angenommen. Das bedeutet, dass neue Logiken für weak memory-Modelle entwickelt werden müssen.

Diese Dissertation präsentiert zwei solche Logiken—*fenced separation logic* (FSL) und *weak separation logic* (Weasel)—die korrekt bezüglich zwei verschiedener weak memory-Modelle sind.

FSL betrachtet das C/C++-Nebenläufigkeitsmodell, und unterstützt einige fortgeschrittene Aspekte dieses Modells. Die Korrektheit von FSL hängt wesentlich von einer spezifischen Verstärkung des Modells ab, die eine spezifische Klasse von unerwünschten Verhalten (sogenannte *out-of-thin-air*-Verhalten) eliminiert, die unbeabsichtigt vom ursprünglichen C/C++-Modell erlaubt wurden.

Weasel funktioniert unter schwächeren Annahmen als FSL, und betrachtet ein Modell mit einer feingranulareren Herangehensweise an das out-of-thin-air-Problem. Der Fokus von Weasel ist die Untersuchung von Programmkonstrukten, die direkt mit out-of-thin-air-Verhalten zusammenhängen, und hat somit wesentlich weniger Features als FSL.

Mit Hilfe von FSL und Weasel erforscht die Dissertation die wesentlichen Herausforderungen beim Argumentieren über weak memory-Modelle, und welchen Effekt verschiedene Lösungen des out-of-thin-air-Problems auf solche Argumente haben. Sie zeigt, welche Argumentationsweisen beim Übergang von stärkeren zu schwächeren Modelle erhalten bleiben, und entwickelt neuartige Beweistechniken zum Nachweis der Korrektheit von Logiken unter schwächeren Modellen.

Acknowledgments

This PhD thesis would not have been possible without the help and support from a great number of people. I will give my best to extend the proper gratitude to all of you. To those I inadvertently fail to mention, I sincerely apologize.

First and foremost, a huge thank you goes to my advisor Viktor Vafeiadis, not only for the continual guidance and support during my PhD studies, but also for building a group which was significantly more than just a research group. You facilitated a sense of camaraderie in the group, and showed us how scientific work can be done and organized by focusing on research and enjoying it, instead of chasing deadlines.

Everyone at the institute deserves acknowledgment for creating the most welcoming and supportive environment I have ever encountered.
I would especially like to thank our office staff in Kaiserslautern, Mouna, Roslyn, Susanne, and Vera, for always being there not only to help guide me (and others) through the forest of bureaucracy, but also providing immeasurable help in navigating the German society for us foreigners. You are the unsung heroes of academia.
Mary-Lou, thank you for not giving up on me and making sure I push through the rough times. Your support meant a lot.

During the time spent in Kaiserslautern, I had the privilege of enjoying friendship of many people with whom I had long discussions, fun times, travels, and enjoyed the simple pleasures of life. These are also the people who were with me in the difficult times; they made me feel welcome in a new country; they helped me move (twice!); and they checked on me and picked me up when I was down. In alphabetical order, they include: Amir Bahador, Anthony, Azalea, Burcu, Damien, Dmitry, Filip, Goce, Isa, Ivan G., James, Johannes, Kata, Kaushik, Laura, Lovro, Manuel, Marko H., Michalis, Mitra, Murat, Nastaran, Nivedita, Ori, Rayna, Rosa, Sai, Simin, Soham, Utkarsh, and William.

To my friend group from the University of Zagreb days, Darko, Davor, Iva, Ivan V., Ivo U., Jakiša, Josip, Jurica, Lidija, Lucija, Mia, Stipe, Veky, thank you for welcoming me whenever I come back, even if we were not keeping in touch as much as we should have. You are the reason I sometimes regret not using social networks.

Ružica, thank you for believing in me and steering me towards the PhD program at the MPI-SWS. I was always able to rely on you.

Three of my schoolteachers deserve a special recognition. Blaženka Jerčić, for protecting and supporting me when I needed it the most; Ratko Čizmić, for providing me with insights I realized came from him only later in life; and Željko Matić who taught me programming and acted as a mentor and a friend throughout high-school and beyond.

Shelley, thank you for the love and support over the last two years. It has been a rough time, and I am very lucky to have met you.

Finally, I would like to thank my immediate family, my brother Ivo, mother Pavica, father Tomo, and grandma Pavica. Thank you for everything you have done for me; I needed you more than you will ever know. Mom, you saved my life.

1. Introduction

In the modern age of multi-core processors, being able to reason about concurrent programs is of utmost importance. There are various approaches to reasoning about programs, but this thesis focuses on deductive reasoning; more specifically program logics for reasoning about concurrent programs.

Program logics are invaluable tools for formal verification. They enable the user to construct a proof of programs' correctness without the need to understand the intricacies of the underlying execution model assumed by the programming language. How fruitful and influential has this idea been in the wider area of program verification can be seen from the diagram in Fig. 1.1. There we see a snapshot of the history of program logics, showing some notable developments and their connections.

Traditionally, the vast majority of verification techniques assume *sequential consistency* (Lamport, 1979) as the semantic model for the execution of concurrent programs—in fact, all but four program logic shown in Fig. 1.1 assume sequential consistency as the underlying model.

Sequential consistency is a useful, simple, and intuitive model. According to sequential consistency, an execution of a concurrent program is some interleaving of the instructions of the program's threads. In other words, the program's threads take turns executing. Unfortunately, sequential consistency does not capture all the intricacies that happen when executing concurrent programs on real-world hardware. To see the problem, consider the following example, known as *store-buffering*.

```
              x = y = 0;
x = 1;                ║ y = 1;
a = y; // reads 0     ║ b = x; // reads 0
```
(SB)

Can both of the loads get the value 0, as suggested by the comments, i.e., can both thread-local variables `a` and `b` have the value 0 at the end of the program's execution?

According to sequential consistency, at least one of `a` and `b` will have the value 1. The reason is that after initializing the shared locations `x` and `y` to 0, either the left or the right thread starts execution, so either `x` or `y` gets set to 1 before any of the loads execute. Therefore at least one of the loads has to see the updated value.

However, running the (SB) program on any modern system will occasionally produce the outcome where both `a` and `b` are set to 0. How is that possible? We can explain the unexpected outcome by looking at either hardware behavior or at compiler optimizations.

Looking at hardware, we should have in mind that modern processors, when executing store instructions, do not write the value directly to the main memory, but instead write to the local cache, and changes to the cache get propagated to the main memory at some

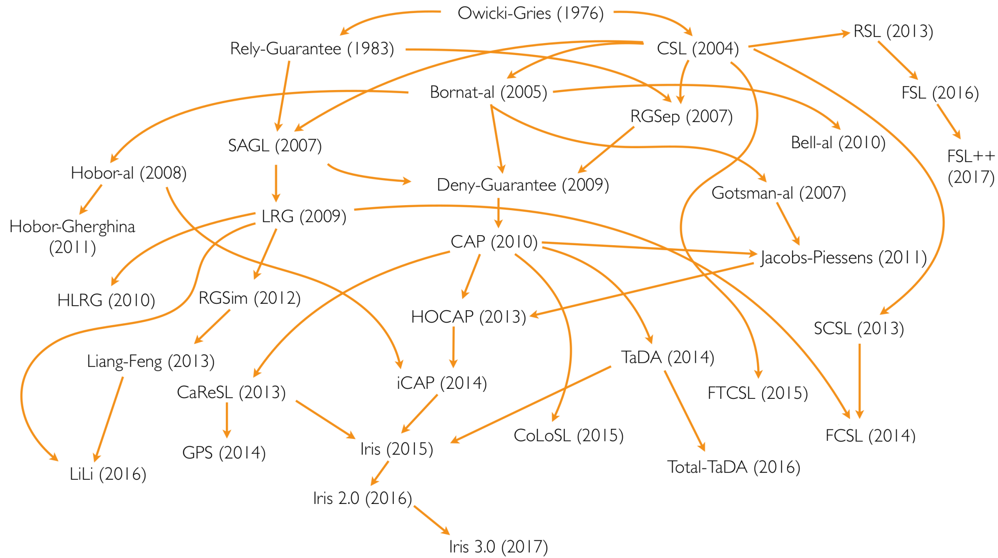


Figure 1.1.: History of concurrent program logics.
Credit to Ilya Sergey (https://ilyasergey.net/).

later time. Knowing that, it is easy to see that a possible execution of (SB) is for the left thread to execute to completion (putting `x = 1` in its cache without propagating it to the main memory), then the right thread executes to completion, and only then does the store made by the left thread get committed to the main memory. This sequence of events results in both threads observing the value 0 when executing their loads.

An alternative explanation comes from the point of view of optimizing compilers. The two instructions in the left thread (`x = 1` and `a = y`) are independent because they access two different locations. As such, the compiler is well within its rights to reorder those two instructions. Once the compiler transforms the program by placing `a = y` before `x = 1` in the left thread, we can get the outcome where both `a` and `b` are set to 0 even if we assume the hardware behaves according to sequential consistency.

As we can see, as a consequence of hardware features and compiler optimizations (which are necessary for obtaining good performance), sequential consistency is not a fitting model for the low-level behavior of concurrent programs. As a result, *weak*[1] *memory models* started emerging. The goal of those models is to provide a comprehensive model of concurrent programs' behavior, taking into account the effects of modern hardware architecture, compiler optimizations, and their interplay. The two most notable early attempts are the Java Memory Model (Manson et al., 2005) and the C11 memory model (ISO/IEC, 2011a,b), coming from Java and C/C++ programming languages, respectively.

Most early models followed the "per-execution" definition style, where program behaviors are described by looking at candidate executions, and deciding whether to accept or discard each one of the candidates based on a set of axiomatic constraints. Soon, certain

[1] "Weak" in this context refers to being weaker (i.e., providing fewer guarantees) than sequential consistency

LB	
`x = y = 0;`	
`a = y;`	`b = x;`
`x = 1;`	`y = 1;`
$\mathtt{a} = 1 \wedge \mathtt{b} = 1$ observable	

LB+FAKEDEP	
`x = y = 0;`	
`a = y;`	`b = x;`
`x = 1 + a - a;`	`y = 1 + b - b;`
$\mathtt{a} = 1 \wedge \mathtt{b} = 1$ observable	

LB+DEP	
`x = y = 0;`	
`a = y;`	`b = x;`
`x = a;`	`y = b;`
$\mathtt{a} = 1 \wedge \mathtt{b} = 1$ not observable	

Figure 1.2.: Three load buffering variants. From left to right: classic load buffering, load buffering with fake dependency, and load buffering with data dependency.

deficiencies in the per-execution models became evident. The easiest way to see the issue is by way of the examples given in Fig. 1.2 showcasing the behavior known as *load buffering*.

The (LB) example can be understood similarly to the (SB) example by a reodering of independent instructions, either by the compiler or by clever hardware. In (LB+DEP), the instructions of each thread cannot be reordered because of the dependency between them and so the annotated outcome is not possible. Finally, in (LB+FAKEDEP), while there is a dependency between the instructions of each thread, it is a fake syntactic dependency that can be removed by an optimizing compiler. A compiler can transform the expression `1 + a - a` into `1` (and similarly for the right thread), thereby transforming the program to (LB). For a detailed discussion of the intricacies of load buffering, see Section 2.5.

Unfortunately, as demonstrated by Batty et al. (2015), the three load buffering variants cannot be classified in accordance with the observed behavior produced by modern hardware and compilers. As a response to this issue, a new generation of memory models arose: models that look at multiple potential executions of the program in order to explain a single behavior (Pichon-Pharabod and Sewell, 2016; Kang et al., 2017; Chakraborty and Vafeiadis, 2019). To distinguish them from the per-execution models, we will henceforth refer to these newer models as multi-execution models.

Verification under weak memory models is challenging due to the fact that many proof techniques useful in the sequentially consistent setting (e.g., relying on linear execution traces) do not apply any more. This challenge is made more difficult by the existence of two significantly different definition styles of weak memory models (per-execution and multi-execution).

This thesis addresses the following three questions regarding verification in the weak memory setting.

1. Which reasoning principles "survive" the transition from sequential consistency to weak memory, and are those reasoning principles strong enough to reason about programming patterns appearing in practice?

2. Which new proof techniques do we need in order to prove soundness of reasoning principles under weak memory models, and how do these proof techniques differ between the two different styles of models?

3. Are there significant differences in reasoning principles under per-execution and multi-execution models?

In order to answer the above questions, we developed two program logics; one for a per-execution model and one for a multi-execution model. Both of those logics build on top of the first weak memory logic, relaxed separation logic (RSL) (Vafeiadis and Narayan, 2013). RSL is a good starting point as it identifies the basic reasoning principles sound under weak memory. However, RSL keeps to the well-grounded part of the weak-memory world (where all models agree on program behavior) and shies away from the more shaky ground of disagreement between per-execution and multi-execution models. In this thesis, we show that it is possible to establish a firm foothold in this uncertain area of weak memory.

The first part of the thesis presents *fenced separation logic* (FSL). This logic extends RSL onto a much wider fragment of the C11 memory model (which is a per-execution model), and it proves to be expressive enough to enable verification of some complex synchronization patterns appearing in practice.

The second part of the thesis showcases *weak separation logic* (Weasel), a fragment of SLR (Svendsen et al., 2018), an extension/adaptation of RSL to the multi-execution promising semantics of Kang et al. (2017). Compared to FSL, SLR (and hence Weasel too) is a much less expressive logic. This is a consequence of significant challenges which arise when the underlying model changes from a per-execution one to a multi-execution one.

The answers to the three questions posed above, provided by FSL and SLR are:

1. Enough desirable reasoning principles remain valid under weak memory to enable deductive reasoning about programs. In the case of per-execution models, we can confidently say that those reasoning principles allow reasoning about more complex programming patterns.

2. Novel proof techniques are needed when moving from sequential consistency to per-execution weak memory models. Further more intricate approaches are needed when considering multi-execution models.

3. There are important and useful reasoning principles which are sound under per-execution models, but do not hold under multi-execution models.

Besides the three main questions, the design of FSL and SLR was guided by the view of program logics as a way to hide the underlying complexity of the model from the programmers, and present them with a cleaner and simpler set of rules. This design choice led us towards concentrating on minimalistic rules which trade expressiveness for readability and simplicity.

Part I.

Per-Execution Models

2. The C11 Memory Model

This chapter presents the memory model which is used as the basis for building the logic presented in Chapter 3.

The model we will be using is a variant of the memory model introduced by the C and C++ standardizing committees (ISO/IEC, 2011a,b). This model (and its variations) has become commonly known as the C11 memory model. The first formalization of the C11 memory model has been presented by Batty et al. (2011). Since it first appeared, the model has seen several alternative presentations and revisions (e.g., Vafeiadis and Narayan, 2013; Vafeiadis et al., 2015; Lahav et al., 2017). The presentation in this chapter is primarily based on Lahav et al. (2017), but we restrict our attention to the features considered by fenced separation logic of Chapter 3.

2.1. Programming Language

We use a simple programming language given by the following definition.

Definition 2.1 (Programming language) Let Var (the set of *variables*) be a countable set, and $\mathsf{Val} = \mathsf{Loc} = \mathbb{N}$ (sets of *values* and *locations*). The programming language is defined by the following grammar.

$$\begin{aligned} e \in \mathsf{AExp} &::= x \mid v \qquad \text{where } x \in \mathsf{Var}, \text{ and } v \in \mathsf{Val} \\ E \in \mathsf{Exp} &::= e \mid \mathbf{let}\ x = E\ \mathbf{in}\ E' \mid \mathbf{if}\ e\ \mathbf{then}\ E\ \mathbf{else}\ E' \mid \mathbf{repeat}\ E\ \mathbf{end} \mid E \parallel E' \mid \\ &\qquad \mathbf{alloc}() \mid [e]_\tau \mid [e]_\sigma := e' \mid \mathbf{CAS}_{\zeta,\xi}(e, e', e'') \mid \textsc{fence}_\gamma \\ &\qquad \text{where } \tau \in \{\text{na}, \mathbf{acq}, \mathbf{rlx}\}, \sigma \in \{\text{na}, \mathbf{rel}, \mathbf{rlx}\}, \zeta \in \{\mathbf{acq_rel}, \mathbf{acq}, \mathbf{rel}, \mathbf{rlx}\}, \\ &\qquad \xi \in \{\mathbf{acq}, \mathbf{rlx}\}, \text{ and } \gamma \in \{\mathbf{rel}, \mathbf{acq}\} \end{aligned}$$

In the definition above, both values and locations are natural numbers, but we keep the Val and Loc notations as they can be useful for presentation purposes, as an additional signal telling us when is something treated as a location, and when is it treated as a value.

The atomic expressions of the language consist of variables and values. The program expressions consist of atomic expressions, let-bindings, conditionals, loops, parallel composition, allocation, loads, stores, atomic compare-and-swap (CAS) instructions, and memory fences. We will often write $E; E'$ instead of $\mathbf{let}\ x = E\ \mathbf{in}\ E'$ if x does not appear as a free variable in E'.

As it is customary in C-like languages, conditionals interpret non-zero values as true, and zero as false. The loop $\mathbf{repeat}\ E\ \mathbf{end}$ executes the expression E until it returns a non-zero value.

Memory accesses are annotated by their *access type*, which are broadly subdivided into *non-atomic* (na) accesses and *atomic* accesses. The non-atomic accesses can be thought as the “regular accesses”, i.e., those that are used for general purpose data manipulation in a program. The atomic accesses are further subdivided into *relaxed* (**rlx**), *release* (**rel**), *acquire* (**acq**), and *acquire-release* (**acq_rel**) accesses. The purpose of atomic accesses is to, together with memory fences, provide the means of inter-thread communication and synchronization.

The precise nature of the various access types will be made clear in Sections 2.3 and 2.4 where we define the semantics of our programming language and look at variety of examples.

2.2. Relational Notation

Before we proceed with presenting the semantics of our programming language, let us first review the notation used when talking about binary relations. Most of it is standard, with some non-standard notation introduced in order to make the presentation of the model more concise.

Definition 2.2 Let A and B be sets, $S \subseteq A$, $f\colon A \to B$, and $R, R' \subseteq A \times A$ binary relations on A. We use the following notations and terminology.

- The *domain* of R is
$$\operatorname{dom} R \coloneqq \{x \mid \exists y\,.(x,y) \in R\}\,.$$
- The *range* of R is
$$\operatorname{rng} R \coloneqq \{y \mid \exists x\,.(x,y) \in R\}\,.$$
- The *inverse relation* of R is
$$R^{-1} \coloneqq \{(x,y) \mid (y,x) \in R\}\,.$$
- The *identity relation* on S is
$$[S] \coloneqq \{(x,x) \mid x \in S\}\,.$$
- The *reflexive closure* of R is
$$R^{?} \coloneqq R \cup [A]\,.$$
- The *left-to-right composition* of R and R' is
$$R; R' \coloneqq \left\{(x,z) \mid \exists y.\,(x,y) \in R \wedge (y,z) \in R'\right\}.$$
- The *iterated composition* of relation R is given by the recursion
$$R^1 \coloneqq R\,,$$
$$R^{n+1} \coloneqq R^n; R\,.$$

- The *transitive closure* of R is

$$R^+ \coloneqq \bigcup_{n=1}^{\infty} R^n \,.$$

- The *reflexive and transitive closure* of R is

$$R^* \coloneqq (R^+)^? \,.$$

- The relation R is *irreflexive* if $R \cap [A] = \emptyset$.
- The relation R is *functional* if $\forall x, y, z.\, (x, y) \in R \wedge (x, z) \in R \rightarrow y = z$.
- The f-*equivalence* relation on A is

$$=_f \coloneqq \{(x, y) \in A \times A \mid f(x) = f(y)\} \,.$$

- The f-*restriction* of the relation R is

$$R|_f \coloneqq R \cap =_f \,.$$

Instead of having a special notation for restricting a relation to specific subsets, we will use the fact that for any binary relation R and sets S and T we have $[S]; R; [T] = R \cap (S \times T)$, which enables us to use the composition to talk about restrictions.

2.3. The Model of Executions

The semantics of programs is given *declaratively*, by assigning a set of *execution graphs* to each program. These execution graphs are constructed from the program text so that the nodes of the graph represent actions taken by the program, and various types of edges provide structural information such as which actions belong to the same thread and which load action reads from which store action.

The formal definition of the program semantics is given in three stages. First, from the program text, a set of corresponding *pre-executions* is generated. The pre-executions are graphs that do little more than transcribe the syntactical structure of the program into a graph form. The second step is to enhance the pre-executions with some additional information (which load reads from which store, what is the ordering of the store actions on a particular location, etc.). Intuitively, this second step is what provides us with the "dynamic" part of the execution, telling us what "happened" while the program was running. Lastly, we constrain the set of possible execution graphs by a set of axioms telling us what is possible to happen for a given program. An obvious example of a necessary constraint is to require that a load has to read from stores writing to the same location as the load is reading from.

Now that we have an intuitive overview of the structure of the semantics definition, let us slowly proceed by focusing on the definition of pre-executions. Formally, pre-executions are graphs where the nodes describe shared memory accesses performed by the program, and the edges represent the program order between those accesses (i.e., the ordering between the statements in the source which gave rise to the corresponding actions performed by the program). Each node in the graph is given a label, telling us what kind of a memory access has been performed (load, store, atomic update, or a fence), and which access mode was used. The label `skip` is used to represent all the program actions that are not shared memory accesses (thread-local computation, and forks and joins of threads). Pre-executions also record a return-value, which can be either a value of a finite pre-execution, or a special symbol $\bot$ representing a pre-execution that is a finite prefix of an infinite pre-execution.

Definition 2.3 (Pre-execution) A tuple $(r, \mathcal{A}, \mathsf{lab}, \mathtt{po})$ where

- $\mathcal{A}$ is a finite set of *nodes* (also called *events*),
- $\mathsf{lab} \colon \mathcal{A} \to \mathsf{Lab}$ is a function mapping nodes to *labels*,
- $\mathtt{po}$ is a strict partial order on $\mathcal{A}$, called *program-order* and
- $r \in \mathsf{Val} \uplus \{\bot\}$ is the *result* or *return value*,

where

$$\begin{aligned}\mathsf{Lab} = {} & \{\mathtt{skip}\} \cup \{\mathtt{A}(\ell) \mid \ell \in \mathsf{Loc}\} \cup \\ & \{\mathtt{R}_\tau(\ell, v) \mid \ell \in \mathsf{Loc} \wedge v \in \mathsf{Val} \wedge \tau \in \{\mathrm{na}, \mathbf{acq}, \mathbf{rlx}\}\} \cup \\ & \{\mathtt{W}_\tau(\ell, v) \mid \ell \in \mathsf{Loc} \wedge v \in \mathsf{Val} \wedge \tau \in \{\mathrm{na}, \mathbf{rel}, \mathbf{rlx}\}\} \cup \\ & \{\mathtt{U}_\tau(\ell, v, v') \mid \ell \in \mathsf{Loc} \wedge v, v' \in \mathsf{Val} \wedge \tau \in \{\mathbf{acq_rel}, \mathbf{acq}, \mathbf{rel}, \mathbf{rlx}\}\} \cup \\ & \{\mathtt{F}_\tau \mid \tau \in \{\mathbf{acq}, \mathbf{rel}\}\}\end{aligned}$$

is called a *pre-execution*.

We now need to formally connect programs with their pre-executions. This is done in Definition 2.4 by assigning the corresponding pre-executions to programs via the structural recursion over programs.

Thread-local computations (i.e., the value expressions) are represented by skip nodes, since they do not interact with the shared memory. The return value of thread-local computations is simply the value being computed.

Allocation instructions are represented by allocation nodes ($\mathtt{A}(\ell)$) recording the location ℓ that has been allocated. Here, for the sake of simplicity of presentation and to keep in line with the Coq formalization as much as possible, we allow allocations only of single locations. However, adding support for allocating arbitrarily large memory blocks is completely straightforward. All that is needed is to let allocation instruction generate a po-sequence of allocation nodes which would record allocation of a sequence of consecutive memory locations.

Load and store instructions are represented by read ($\mathtt{R}_\tau(\ell, v)$) and write ($\mathtt{W}_\tau(\ell, v)$) nodes, respectively; recording the location accessed (ℓ), the value read or written (v), as well as the access type (τ). The return value of loads and stores is the value being read or written.

The compare-and-swap instruction $\mathbf{CAS}_{\tau,\sigma}(\ell, v_o, v_n)$ has the most interesting representation. A successful CAS, i.e., one managing to read the expected old value (v_o) and replace it with the new value (v_n), generates an update node ($\mathtt{U}_\tau(\ell, v_o, v_n)$)). An unsuccessful CAS generates a read node ($\mathtt{R}_\sigma(\ell, v)$) which cannot have v_o as the value read. The return value of a CAS instruction is always the value read (regardless of whether the CAS was successful or not).

A fence instruction simply generates a fence node ($\mathtt{F}_\tau$) which records the type of the fence. The return value can be any value (i.e., anything but the $\bot$ symbol). (The return values assigned to fences will never be used for anything. They are there simply for the uniformity of the definition.)

A conditional branch is, as expected, represented by a graph representing either the expression in the if branch or the expression in else branch, depending on the value of the condition.

Loops are where the subtlety of the definition is hidden. The idea is to represent loops by pasting the graphs representing the loop body one after another until the loop body returns a non-zero value. However, we want to avoid generating infinite graphs, and we do that by allowing loops to generate a graph with $\bot$ as the return value. That kind of a graph represents a loop that is "still running", i.e., it is an incomplete pre-execution of a loop which is yet to terminate or may never terminate. This way we cover infinite executions by representing all of their prefixes obtained finite loop unrfoldings.

The sequential composition (i.e., let-binding) is represented in one of two ways. If the first expression terminates (i.e., the return value of the first expression is not $\bot$) then the representation is given by the graphs of the two expressions pasted one after the other. If the first expression does not terminate (i.e., its return value is $\bot$), then the representation of the sequential composition is the graph of the first expression (representing that the first expression is still running, and the second expression has not even begun to execute).

The parallel composition is represented by placing the graphs of the two expressions side by side, and inserting two skip nodes—one at the beginning, representing the fork event, and one at the end, representing the join event. The return value is $\bot$ if at least one of the threads does not terminate, or an arbitrary value otherwise.

Definition 2.4 A pre-execution $G = (r, \mathcal{A}, \mathsf{lab}, \mathtt{po})$ *represents* a program E (denoted $G \vdash E$) according to the following structural recursion on E.

$(v, \{a\}, \mathsf{lab}, \emptyset) \vdash v$
  iff $v \in \mathsf{Val} \land \mathsf{lab}(a) = \mathtt{skip}$

$(\ell, \{a\}, \mathsf{lab}, \emptyset) \vdash \mathbf{alloc}()$
  iff $\ell \in \mathsf{Loc} \land \mathsf{lab}(a) = \mathtt{A}(\ell)$

$(v, \{a\}, \mathsf{lab}, \emptyset) \vdash [\ell]_\tau := v$
  iff $\ell \in \mathsf{Loc} \land v \in \mathsf{Val} \land \mathsf{lab}(a) = \mathtt{W}_\tau(\ell, v)$

$(v, \{a\}, \mathsf{lab}, \emptyset) \vdash [\ell]_\tau$
iff $\ell \in \mathsf{Loc} \wedge v \in \mathsf{Val} \wedge \mathsf{lab}(a) = \mathtt{R}_\tau(\ell, v)$

$(r, \{a\}, \mathsf{lab}, \emptyset) \vdash \mathbf{CAS}_{\tau,\sigma}(\ell, v_o, v_n)$
iff $\ell \in \mathsf{Loc} \wedge v_o, v_n \in \mathsf{Val} \wedge$
$((r = v_o \wedge \mathsf{lab}(a) = \mathtt{U}_\tau(\ell, v_o, v_n)) \vee (r \in \mathsf{Val} \setminus \{v_o\} \wedge \mathsf{lab}(a) = \mathtt{R}_\sigma(\ell, r)))$

$(r, \{a\}, \mathsf{lab}, \emptyset) \vdash \textsc{fence}_\gamma$
iff $r \in \mathsf{Val} \wedge \mathsf{lab}(a) = \mathtt{F}_\tau$

$(r, \mathcal{A}, \mathsf{lab}, \mathtt{po}) \vdash \mathbf{if}\ v\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2$
iff $(v \neq 0 \wedge (r, \mathcal{A}, \mathsf{lab}, \mathtt{po}) \vdash E_1) \vee (v = 0 \wedge (r, \mathcal{A}, \mathsf{lab}, \mathtt{po}) \vdash E_2)$

$(r, \mathcal{A}, \mathsf{lab}, \mathtt{po}) \vdash \mathbf{repeat}\ E\ \mathbf{end}$
iff $\exists r', \mathcal{A}_1, \ldots, \mathcal{A}_n, \mathsf{lab}_1, \ldots, \mathsf{lab}_n, \mathtt{po}_1, \ldots, \mathtt{po}_n.$
$\mathcal{A} = \biguplus_{i=1}^n \mathcal{A}_i \wedge \mathsf{lab} = \bigcup_{i=1}^n \mathsf{lab}_i \wedge$
$\mathtt{po} = \bigcup_{i=1}^n \mathtt{po}_i \cup \bigcup_{1 \leqslant i < j \leqslant n} \mathcal{A}_i \times \mathcal{A}_j \wedge$
$(\forall i < n.\, (0, \mathcal{A}_i, \mathsf{lab}_i, \mathtt{po}_i) \vdash E) \wedge (r', \mathcal{A}_n, \mathsf{lab}_n, \mathtt{po}_n) \vdash E \wedge$
$((r' \neq 0 \wedge r = r') \vee (r' = 0 \wedge r = \bot))$

$(r, \mathcal{A}, \mathsf{lab}, \mathtt{po}) \vdash \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2$
iff $r = \bot \wedge (\bot, \mathcal{A}, \mathsf{lab}, \mathtt{po}) \vdash E_1 \vee$
$\exists r_1, \mathcal{A}_1, \mathcal{A}_2, \mathsf{lab}_1, \mathsf{lab}_2, \mathtt{po}_1, \mathtt{po}_2.$
$\mathcal{A} = \mathcal{A}_1 \uplus \mathcal{A}_2 \wedge \mathsf{lab} = \mathsf{lab}_1 \cup \mathsf{lab}_2 \wedge \mathtt{po} = \mathtt{po}_1 \cup \mathtt{po}_2 \cup \mathcal{A}_1 \times \mathcal{A}_2 \wedge$
$r_1 \neq \bot \wedge (r_1, \mathcal{A}_1, \mathsf{lab}_1, \mathtt{po}_1) \vdash E_1 \wedge (r, \mathcal{A}_2, \mathsf{lab}_2, \mathtt{po}_2) \vdash E_2[r_1/x]$

$(r, \mathcal{A}, \mathsf{lab}, \mathtt{po}) \vdash E_1 \parallel E_2$
iff $\exists r_1, r_2, a_{\text{fork}}, a_{\text{join}}, \mathcal{A}_1, \mathcal{A}_2, \mathsf{lab}_1, \mathsf{lab}_2, \mathtt{po}_1, \mathtt{po}_2.$
$\mathcal{A} = \mathcal{A}_1 \uplus \mathcal{A}_2 \uplus \{a_{\text{fork}}, a_{\text{join}}\} \wedge$
$\mathsf{lab} = \mathsf{lab}_1 \cup \mathsf{lab}_2 \cup \{(a_{\text{fork}}, \mathtt{skip}), (a_{\text{join}}, \mathtt{skip})\} \wedge$
$\mathtt{po} = \mathtt{po}_1 \cup \mathtt{po}_2 \cup (\{a_{\text{fork}}\} \times (\mathcal{A}_1 \cup \mathcal{A}_2)) \cup ((\mathcal{A}_1 \cup \mathcal{A}_2) \times \{a_{\text{join}}\}) \cup \{(a_{\text{fork}}, a_{\text{join}})\}$
$\wedge (r_1, \mathcal{A}_1, \mathsf{lab}_1, \mathtt{po}_1) \vdash E_1 \wedge (r_2, \mathcal{A}_2, \mathsf{lab}_2, \mathtt{po}_2) \vdash E_2 \wedge$
$(\bot \in \{r_1, r_2\} \rightarrow r = \bot) \wedge (\bot \notin \{r_1, r_2\} \rightarrow r \neq \bot)$

Since every pre-execution which represents a program is guaranteed to have the po-smallest, and the po-greatest element in $\mathcal{A}$, we denote those elements as $\mathtt{po}_{\min}(\mathcal{A})$ and $\mathtt{po}_{\max}(\mathcal{A})$, respectively.

Note that the pre-executions are still a very rudimentary semantic representation of programs. Local computations and stores generate appropriate return values. Conditional branches and loops respect the recorded return values and take the appropriate branches. However, loads are allowed to read arbitrary values, which puts us far from a complete description of program semantics. In order to fully describe the behavior of loads (i.e., to specify which values are they allowed to read) we have to enrich the pre-executions with some additional structure.

In the rest of the discussion we will often need to talk about certain kinds of nodes within a pre-execution, such as all the nodes representing stores, or all the nodes representing

atomic accesses. The following two definitions summarize the notations used for this purpose. Definition 2.5 lists the functions which allow us to talk about the access type, the location accessed, and the value read/written by events representing memory accesses. Definition 2.6 gives names to some important subsets of the set of events in a pre-execution.

Definition 2.5 Let $(r, \mathcal{A}, \mathsf{lab}, \mathrm{po})$ be a pre-execution. The functions $\mathtt{loc}$, $\mathtt{val_r}$, $\mathtt{val_w} : \mathcal{A} \to \mathsf{Val} \cup \{\bot\}$, and $\mathtt{typ} : \mathcal{A} \to \{\mathrm{na}, \mathbf{rlx}, \mathbf{rel}, \mathbf{acq}, \mathbf{acq_rel}, \bot\}$ are given by

$$\mathtt{loc}(a) := \begin{cases} \ell & \text{if } \exists v, v', \tau.\, \mathsf{lab}(a) \in \{\mathtt{R}_\tau(\ell, v), \mathtt{W}_\tau(\ell, v), \mathtt{U}_\tau(\ell, v, v')\}, \\ \bot & \text{otherwise;} \end{cases}$$

$$\mathtt{val_r}(a) := \begin{cases} v & \text{if } \exists \ell, v', \tau.\, \mathsf{lab}(a) \in \{\mathtt{R}_\tau(\ell, v), \mathtt{U}_\tau(\ell, v, v')\}, \\ \bot & \text{otherwise;} \end{cases}$$

$$\mathtt{val_w}(a) := \begin{cases} v & \text{if } \exists \ell, v', \tau.\, \mathsf{lab}(a) \in \{\mathtt{W}_\tau(\ell, v), \mathtt{U}_\tau(\ell, v', v)\}, \\ \bot & \text{otherwise;} \end{cases}$$

$$\mathtt{typ}(a) := \begin{cases} \tau & \text{if } \exists \ell, v, v'.\, \mathsf{lab}(a) \in \{\mathtt{R}_\tau(\ell, v), \mathtt{W}_\tau(\ell, v), \mathtt{U}_\tau(\ell, v, v'), \mathtt{F}_\tau\}, \\ \bot & \text{otherwise.} \end{cases}$$

Definition 2.6 Let $(r, \mathcal{A}, \mathsf{lab}, \mathrm{po})$ be a pre-execution. We introduce notation for the following subsets of $\mathcal{A}$:

$$\begin{aligned}
\mathtt{NA} &:= \{a \in \mathcal{A} \mid \mathtt{typ}(a) = \mathrm{na}\} \\
\mathtt{Rel} &:= \{a \in \mathcal{A} \mid \mathtt{typ}(a) \in \{\mathbf{rel}, \mathbf{acq_rel}\}\} \\
\mathtt{Acq} &:= \{a \in \mathcal{A} \mid \mathtt{typ}(a) \in \{\mathbf{acq}, \mathbf{acq_rel}\}\} \\
\mathtt{W} &:= \{a \in \mathcal{A} \mid \exists \ell, v, \tau.\ \mathsf{lab}(a) = \mathtt{W}_\tau(l, v)\} \\
\mathtt{W_{at}} &:= \mathtt{W} \setminus \mathtt{NA} \\
\mathtt{W}_\ell &:= \{a \in \mathtt{W} \mid \mathtt{loc}(a) = \ell\} \\
\mathtt{R} &:= \{a \in \mathcal{A} \mid \exists \ell, v, \tau.\ \mathsf{lab}(a) = \mathtt{R}_\tau(l, v)\} \\
\mathtt{R_{at}} &:= \mathtt{R} \setminus \mathtt{NA} \\
\mathtt{R}_\ell &:= \{a \in \mathtt{R} \mid \mathtt{loc}(a) = \ell\} \\
\mathtt{U} &:= \{a \in \mathcal{A} \mid \exists \ell, v, v', \tau.\ \mathsf{lab}(a) = \mathtt{U}_\tau(l, v, v')\} \\
\mathtt{U}_\ell &:= \{a \in \mathtt{U} \mid \mathtt{loc}(a) = \ell\} \\
\mathtt{F} &:= \{a \in \mathcal{A} \mid \exists \tau.\ \mathsf{lab}(a) = \mathtt{F}_\tau\}
\end{aligned}$$

With notational necessities out of the way, let us get back to building executions.

In order to get from pre-executions to executions, we are going to add two additional types of edges to the pre-execution graphs. In addition to the *program-order* (po) edges contained in pre-executions, executions contain *reads-from* (rf) edges, and *modification-*

order ($\mathtt{mo}$) edges. Reads-from edges tell us which store read from which load; while modification-order edges give a total order on all the stores to a single location, providing us with a coherent linear history of modifications for each of the locations accessed by a program.

Definition 2.7 (Execution) A tuple $(r, \mathcal{A}, \mathsf{lab}, \mathrm{po}, \mathtt{rf}, \mathtt{mo})$ is an *execution* if

- $(r, \mathcal{A}, \mathsf{lab}, \mathrm{po})$ is a pre-execution,
- $\mathtt{rf} \subseteq [\mathtt{W} \cup \mathtt{U}]; =_{\mathtt{loc}}; [\mathtt{R} \cup \mathtt{U}]$,
- $\mathtt{val}_{\mathtt{w}}(a) = \mathtt{val}_{\mathtt{r}}(b)$ for every $(a, b) \in \mathtt{rf}$,
- $\mathtt{rf}^{-1}$ is functional,
- $\mathtt{mo} \subseteq [\mathtt{W} \cup \mathtt{U}]; =_{\mathtt{loc}}; [\mathtt{W} \cup \mathtt{U}]$, and
- for every location $\ell \in \mathsf{Loc}$, $[\mathtt{W}_\ell \cup \mathtt{U}_\ell]; \mathtt{mo}; [\mathtt{W}_\ell \cup \mathtt{U}_\ell]$ is a strict total order on $\mathtt{W}_\ell \cup \mathtt{U}_\ell$.

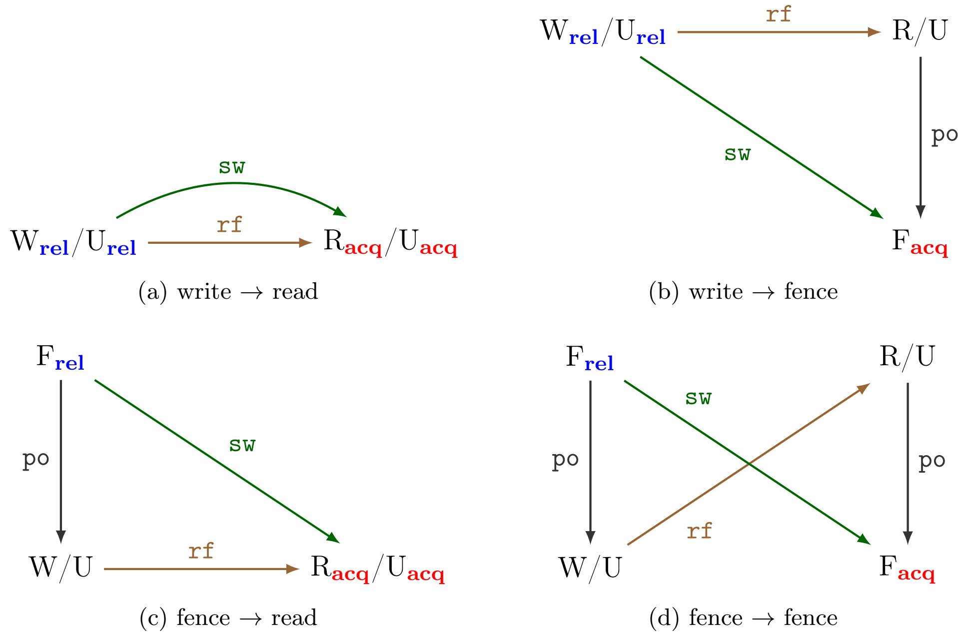
We can now use the rich structure of executions to express which executions represent "sensible" program behaviors. For example, our notion of "sensible" should include the behavior illustrated in the (SB) example from the introduction, while it should disallow the outcome $\mathtt{a} = 1 \land \mathtt{b} = 1$ for the (LB+DEP) program shown in Fig. 1.2.

In order to formally specify which behaviors should be allowed or disallowed, we will define several additional types of edges on the execution graph. All of these edges are relations that can be derived from the three basic relations (program-order, reads-from, and modification-order).

Definition 2.8 (Derived relations) On an execution $(r, \mathcal{A}, \mathsf{lab}, \mathrm{po}, \mathtt{rf}, \mathtt{mo})$, we define the following *derived relations*.

- reads-before: $\mathtt{rb} \coloneqq (\mathtt{rf}^{-1}; \mathtt{mo}) \setminus [\mathcal{A}]$,
- extended-coherence-order: $\mathtt{eco} \coloneqq (\mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb})^{+}$,
- release-sequence: $\mathtt{rs} \coloneqq [\mathtt{W}_{\mathtt{at}} \cup \mathtt{U}]; (\mathtt{rf}; [\mathtt{U}])^{*}$,
- synchronizes-with: $\mathtt{sw} \coloneqq [\mathtt{Rel}]; ([\mathtt{F}]; \mathrm{po})^{?}; \mathtt{rs}; \mathtt{rf}; [\mathtt{R}_{\mathtt{at}} \cup \mathtt{U}]; (\mathrm{po}; [\mathtt{F}])^{?}; [\mathtt{Acq}]$,
- happens-before: $\mathtt{hb} \coloneqq (\mathrm{po} \cup \mathtt{sw})^{+}$.

A load (or update) r *reads-before* a store (or update) w if w comes modification-order-after the store (or update) that r read from. Stated more intuitively, for a given load, reads-before tells us which stores reach the shared memory after the one the load got its value from. For technical reasons, we make sure that read-before is irreflexive—any atomic update in a "sensible" execution will end up reading from a store (or update) that is modification-order before itself, but including such reflexive pairs would unnecessarily complicate further definitions.



(a) write → read

(b) write → fence

(c) fence → read

(d) fence → fence

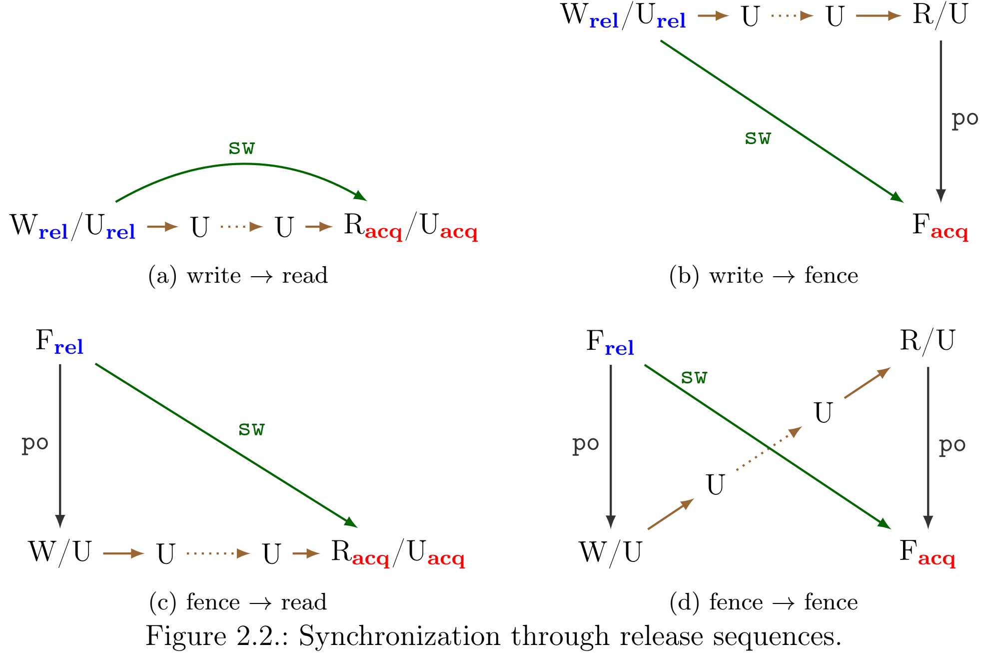
Figure 2.1.: Basic release-acquire synchronization.

The *extended-coherence-order* is a relation stating that two events are loosely ordered one before the other. This order combines three distinct notions of "being before": reads-from (a store being read comes before a load reading from it), modification-order (stores to the same location arrive in a particular order), and reads-before described above.

Happens-before is the notion of strong ordering between the actions of a program. Intuitively, an event a happening before an event b means that the execution of the event a completed before the execution of the event b started. Clearly, if two events are ordered by the program-order (i.e., they execute in a certain sequence within the same thread), they should be ordered accordingly in happens-before. The tricky part is the question of when do events from different threads get ordered by happens-before, and the answer is that the only way to get happens-before to connect two threads is if *synchronization* happens between the two threads.

The idea of synchronization, the most intricate part of the C11 model, is formally described by the *synchronizes-with* relation. In simple terms, synchronization always connects a release event with an acquire event, and always happens as a consequence of a read. In Fig. 2.1a 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. 2.1, relaxed accesses are helped along by fences in order to achieve synchronization.

To see why is synchronization defined this way, consider the release constructs as barriers which prevent instructions from being delayed past the barrier, and conversely the acquire constructs prevent instructions that follow the acquire to be executed before the acquire instruction has been executed. With this in mind, it should be intuitive that



(a) write → read

(b) write → fence

(c) fence → read

(d) fence → fence

Figure 2.2.: Synchronization through release sequences.

if there was communication (in the form of reads-from relation) between a release event and an acquire event, then the events preceding the release event should happen before the events which follow after the acquire event.

The reasoning above requires a direct communication line between two threads in order to achieve synchronization. However, that requirement can be relaxed if we consider a potential effect of CAS instructions. Since each successful CAS instruction has to read from its immediate predecessor in the modification order, we can see a chain of successful CAS instructions which in turn read from each other as a way of indirect inter-thread communication. If such a chain were to extend between a release and an acquire event, we should consider them synchronized as well. This idea is captured by the notion of *release sequences*, where a release sequence is a chain of successful CAS-es described earlier. This mode of synchronization is depicted in Fig. 2.2: the general synchronization pattern remains the same as in Fig. 2.1, with reads-from edges replaced by release sequence chains.

Remark (About the definition of release sequences) The original C11 memory model (ISO/IEC, 2011a,b), formed release sequences not only by atomic update chains, but also allowed a sequence of loads from the same location, all belonging to the same thread, to precede the chain of updates. The original definition had some unintended consequences (Vafeiadis et al., 2015, §4.3) pertaining to the sequence of loads coming before the chain of updates, which prompted proposals (Vafeiadis et al., 2015, §4.3; Lahav et al., 2017, §3.4) to strengthen the guarantees provided by the release sequences.

In the end, the official model (ISO/IEC, 2020) dropped the sequence of reads from the definition of release sequences, aligning the standard with the definition presented above.

Armed with all these relations, we can finally formalize the notion of what makes executions "sensible", which we do in the following definition of *consistency*. The idea is to give the semantics of programs in terms of consistent executions.

Definition 2.9 (Consistency) An execution $(r, \mathcal{A}, \mathsf{lab}, \mathtt{po}, \mathtt{rf}, \mathtt{mo})$ is *consistent* if the following conditions hold.

- $\forall a, b \in \mathcal{A}, \ell \in \mathsf{Loc}.\, \mathsf{lab}(a) = \mathsf{lab}(b) = \mathtt{A}(\ell) \to a = b$
- $\mathrm{rng}([\mathtt{W} \cup \mathtt{U}]; \mathtt{rf}^?; \mathtt{hb})|_{\mathtt{loc}} \subseteq \mathrm{rng}\, \mathtt{rf}$
- $\mathtt{hb}; \mathtt{eco}^?$ is irreflexive
- $[\mathtt{U}] \cap (\mathtt{rb}; \mathtt{mo}) = \emptyset$
- $(\mathtt{po} \cup \mathtt{rf})^+$ is irreflexive

Since we are not modeling deallocation, we can simply require that each location gets allocated at only one point in the whole execution. That is what the first condition formally states.

The second condition limits which loads are allowed to be uninitialized. Intuitively, a load can be uninitialized if it is not necessarily "aware" of a store to the same location. The notion of "being aware of a store" is formalized by saying that there is a happens-before-earlier store to the same location, or a happens-before-earlier initialized load from the same location.

The third condition (irreflexivity of $\mathtt{hb}; \mathtt{eco}^?$) tells us that the happens-before relation has no cycles, and that it does not contradict the weaker notion of "being before" given by extended-coherence-order. An important consequence of this condition is that programs with only one shared location are sequentially consistent.

The fourth condition ensures atomicity of updates by requiring the updates to read from their immediate predecessors in modification-order. For a more detailed explanation of this condition, see the last example in Section 2.4.

The last condition eliminates the so-called "out-of-thin-air" behaviors, such as the undesirable behavior of (LB+DEP) program from Fig. 1.2. Admittedly, this restriction is somewhat heavy-handed, as it also forbids the two observable behaviors shown in Fig. 1.2. Justification for this condition is further discussed in Section 2.5.

Definition 2.10 (Program executions) A consistent execution $(r, \mathcal{A}, \mathsf{lab}, \mathtt{po}, \mathtt{rf}, \mathtt{mo})$ is an execution of a program E if $(r, \mathcal{A}, \mathsf{lab}, \mathtt{po}) \vdash E$.

We now know which executions correspond to which programs, but we cannot simply proclaim that the semantics of a program is given by its consistent executions. Intuitively, the notion of consistency describes the constraints on the environment[1] used to run our programs. In effect, we are saying that the execution environment will produce only consistent executions.

[1] Think: compiler used to compile the code and hardware on which the resulting binary is being executed.

However, insisting that the environment produces consistent executions (and nothing else) for every program puts too heavy of a burden on the environment. In fact, programs might be faulty due to programmers' errors, such as data races, attempts to access unallocated locations, or read uninitialized variables. In case of programmers' errors, we want our semantics to put the blame for the program's misbehavior on the programmer, and not on the environment.

We are going to declare that the programs which commit data races, read from uninitialized locations, or commit memory faults are "ill-formed", and (in the true C fashion) simply say that the ill-formed programs have undefined behavior. For well-formed programs, the semantics will be given by the corresponding consistent executions.

Let us now give the formal definitions of the ways in which a program can be faulty as a result of a programmer's error.

First, we deal with accessing unallocated locations. We do this via the notion of memory safety: one is only allowed to access locations after they have been allocated. The notion of "after" is, of course, provided by the happens-before relation.

Definition 2.11 (Memory-safety) An execution $(r, \mathcal{A}, \mathsf{lab}, \mathtt{po}, \mathtt{rf}, \mathtt{mo})$ is *memory-safe* if

$$\forall a \in \mathtt{R} \cup \mathtt{W} \cup \mathtt{U}.\ \exists b \in \mathcal{A}.\ \mathtt{hb}(b, a) \wedge \mathsf{lab}(b) = \mathtt{A}(\mathtt{loc}(a))\,.$$

A program is memory-safe if all of its consistent executions are memory-safe.

Next, we turn our attention to the problem of reading from uninitialized locations. The notion of which load reads from which store is encoded in the reads-from relation, so we can simply say that the uninitialized reads are those which are unaccounted for by the reads-from relation.

Definition 2.12 (Uninitialized reads) A program E has *uninitialized reads* if there is a consistent execution $(r, \mathcal{A}, \mathsf{lab}, \mathtt{po}, \mathtt{rf}, \mathtt{mo})$ of E such that $\operatorname{rng} \mathtt{rf} \neq \mathtt{R} \cup \mathtt{U}$.

Lastly, we formalize the notion of data races. This is the subtlest of the three sources of programmer's errors we are considering, because we do not want to simply declare all the happens-before-unordered accesses to the same location to be bad. Recall that the atomic locations are supposed to be used in order to establish synchronization between threads, so concurrent atomic accesses should not be seen as problematic. However, non-atomic accesses are not supposed to be used in a racy way. Therefore, we are going to declare racy access patterns that contain non-atomic accesses to be undesirable.

Definition 2.13 (Conflicting events) Let $(r, \mathcal{A}, \mathsf{lab}, \mathtt{po}, \mathtt{rf}, \mathtt{mo})$ be an execution. We say that two events $a, b \in \mathcal{A}$ are *conflicting* if $a \neq b$, $\{a, b\} \cap (\mathtt{W} \cup \mathtt{U}) \neq \emptyset$, and $\mathtt{loc}(a) = \mathtt{loc}(b)$.

Definition 2.14 (Data races) We say that an execution $(r, \mathcal{A}, \mathsf{lab}, \mathtt{po}, \mathtt{rf}, \mathtt{mo})$ contains a *non-atomic data race* if there are conflicting events $a, b \in \mathcal{A}$ such that $\{a, b\} \cap \mathtt{NA} \neq \emptyset$ and $(a, b) \notin \mathtt{hb} \cup \mathtt{hb}^{-1}$. A program E is called *racy* if there is a consistent execution of E which contains a non-atomic data race.

After formalizing the notions behind potential programmer's errors, we can give the formal notion of a well-formed program.

Definition 2.15 (Well-formed programs) A program E is *well-formed* if it is memory-safe, has no uninitialized reads, and is not racy.

Finally, we are able to give the definition of the program semantics.

Definition 2.16 (Semantics of programs) An execution X *models* a program E (notation $X \models E$) if X is a consistent execution of E, or E is not well-formed.

Definition 2.16 tells us that the semantics of well-formed programs is given by their consistent executions, while the semantics of ill-formed programs is simply given by all possible executions. So, as we discussed above, given a well-formed program, the execution environment is going to produce a consistent execution corresponding to the program, while we place absolutely no constraints on what kind of an execution can be produced for ill-formed programs.

2.4. Examples

2.4.1. Store Buffering

As the first example, we are going to look at the (SB) program from the introduction. Of course, we need to rewrite the code from the introduction to make it conform to Definition 2.1. Strictly following the definition, the store buffering example looks as follows.

$$
\begin{array}{c}
\textbf{let } x = \textbf{alloc}() \textbf{ in} \\
\textbf{let } y = \textbf{alloc}() \textbf{ in} \\
[x]_{\mathbf{rlx}} = 0; \\
[y]_{\mathbf{rlx}} = 0; \\
\begin{array}{l||l}
[x]_{\mathbf{rlx}} = 1; & [y]_{\mathbf{rlx}} = 1; \\
[y]_{\mathbf{rlx}} & [x]_{\mathbf{rlx}}
\end{array}
\end{array}
$$

We are interested in the possible values that can be seen by the read instructions at the end of each thread. The semantics of the program is given by its consistent executions, all of which are listed in Fig. 2.3. As we can see, the possible outcomes include the three outcomes allowed by the thread-interleaving semantics of sequential consistency, as well as the additional outcome where both threads read the value 0.

It is interesting to note that the execution-based semantics is more concise than the interleaving-based semantics. For this particular example, according to sequential consistency, there are six interleavings which produce three observationally distinct outcomes; while the C11 model produces only four distinct executions, each accounting for one observable behavior.

In general, we will not have the one-to-one correspondence of executions and observable behaviors, but the executions will always be a more concise representation than interleavings are, simply because, by design, the executions do not order the events which do not need to be ordered in any particular way.

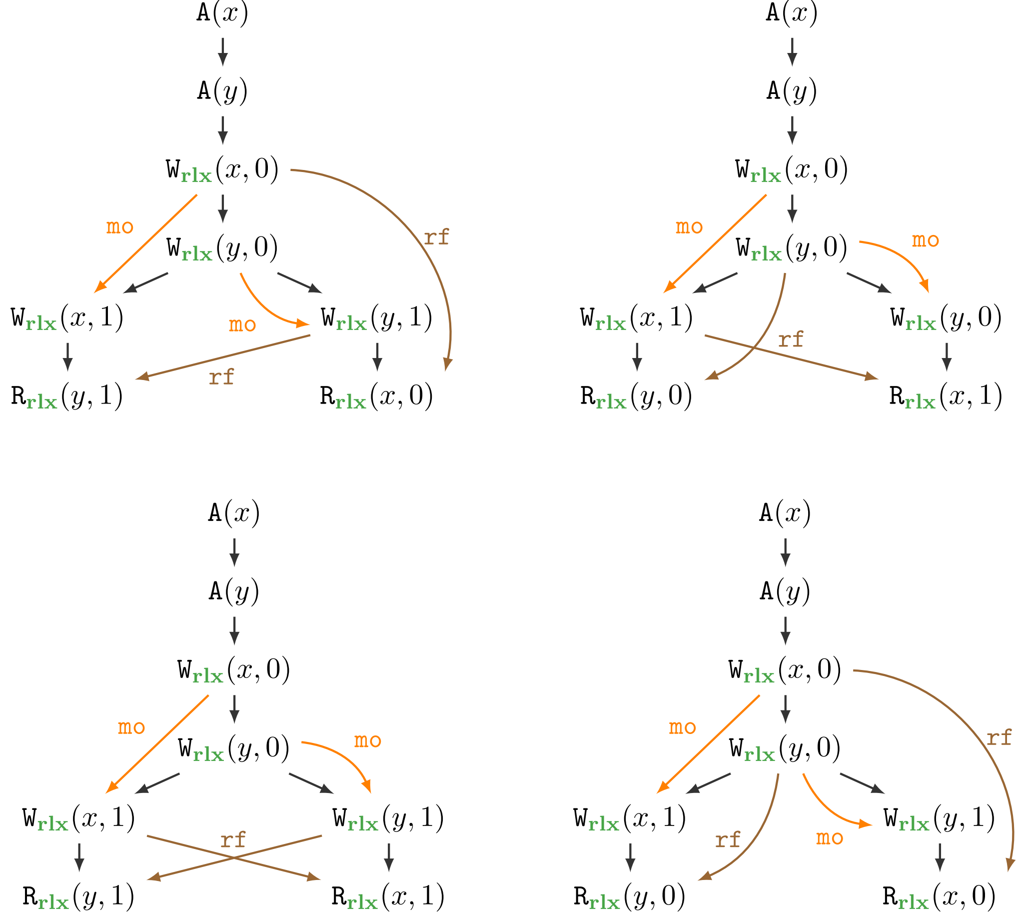


Figure 2.3.: Executions of the store buffering example. The unlabeled edges represent the program-order relation.

In the future examples, we are going to be more lax when presenting the programs. In particular, we are going to omit allocation and simply assume that all relevant locations have been properly allocated, and we will allow thread-local variables to appear without being bound by let expressions, as long as the context makes the meaning unambiguous. With these conventions, the store buffering example can be written as

$$
\begin{array}{c}
[x]_{\mathbf{rlx}} = 0; \\
[y]_{\mathbf{rlx}} = 0; \\
\begin{array}{l|l}
[x]_{\mathbf{rlx}} = 1; & [y]_{\mathbf{rlx}} = 1; \\
a = [y]_{\mathbf{rlx}} & b = [x]_{\mathbf{rlx}}
\end{array}
\end{array}
$$

and we would allow ourselves to talk about the values of the "local registers" a and b when describing the behavior of the program. We will also not show the modification-order relation in the execution graphs if it is not needed to explain the behavior in question.

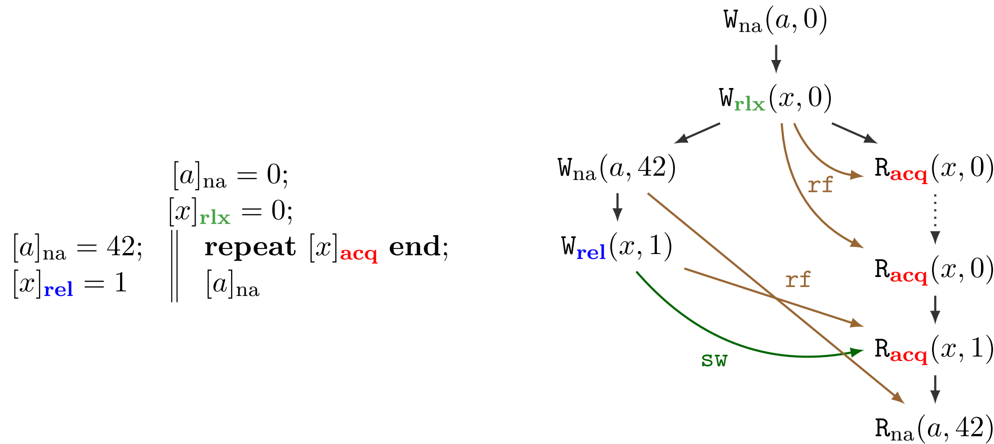


Figure 2.4.: Message passing via release write/acquire read synchronization.

2.4.2. Message Passing

After revisiting store buffering example, let us look at some examples showing how to utilize the synchronization mechanisms provided by the model. For this purpose, we are going to use the message passing pattern in which one thread prepares a certain resource, and then sets a flag notifying the other thread that the resource is ready. The second thread waits for the flag, and after observing the flag to be set proceeds to use the resource.

The simplest way to implement the message passing pattern is shown in Fig. 2.4, along with the summary of all the terminating executions of the program. The location a plays the role of the resource being exchanged between the threads, and x acts as the flag through which the two threads communicate. Note how the "regular data" (i.e., the value stored at a) is accessed non-atomically, while the synchronization flag x is accessed using atomic accesses.

Initially both a and x are set to 0. The left thread sets a to 42 and informs the other thread about this change by doing a release store of 1 to x. The right thread spins in a loop, making acquire loads from x until it observes the value 1. Once it observes x to be set to 1, the second thread proceeds to load the value stored in a.

All terminating executions of the message passing program contain the two events which initialize a and x to 0; the left thread is represented by the two write events setting a to 42 and x to 1; and the right thread generates a (possibly zero-length) sequence of read events which read the initial value of x, followed by a read which sees the updated value of x, and terminates with a read from a.

The acquire read event in the right thread which reads from the left thread's release write establishes synchronization, which in turn guarantees that the read from a issued by the right thread sees the value 42 set by the left thread. To see how, consider the inconsistent execution of our message passing program shown in Fig. 2.5. By reading from the initializing write to a, the right thread induces a reads-before edge between its read from a and the write to a executed by the left thread. This read-before edge, together with the

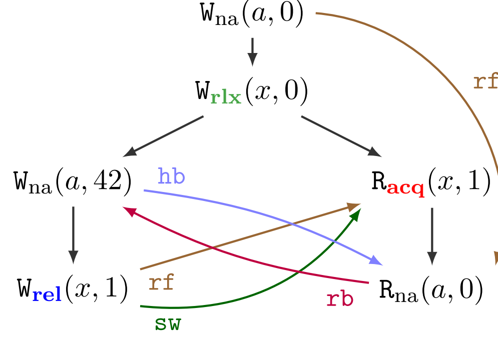


Figure 2.5.: An inconsistent execution of the message passing program from Fig. 2.4. The cycle in the hb; rb relation violates the definition of consistency.

happens-before relation creates a cycle forbidden by the third condition in Definition 2.9.

Looking at the executions of the message passing program from Fig. 2.4, we can see that the waiting loop is repeatedly executing an acquire read while it waits for the flag to be set. Intuitively, this seems needlessly expensive as we are attempting to establish synchronization in every loop iteration even though we know that only the final attempt will succeed in creating a synchronization edge. This intuition also holds true in practice because (on most architectures) acquire load instructions get compiled into more expensive constructs than relaxed load instructions which do not attempt to synchronize.

Avoiding needless synchronization attempts is possible by utilizing memory fences. Figure 2.6 shows a variant of the message passing program where the waiting loop performs relaxed loads and the synchronization is achieved by placing an acquire fence after the loop. The synchronization pattern which guarantees that the right thread observes the correct value of a is preserved while executing only a single acquire instruction.

It is also possible to decouple the release store into a release fence followed by a relaxed store, as shown in Fig. 2.7. Compared to the previously discussed decoupling of acquire loads into relaxed loads followed by an acquire fence, the decoupling of the release store does not provide any performance benefits for the message passing program. In a more complex situation, replacing a sequence of release stores by a single release fence followed by a sequence of relaxed stores would be beneficial for performance.

We conclude the discussion of the message passing pattern by looking at how the C11 semantics treats programs which do not provide enough synchronization. One such program is given in Fig. 2.8. In the waiting loop, the acquire loads have been replaced by relaxed loads, but the acquire fence is missing. The lack of proper synchronization gives rise to consistent racy executions (the non-atomic write to a in the left thread and the non-atomic read from a in the right thread are not ordered by happens-before), which means the program has undefined behavior (i.e., the semantics provides absolutely no guarantees about the behavior of the program).

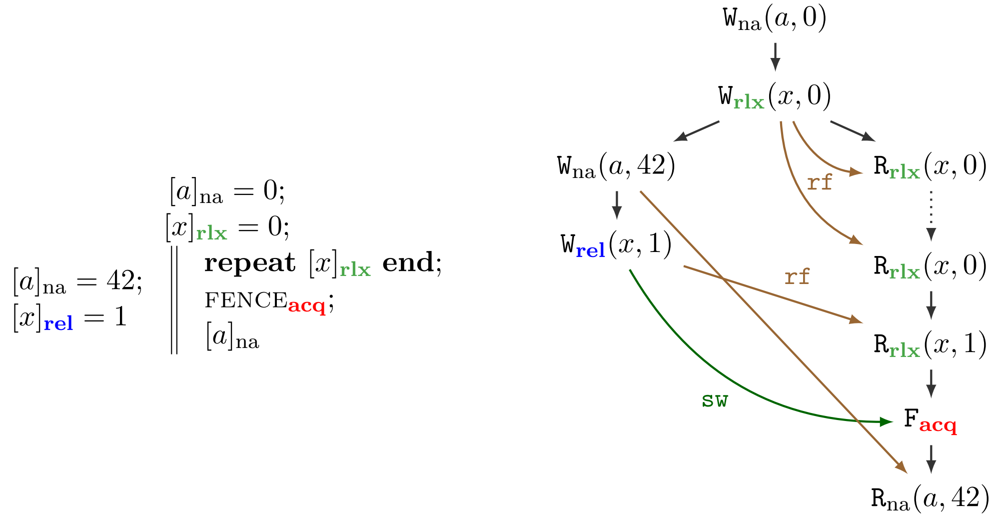


Figure 2.6.: Message passing via release write/acquire fence synchronization.

2.4.3. CAS Atomicity

For the last example we turn our attention to compare-and-swap instructions. In the program from Fig. 2.9 we see two concurrent CAS instructions trying to update the value of x from 0 to 1. Clearly, any sensible semantics will ensure that the at most one of the CAS instructions succeeds. So, how does C11 provide that guarantee?

Consider an execution in which both CAS instructions succeed. In that case both CAS instructions generate update events (labeled b and c in Fig. 2.9), and both of them read from the write event (labeled a) which initialized x to 0. The modification-order cannot contradict the program-order (due to the third clause in Definition 2.9), so $\mathtt{mo}(a, b)$ and $\mathtt{mo}(a, c)$ is necessary, while the modification-order between the events b and c is unconstrained. Because the situation is symmetric, we can, without loss of generality, assume $\mathtt{mo}(b, c)$. Now we see that the update event c reads-before the event b, which means $(c, c) \in \mathtt{rb}; \mathtt{mo}$, and that situation is forbidden by the fourth clause in Definition 2.9. This forces us to conclude that there are no consistent executions in which both concurrent CAS instructions succeed.

In general, an $\mathtt{rb}; \mathtt{mo}$ cycle as the one discussed above will appear whenever there is an update event which does not read from its immediate $\mathtt{mo}$ predecessor,

2.5. The Out-of-Thin-Air Problem

In the previous section, we looked at some examples of programs and discussed how the model presented in this chapter gives them appropriate semantics. We opened the section with the store buffering example from the introduction, proceeded to look at several other programs, but were strangely silent about the other example brought up in the

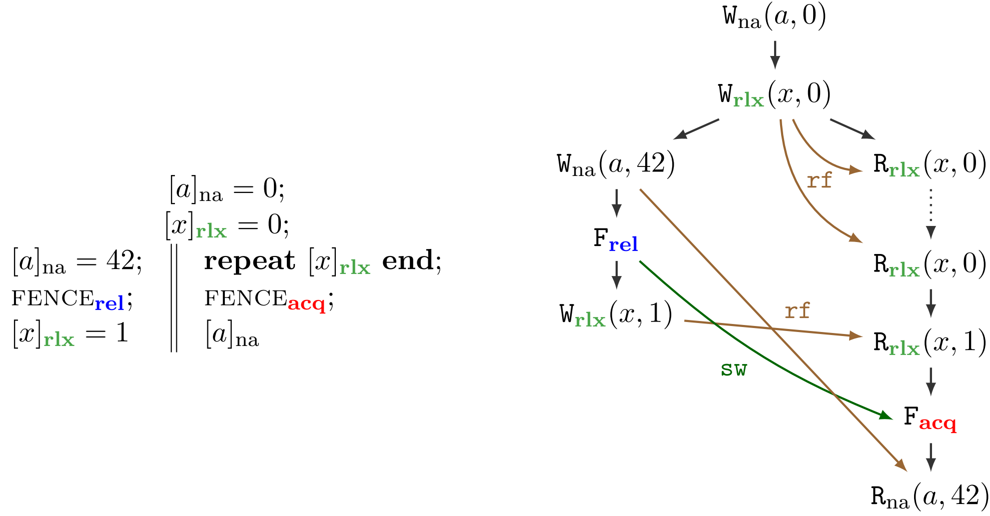


Figure 2.7.: Message passing via release fence/acquire fence synchronization.

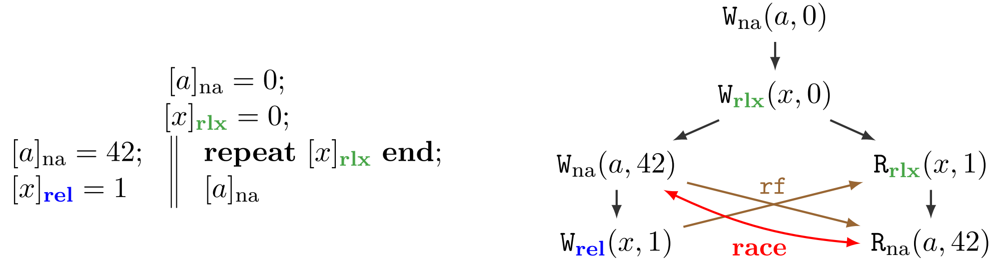


Figure 2.8.: Message passing program without proper synchronization is racy.

introduction. The discussion about the load buffering example was not present in the previous section because load buffering (one might say unfortunately) deserves a section for itself. This is that section. Let us talk about the load buffering.

Back in Fig. 1.2 we saw three programs all of which access two shared locations in a similar pattern. The left thread reads from y and writes to x, and the right thread reads from y and writes to x. The difference between the programs is in which exact stores they execute. In the classic load buffering program both threads write the value 1. The load buffering program with fake dependency does the same thing as the classic load buffering, but the threads "pretend" to use the value they obtained by the loads. Finally, in load buffering with data dependency each thread copies the value it read to the location it is writing to.

The behavior we are interested in, for each of the three variants, is: *can both load instructions see the value 1?*

Even though at the first glance one might answer with *"of course not"*, upon deeper inspection the situation is not so simple. Several modern architectures (e.g., Power, ARM) allow the behavior in question for the classic load buffering program. Therefore, having a

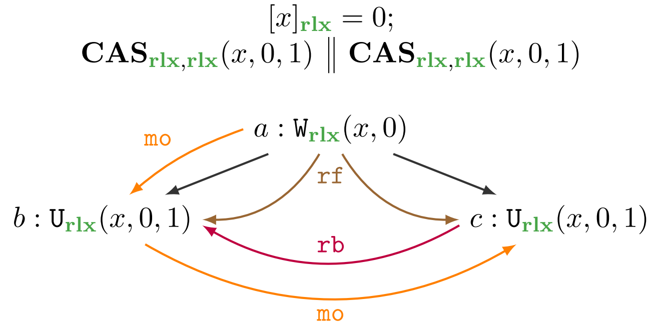


Figure 2.9.: Two concurrent CAS instructions cannot both succeed. The depicted execution is inconsistent because $(c, c) \in [\mathtt{U}] \cap (\mathtt{rb}; \mathtt{mo})$.

model which allows both reads in the classic load buffering program to read 1 is desirable, since that would correspond to a behavior observed in practice.

Similarly, the ideal model should also allow the same bahavior for the load buffering with fake dependency. The reason is simple: any optimizing compiler will optimize the expressions `1 + a - a` and `1 + b - b` replacing them by `1`, thus transforming the program into the classic load buffering, for which we already established that both threads should be allowed to read 1.

The load buffering program with data dependency is different. In this program the initial values of x and y just get copied over each other. The only value that appears in the program is 0, and the value 1 appearing out of nowhere does not seem reasonable at all. As expected, no compiler and hardware combination will conspire to produce values out of thin air, so anything besides the reads of 0 cannot be observed in practice for the load buffering with data dependency.

To summarize, the ideal memory model which aims to encapsulate compiler and hardware optimizations should allow both threads to read 1 in the classic load buffering and the load buffering with fake dependency examples, while disallowing the same behavior for the load buffering with data dependency. In our framework that would mean looking at the executions which provide the behavior in question, and setting up the consistency definition in such a way that the executions belonging to the first two load buffering variants are deemed consistent while the executions coming from the variant with data dependency is declared inconsistent.

So, what do the executions in question look like? They are depicted in Fig. 2.10. Yes, the exact same execution is generated by all three load buffering variants! This means that our options are to allow the behavior we are considering for all the examples, disallow it for all the examples, or change the framework so that it can differentiate between the three examples.

The option to allow both threads to see the value 1 was taken by the C and C++ standardizing committees (ISO/IEC, 2011a,b) when first developing the C11 model as they did not wish an overluy restrictive model. (The standard does warn compiler

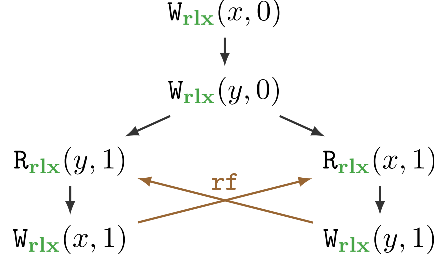


Figure 2.10.: The problematic load buffering execution.

writers not to exploit undesirable behaviors, although without specifying what those are.) Unfortunately, this is not a viable solution because, as noted by Batty et al. (2013); Vafeiadis and Narayan (2013), allowing programs to read values appearing out of thin air invalidates even the most basic reasoning principles.

Changing the framework can allow us to differentiate the three examples, but as shown by Batty et al. (2015), it is not possible to classify them in the desired way while maintaining the C11-style of per-execution memory model definition. The best that can be achieved is to introduce additional types of edges which would track syntactic dependencies, as is done by several hardware models (e.g., Power, ARM). This approach is unable to distinguish between fake and real data dependencies, so it allows the load buffering behavior for the classic example, while forbidding it for both the fake dependency and the real data dependency example. Restriction of this style is perfectly good for hardware models (because hardware is not performing arithmetic-based optimizations), but it is completely unacceptable for a language-level semantcis, because it would render unsound any compiler optimizations which perform simplification of expressions (as such simplification may remove syntactic dependencies).

Finally, we are left with the option to simply forbid the load buffering behavior in all its instances. Our definition of execution consistency (Definition 2.9) does it by requiring acyclicity of the $\texttt{po} \cup \texttt{rf}$ relation (i.e. irreflexivity of $(\texttt{po} \cup \texttt{rf})^+$). As we can see, the execution in Fig. 2.10 contains a $\texttt{po} \cup \texttt{rf}$ cycle and is thus deemed inconsistent.

Entirely forbidding the load buffering behavior does not come without drawbacks. Declaring a behavior observable on certain hardware inconsistent results in suboptimal compilation schemes for the affected platforms (Power and ARM), and a certain desirable compiler transformations become unsound.

The suboptimal compilation for Power and ARM is in regards to the compilation of relaxed accesses. The intended compliation of relaxed accesses in C-like languages is to compile them down to plain access instructions.[1] Unfortunately, in order to prevent the load buffering behavior, the relaxed loads have to be compiled to a load instruction followed by a “fake dependency” (i.e., a conditional branch to the next instruction). This compilation scheme forbids the hardware to perform load-to-store reorderings which cause

[1]Relaxed load/store should compile to a hardware load/store instruction, without any bells and whistles, such as fence instructions surrounding them.

the load buffering behavior. Note that this solution affects only relaxed loads and uses a trick which is not as heavy-handed as introducing fence instructions.

As noted above, certain load-to-store reorderings lead to the load buffering behavior. Therefore, compiler optimizations performing such reorderings are unsound under our model. In particular reordering of relaxed loads followed by relaxed stores ($[x]_{\mathbf{rlx}}; [y]_{\mathbf{rlx}} = v \leadsto [y]_{\mathbf{rlx}} = v; [x]_{\mathbf{rlx}}$) are not allowed. Again, note that this affects only relaxed accesses. Optimizations concerning other types of accesses (in particular, the non-atomic accesses which should be used to manipulate data, and will therefore be the most commonly used instructions in most programs) are completely unaffected.

The practical impact of requiring $\mathtt{po} \cup \mathtt{rf}$ to be acyclic as a way of eliminating out-of-thin-air values has been studied by Ou and Demsky (2018) who found the performance loss to be fairly minimal.[1] This work indicates that the solution we consider here might be not only good from the standpoint of restoring soundness of reasoning principles, but the performance trade-off could be acceptable enough to be considered for some practical use.

[1] They report no overhead on average and a maximum overhead of 6.3% on 43 concurrent data structures.

3. Fenced Separation Logic

This chapter presents fenced separation logic (FSL), a separation logic for reasoning about concurrent programs under the C11 memory model described in Chapter 2.

FSL is an extension of relaxed separation logic (RSL) (Vafeiadis and Narayan, 2013), which is in turn an extension of concurrent separation logic (CSL) (O'Hearn, 2007). This chapter presents FSL as an extension of CSL, without assuming any level of familiarity with RSL from the reader. Some level of background knowledge about CSL and separation logic in general is assumed (such as the notions of separating conjunction and the frame rule). For the comments on additions FSL makes on top of RSL, see Section 7.1.1.

First we present the syntax of FSL by going over all the inference rules, paying special attention to provide the intuitive feel underlying the novel assertions and rules stemming from the intricate way the C11 model treats atomic accesses and synchronization. We continue by presenting the formal semantics of FSL, and conclude with a high-level overview of the proof of soundness.

FSL has been mechanized using the Coq proof assistant (INRIA). See Appendix A for further details.

3.1. Syntax

We start by reviewing the grammar of FSL assertions, and how to read the FSL's version of Hoare triples.

FSL assertions (formally defined in Definition 3.2) contain all the usual quantifiers, logical connectives, and assertions one would expect to see in an extension of concurrent separation logic, together with several novel assertions which are there in order to support reasoning about various access types which exist in the C11 memory model.

The locations being accessed non-atomically are handled by the $\mathsf{Uninit}(\ell)$ assertion, representing an allocated, but uninitialized location ℓ, and the customary *points-to* assertion ($\ell \mapsto v$), representing non-atomic location ℓ which holds a value v. The FSL points-to assertion has an additional permission parameter, implementing the idea of *partial permissions* (Boyland, 2003; Bornat et al., 2005). We will talk about the permissions and how are they used in Section 3.1.2 where we present the inference rules for non-atomic accesses.

The assertions $\mathsf{Rel}(e, \mathcal{Q})$, $\mathsf{Acq}(e, \mathcal{Q})$, $\mathsf{RMWAcq}(e, \mathcal{Q})$, $\mathsf{Init}(e)$, and the two modalities, $\triangle$ and $\triangledown$, are there to reason about the atomic accesses, memory fences, and synchronization achieved by their interplay. See Sections 3.1.3 and 3.1.5 for the discussion on intended meaning behind those assertions, and how to intuitively think about them.

The assertion $\boxed{e : g}$ talks about the *ghost state*, i.e., data structures we use for reasoning,

which are not accessed by the program we are reasoning about. Section 3.1.7 is devoted to the discussion of the ghost state.

Definition 3.1 (Permission structures) Tuple $(M, \oplus, \varepsilon, \mathbb{1})$ is a *permission structure* if $(M, \oplus)$ forms a partial commutative monoid with ε as the neutral element, and $\mathbb{1} \in M \setminus \{\varepsilon\}$ is composable only with the neutral element, i.e., $\mathbb{1} \oplus q$ is undefined for every $q \in M \setminus \{\varepsilon\}$. The elements ε and $\mathbb{1}$ are called the *empty permission* and the *full permission*, respectively.

Definition 3.2 (FSL grammar) Let $(\mathbf{P}, +, \varepsilon, \mathbb{1})$ be a permission structure, 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, $p \in \mathbf{P} \setminus \{\varepsilon\}$, $g \in \mathbf{G}$, and $\mathcal{Q}$ is a mapping from values to assertions.

The common quantifiers and connectives not present in Definition 3.2 can be seen as syntactic sugar, e.g., $\neg P \equiv P \to \mathsf{false}$ and $\exists \equiv \neg\forall\neg$.

Hoare triples in FSL 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.

The triples should be read from the standpoint of *partial correctness*, meaning that $\{P\}\, E\, \{v.\, Q\}$ claims nothing about the termination of the program E, but rather the meaning of the triple is that starting from a state which satisfies the precondition P, the program E will run without any faults, and if it terminates, the final state will satisfy the postcondition Q.

3.1.1. Standard Rules

FSL, as an extension of CSL, supports all the standard rules coming from Hoare logic and concurrent separation logic. We list them here for the sake of completeness.

$$\{P\}\, v\, \{y.\, P \wedge y = v\} \quad (\text{VAL})$$

$$\frac{\{P\}\, E_1\, \{x.\, Q\} \qquad \forall x.\ \{Q\}\, E_2\, \{y.\, R\}}{\{P\}\, \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2\, \{y.\, R\}} \ (\text{LET}) \qquad \frac{\{P\}\, E\, \{y.\, Q\} \qquad y \notin \mathsf{fv}(R)}{\{P * R\}\, E\, \{y.\, Q * R\}} \ (\text{FRAME})$$

$$\frac{\{P \wedge b \neq 0\}\, E_1\, \{y.\, Q\} \qquad \{P \wedge b = 0\}\, E_2\, \{y.\, Q\}}{\{P\}\, \textbf{if } b \textbf{ then } E_1 \textbf{ else } E_2\, \{y.\, Q\}}\ (\textsc{if})$$

$$\frac{\{P\}\, E\, \{y.\, Q\} \qquad P' \implies P \qquad \forall y.\, (Q \implies Q')}{\{P'\}\, E\, \{y.\, Q'\}}\ (\textsc{conseq})$$

$$\frac{\{P\}\, E\, \{y.\, Q\} \qquad Q[0/y] \implies P}{\{P\}\, \textbf{repeat } E \textbf{ end}\, \{y.\, Q \wedge y \neq 0\}}\ (\textsc{loop})$$

$$\frac{\{P\}\, E\, \{y.\, Q\} \qquad \{P'\}\, E\, \{y.\, Q'\}}{\{P \vee P'\}\, E\, \{y.\, Q \vee Q'\}}\ (\textsc{disj})$$

$$\frac{\{P_1\}\, E_1\, \{Q_1\} \qquad \{P_2\}\, E_2\, \{Q_2\}}{\{P_1 * P_2\}\, E_1 \parallel E_2\, \{Q_1 * Q_2\}}\ (\textsc{par})$$

$$\frac{\{P\}\, E\, \{y.\, Q\}}{\{\exists x.\, P\}\, E\, \{y.\, \exists x\, .Q\}}\ (\exists\textsc{-intro})$$

Note that the (PAR) rule requires the postconditions Q_1 and Q_2 to be formulas which do not depend on the return value. This is needed because the parallel composition ignores the return value of its constituent expressions.

3.1.2. Rules for Non-atomic Accesses

Non-atomic accesses can be seen as the "regular" accesses, i.e., the accesses one would use for regular data manipulation. A non-atomic location ℓ can be uninitialized, represented by the assertion $\mathsf{Uninit}(\ell)$, or hold some known value v, represented by the usual points-to assertion $\ell \overset{p}{\mapsto} v$.

The parameter p in the points-to assertion is a permission. The permissions allow us to reason about programs featuring concurrent readers from a single non-atomic location. In order to do that, we need to be able to split a single points-to assertion into smaller constituent parts.

The most well known way of splitting the points-to assertion is by using the *fractional permissions* (Boyland, 2003), where the permissions take values from the interval $(0, 1]$, and splitting is done according to the equivalence

$$\ell \overset{p}{\mapsto} v * \ell \overset{q}{\mapsto} v \iff \begin{cases} \ell \overset{p+q}{\mapsto} v & \text{if } p + q \leqslant 1, \\ \mathsf{false} & \text{otherwise.} \end{cases}$$

FSL takes a generalized approach to permissions, which can come from any general permission structure (see Definition 3.1), and the splitting is done according to

$$\ell \overset{p}{\mapsto} v * \ell \overset{q}{\mapsto} v \iff \begin{cases} \ell \overset{p \oplus q}{\mapsto} v & \text{if } p \oplus q \text{ is defined,} \\ \mathsf{false} & \text{otherwise.} \end{cases} \qquad (\textsc{na-split})$$

One important consequence of (NA-SPLIT) is that if a thread owns a non-atomic location with the full permission ($\ell \overset{\mathbb{1}}{\mapsto} v$), then no other thread can concurrently access ℓ. Due to the special status of the full permission, we will commonly write $\ell \mapsto v$ instead of $\ell \overset{\mathbb{1}}{\mapsto} v$.

The inference rules governing the use of non-atomic relations follow the standard separation-logic rules for the points-to assertions.

When allocating a non-atomic location ℓ, we get a resource $\mathsf{Uninit}(\ell)$ telling us that we now have an uninitialized location ℓ.

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

Writing to a non-atomic location ℓ requires having the complete ownership of the location. The complete ownership can be in the form of owning the uninitialized location, or we can own the full permission to the initialized location. In either case, after the command has been executed, we still have the full ownership, and we know that the value of the location ℓ is the value we wanted to write to it.

$$\left\{\mathsf{Uninit}(\ell) \vee \exists v.\, \ell \overset{\mathbb{1}}{\mapsto} v\right\} [\ell]_{\mathrm{na}} = v \left\{\ell \overset{\mathbb{1}}{\mapsto} v\right\} \tag{W-NA}$$

If we have a partial ownership of an initialized non-atomic location ℓ, we can read from it, and the resulting value will be the value stored at the location.

$$\left\{\ell \overset{p}{\mapsto} v\right\} [\ell]_{\mathrm{na}} \left\{y.\, y = v \wedge \ell \overset{p}{\mapsto} v\right\} \tag{R-NA}$$

3.1.3. Basic Rules for Atomic Accesses

The atomic accesses are used for inter-thread communication, in order to provide proper synchronization among the threads of the program. In FSL, we think of threads owning certain resources that can be transferred from one thread to another at synchronization points. Think of the resources as if they are "flowing" down the execution graphs, and the only way from them to move from one thread to another is if there are some synchronization edges between the events of the two threads. The rules for atomic accesses are designed to describe the situations in which the synchronizations happen, and to facilitate the resource transfer along the synchronization edges.

Let us start by looking at the rule for allocating a new atomic location (i.e., a location which we intend to access using atomic accesses).

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

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 think of $\mathcal{Q}$ as a kind of an invariant, but instead of stating "if the location ℓ holds the value v, then the assertion $\mathcal{Q}(v)$ is true", the invariant $\mathcal{Q}$ is saying "by writing the value v to the location ℓ, the resource $\mathcal{Q}(v)$ is being transferred."

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. 2.1a and 2.1b), we can simply transfer the resource we want without any further complications. This is summarized in the following rule.

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

In the precondition, the assertion $\mathsf{Rel}(\ell, \mathcal{Q})$ grants us permission to write to the atomic location ℓ, and $\mathcal{Q}$ 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 ℓ, 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. Additionally, we get the assertion $\mathsf{Init}(\ell)$ stating that the location ℓ has been initialized.

The (W-REL) rule features a normalizability requirement in its premise (formally defined in Definition 3.11). For now, it is important to note that it is a technical requirement, not causing any trouble in practice. This is primarily reflected in the fact that all positive[1] formulas which do not contain $\triangle$ and $\triangledown$ modalities are normalizable.

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. 2.1c and 2.1d. 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.

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

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

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 function as end points of both resource transfer and synchronization (see Figs. 2.1a and 2.1c). For this reason, resource acquisition by acquire reads is quite simple.

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

In order to read from a location, we have to know that it has been initialized, and have a permission to perform the atomic read ($\mathsf{Acq}(\ell, \mathcal{Q})$). 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 $\mathcal{Q}(v)$. Note that the act of reading changed the read permission from the one we started with in the precondition. The notation $\mathcal{Q}[v{:=}\mathsf{emp}]$ denotes the function that is the same as $\mathcal{Q}$ for all the arguments except v, and takes on the value emp at v. This change in the read permission, which can be seen as "dropping" the permission to acquire

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

resources by reading the value v again, is there precisely to disallow double acquisition of the same resource, which would lead to inconsistencies.

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. 2.1b and 2.1d). Only after synchronization are we allowed to use the acquired resource. The following two rules represent this case.

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

$$\{\triangledown P\}\ \mathtt{fence}_{\mathbf{acq}}\ \{P\} \qquad \text{(F-ACQ)}$$

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.

Before we take a look at some examples of how the rules presented in this section are used, it is important to bring to attention some very important properties of the Init, Rel, and Acq assertions, which are vital in enabling concurrent accesses to atomic locations. The Init and Rel assertions are *duplicable*

$$\mathsf{Init}(x) \iff \mathsf{Init}(x) * \mathsf{Init}(x) \qquad \text{(INIT-SPLIT)}$$

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

while the Acq assertion is *splittable*

$$\mathsf{Acq}(\ell, \lambda v.\ \mathcal{Q}_1(v) * \mathcal{Q}_2(v)) \iff \mathsf{Acq}(\ell, \mathcal{Q}_1) * \mathsf{Acq}(\ell, \mathcal{Q}_2)\,. \qquad \text{(ACQ-SPLIT)}$$

Note that it is not a problem to allow multiple writers to have the same Rel permission, since in order to execute a read one must provide the resource requested by the invariant. Two concurrent writers will not be able to clash on which resources are being provided, since the logic ensures that concurrent events have disjoint resources available. On the other hand, the Acq permission needs to be split apart when being distributed to concurrent readers, because we cannot allow concurrent readers to acquire the same resource. (ACQ-SPLIT) ensures that each concurrent reader gets their own part of the shared resource.

3.1.4. Basic Examples

Message Passing

Now that we have the rules for handling atomic and non-atomic accesses, we can already reason about some interesting programs. Message passing programs, which we encountered back in Section 2.4.2, are ideal for showcasing how the rules discussed so far operate.

Please keep in mind that while discussing examples we will in principle implicitly apply the standard rules from Section 3.1.1. In particular, this means that we will not be pointing out when the (FRAME) and (CONSEQ) rules are being used.

$$\{\mathsf{emp}\}$$
$$a = \textbf{alloc}();$$
$$\{\mathsf{Uninit}(a)\}$$
$$x = \textbf{alloc}();$$
$$\{\mathsf{Uninit}(a) * \mathsf{Rel}(x, \mathcal{Q}) * \mathsf{Acq}(x, \mathcal{Q})\}$$
$$[x]_{\mathbf{rlx}} = 0;$$
$$\{\mathsf{Uninit}(a) * \mathsf{Rel}(x, \mathcal{Q}) * \mathsf{Acq}(x, \mathcal{Q}) * \mathsf{Init}(x)\}$$

Producer	Consumer
$\{\mathsf{Uninit}(a) * \mathsf{Rel}(x, \mathcal{Q})\}$	$\{\mathsf{Acq}(x, \mathcal{Q}) * \mathsf{Init}(x)\}$
$[a]_{\mathrm{na}} = 42;$	$\textbf{repeat } [x]_{\mathbf{acq}} \textbf{ end};$
$\{a \mapsto 42 * \mathsf{Rel}(x, \mathcal{Q})\}$	$\{v.\, v \neq 0 \land \mathsf{Acq}(x, \mathcal{Q}[v := \mathsf{emp}]) * \mathsf{Init}(x) * a \mapsto 42\}$
$[x]_{\mathbf{rel}} = 1;$	$[a]_{\mathrm{na}}$
$\{\mathsf{Rel}(x, \mathcal{Q}) * \mathsf{Init}(x)\}$	$\{r.\, r = 42 \land v \neq 0 \land \mathsf{Acq}(x, \mathcal{Q}[v := \mathsf{emp}]) * \mathsf{Init}(x) * a \mapsto 42\}$

$$\{\mathsf{true}\}$$
where $\mathcal{Q} = \lambda v.\, \textbf{if } v = 0 \textbf{ then } \mathsf{emp} \textbf{ else } a \mapsto 42.$

Figure 3.1.: Correctness proof for message passing via release write/acquire read synchronization program from Fig. 2.4.

In Fig. 3.1 we have the correctness proof for the simplest version of the message passing program (i.e., the one where the synchronization is achieved by an acquire load reading directly from a release store). The proof starts with applying (A-NA) to allocate the non-atomic location a, and proceeds with the application of (A-AT) to allocate the atomic location x.

When allocating x, we have to decide on a mapping $\mathcal{Q}\colon \mathsf{Val} \to \mathsf{Assn}$ which describes the protocol the threads follow when accessing x. Here, the producer thread on the left will initialize the location a, and set x to 1 to signal that it is safe to read from a. The consumer thread on the right waits for the signal before it reads a. We thus select the mapping $\mathcal{Q}$ such that $\mathcal{Q}(0) = \mathsf{emp}$, representing that x can always be freely set to 0 and no knowledge can be gained when seeing the value 0 at location x; and for $v \neq 0$ we set $\mathcal{Q}(v) = a \mapsto 42$, representing that in order to set x to a non-zero value one must have set a to 42, and anyone who sees a non-zero value at location x knows that a has been set to 42.

After allocating x, it immediately gets initialized to 0. Here, (W-RLX) rule is used, and looking closely, we see that besides having $\mathsf{Rel}(x, \mathcal{Q})$ in the precondition, in order to set x to 0, we should also have $\triangle\mathcal{Q}(0)$, which is apparently missing. However, $\mathcal{Q}(0) = \mathsf{emp}$ and $\triangle\mathsf{emp} \iff \mathsf{emp}$, so the precondition of (W-RLX) is satisfied. Intuitively, the equivalence $\triangle\mathsf{emp} \iff \mathsf{emp}$ holds because the $\triangle$ modality is there to prevent us from committing memory races by accessing the resource under the modality, but when the protected resource is the empty resource, the modality becomes inconsequential. Formally, the equivalence follows directly from the assertion semantics which will be given in Definition 3.10.

Coming to the point where the program forks into two threads, we give the ownership of the non-atomic location a and the permission to write to the atomic-location x to the producer thread, and the permission to read from x goes to the consumer thread.

The producer thread first sets a to 42 (here (W-NA) is applied), and then signals that a has been initialized by performing a release store to x.

Setting x to 1 uses the (W-REL) rule, which asks for $\mathsf{Rel}(x, \mathcal{Q})$ and $\mathcal{Q}(1)$ to be in the precondition. $\mathcal{Q}(1) = a \mapsto 42$, and a has just been set to 42, so that works out well. By setting x to 1, the producer thread gives up its ownership of the location a, which will be transferred to a thread that ends up reading the value the producer just wrote to x.

The consumer thread starts by repeatedly reading from x. Each pass through the loop utilizes the (R-ACQ) rule. As long as the value 0 is observed, the postcondition remains the same as the precondition because $\mathcal{Q}(0) = \mathsf{emp}$, so no resources are acquired by reading and $\mathcal{Q}([0{:=}\mathsf{emp}]) = \mathcal{Q}$. This reestablishes the loop invariant, letting us continue reading from x until we observe a non-zero value. Once a non-zero value has been observed, the consumer obtains the ownership of the location a, together with the knowledge that a is set to 42.

After exiting the loop, the consumer thread looks at the value stored in a, and according to (R-NA) the observed value has to be 42.

Finally, the two threads join. We are no longer interested in any details, so we can simply let the postcondition be true.

Note that our proof establishes much more than simply saying that whenever the loop in the consumer thread terminates, the value observed at location a will be 42. By completing a proof in FSL we also establish that the program is well-formed, i.e., none of its executions (including non-terminating ones!) contain data-races, memory faults, or accesses to uninitialized locations.

The correctness proof of the message passing program where the synchronization is achieved by placing fences (shown in Fig. 3.2) is largely the same as the one we just went through in detail. The novelty is in how the fences work in tandem with the relaxed accesses, and the proof structure follows the idea that release and acquire accesses can be decomposed into relaxed accesses coupled with a fence of the appropriate kind.

Once the producer thread reaches the release fence we utilize the (F-REL) rule to designate the resources which are no longer needed by the producer thread and can be transferred to other threads. Of course, we put $a \mapsto 42$ under the $\triangle$ modality, which in turn enables us to apply (W-RLX), completing the process of relinquishing the ownership of location a.

Conversely, in the consumer thread, the waiting loop is executing relaxed writes, applying (R-RLX) in each loop iteration. Upon reading a non-zero value, the thread obtains the ownership of the location a, but it remains guarded by the $\bigtriangledown$ modality. Executing the acquire fence enables us to get rid of $\bigtriangledown$, thanks to (F-ACQ), thus completing the ownership transfer from the producer to the consumer thread.

Note how in both proofs the assertion $a \mapsto 42$ effectively disappears from the producer thread at the point of origin of the synchronization edge and reappears in the consumer thread at the end-point of the synchronization edge (see the executions shown in Figs. 2.4 and 2.7. When the synchronization is achieved in more intricate ways, we have the two modalities to help us keep track of which resources need to go where, but the modalities render those resources completely unusable by the threads. Therefore, the intuition

$$\{\mathsf{emp}\}$$
$$a = \mathbf{alloc}();$$
$$\{\mathsf{Uninit}(a)\}$$
$$x = \mathbf{alloc}();$$
$$\{\mathsf{Uninit}(a) * \mathsf{Rel}(x, \mathcal{Q}) * \mathsf{Acq}(x, \mathcal{Q})\}$$
$$[x]_{\mathbf{rlx}} = 0;$$
$$\{\mathsf{Uninit}(a) * \mathsf{Rel}(x, \mathcal{Q}) * \mathsf{Acq}(x, \mathcal{Q}) * \mathsf{Init}(x)\}$$

$\{\mathsf{Uninit}(a) * \mathsf{Rel}(x, \mathcal{Q})\}$	$\{\mathsf{Acq}(x, \mathcal{Q}) * \mathsf{Init}(x)\}$
$[a]_{\mathrm{na}} = 42;$	$\mathbf{repeat}\ [x]_{\mathbf{rlx}}\ \mathbf{end};$
$\{a \mapsto 42 * \mathsf{Rel}(x, \mathcal{Q})\}$	$\{v.\, v \neq 0 \land \mathsf{Acq}(x, \mathcal{Q}[v := \mathsf{emp}]) * \mathsf{Init}(x) * \triangledown(a \mapsto 42)\}$
$\mathrm{FENCE}_{\mathbf{rel}};$	$\mathrm{FENCE}_{\mathbf{acq}};$
$\{\vartriangle(a \mapsto 42) * \mathsf{Rel}(x, \mathcal{Q})\}$	$\{\mathsf{Acq}(x, \mathcal{Q}[v := \mathsf{emp}]) * \mathsf{Init}(x) * a \mapsto 42\}$
$[x]_{\mathbf{rlx}} = 1;$	$[a]_{\mathrm{na}}$
$\{\mathsf{Rel}(x, \mathcal{Q}) * \mathsf{Init}(x)\}$	$\{r.\, r = 42 \land v \neq 0 \land \mathsf{Acq}(x, \mathcal{Q}[v := \mathsf{emp}]) * \mathsf{Init}(x) * a \mapsto 42\}$

$$\{\mathit{true}\}$$
where $\mathcal{Q} = \lambda v.\, \mathbf{if}\ v = 0\ \mathbf{then}\ \mathsf{emp}\ \mathbf{else}\ a \mapsto 42.$

Figure 3.2.: Correctness proof for message passing via release fence/acquire fence synchronization program from Fig. 2.7.

of resources being exchanged along the synchronization edges remains a good way of visualizing the proofs.

Why Have Two Distinct Modalities?

An obvious question, which has not been addressed yet, is what is the point of having two different modalities? What would go wrong if we combined $\vartriangle$ and $\triangledown$ into a single modality (let us call it $\diamond$), and simply adapt the inference rules by replacing all the occurrences of $\vartriangle$ and $\triangledown$ by $\diamond$?

At the first glance, everything seems to work well. We would still be unable to access resources hidden behind the modality, and the proofs of programs like the message passing examples we saw before go through as expected. However, as can be seen in Fig. 3.3, things fall apart as soon as we look at slightly more complex programs.

We are looking at a message passing variant where the producer thread on the left is sending a message to the consumer thread on the right, but the communication is not direct. Instead, the middle thread is used to pass the message along from the producer to the consumer.

Note that the program is racy! The producer and the consumer threads do not communicate directly, so a synchronization edge cannot appear between them. The middle thread cannot help, because it contains only relaxed loads and stores, which cannot create synchronization on their own. Therefore, there is absolutely no synchronization happening between the three concurrent threads. In particular, this means that the non-atomic store to a in the left thread is racing with the non-atomic load from a in the right thread.

$$
\begin{array}{c}
\{\mathsf{emp}\} \\
a = \mathbf{alloc}(); \\
\{\mathsf{Uninit}(a)\} \\
x = \mathbf{alloc}(); \\
\{\mathsf{Uninit}(a) * \mathsf{Rel}(x, \mathcal{Q}) * \mathsf{Acq}(x, \mathcal{Q})\} \\
[x]_{\mathbf{rlx}} = 0; \\
\{\mathsf{Uninit}(a) * \mathsf{Rel}(x, \mathcal{Q}) * \mathsf{Acq}(x, \mathcal{Q}) * \mathsf{Init}(x)\} \\
y = \mathbf{alloc}(); \\
\{\mathsf{Uninit}(a) * \mathsf{Rel}(x, \mathcal{Q}) * \mathsf{Acq}(x, \mathcal{Q}) * \mathsf{Init}(x) * \mathsf{Rel}(x, \mathcal{Q}) * \mathsf{Acq}(x, \mathcal{Q})\} \\
[y]_{\mathbf{rlx}} = 0; \\
\{\mathsf{Uninit}(a) * \mathsf{Rel}(x, \mathcal{Q}) * \mathsf{Acq}(x, \mathcal{Q}) * \mathsf{Init}(x) * \mathsf{Rel}(x, \mathcal{Q}) * \mathsf{Acq}(x, \mathcal{Q}) * \mathsf{Init}(y)\} \\
\begin{array}{l||l||l}
\{\mathsf{Uninit}(a) * \mathsf{Rel}(x, \mathcal{Q})\} & \{\mathsf{Acq}(x, \mathcal{Q}) * \mathsf{Init}(x) * \mathsf{Rel}((, y), \mathcal{Q})\} & \{\mathsf{Acq}(y, \mathcal{Q}) * \mathsf{Init}(y)\} \\
[a]_{\mathrm{na}} = 42; & \mathbf{repeat}\ [x]_{\mathbf{rlx}}\ \mathbf{end}; & \mathbf{repeat}\ [y]_{\mathbf{acq}}\ \mathbf{end}; \\
\{a \mapsto 42 * \mathsf{Rel}(x, \mathcal{Q})\} & \{\Diamond a \mapsto 42 * \mathsf{true}\} & \{a \mapsto 42 * \mathsf{true}\} \\
[x]_{\mathbf{rel}} = 1; & [y]_{\mathbf{rlx}} = 1; & [a]_{\mathrm{na}}; \\
\{\mathsf{true}\} & \{\mathsf{true}\} & \{\mathsf{true}\}
\end{array} \\
\{\mathsf{true}\} \\
\text{where } \mathcal{Q} = \lambda v.\, \mathbf{if}\ v = 0\ \mathbf{then}\ \mathsf{emp}\ \mathbf{else}\ a \mapsto 42.
\end{array}
$$

Figure 3.3.: Counterexample demonstrating unsoundness of having only one modality.

Since the program is racy, we should not be able to complete a proof of correctness for it, because the proof should imply that the program is not racy. As we can see in Fig. 3.3, having only one modality allows us to complete a correctness proof of a racy program, which means that a logic which does not differentiate between $\triangle$ and $\triangledown$ is unsound.

The problematic situation with having only one modality happens in the middle thread. After completing the waiting loop, the thread acquires $\Diamond a \mapsto 42$, and the modality does prevent it from committing any races by accessing a before it synchronizes with the thread which produced the resource. However, the middle thread has no intention of accessing a, and simply transfers the ownership of it away using the appropriately modified (W-RLX) rule, and that puts the thread on the right in the position where it is forced to commit a data race.

The lesson here is that enforcing proper synchronization patterns requires more than just a modality which would make resources inaccessible to loads and stores. We really need to differentiate the situation in which the resource is hidden because there was a release fence and it is waiting to be sent away, from the situation in which the resource has be obtained by a relaxed load and is waiting for an acquire fence to complete the ownership transfer.

3.1.5. CAS rules

FSL has a special access permission for allowing atomic update (i.e., compare-and-swap) access to a location. For this reason, a special allocation rule has to be provided, allowing

us to allocate an atomic location on which we can do CAS accesses.

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

This rule is perfectly analogous to the Eq. (A-AT) rule. The only difference is that instead of the atomic read permission $\mathsf{Acq}(\ell, \mathcal{Q})$, the atomic update permission $\mathsf{RMWAcq}(\ell, \mathcal{Q})$ is generated.[1] It is important to note that it is still possible to do atomic reads from the locations allocated using the (A-AT-RMW) rule, due to the following equivalence:

$$\frac{\forall v.\ (\mathcal{Q}'(v) = \mathsf{emp} \vee \mathcal{Q}(v) = \mathcal{Q}'(v) = \mathsf{false})}{\mathsf{RMWAcq}(\ell, \mathcal{Q}) \iff \mathsf{RMWAcq}(\ell, \mathcal{Q}) * \mathsf{Acq}(\ell, \mathcal{Q}')} \tag{RMW-ACQ-SPLIT}$$

Note that this means that we will be unable to do resource transfer using both atomic updates and atomic reads, since (RMW-ACQ-SPLIT) gives us the permission to execute atomic rads, but that permission tells us that we are allowed to transfer only the empty resource emp. Furthermore, if the $\mathsf{RMWAcq}(\ell, \mathcal{Q})$ permission had the information that the location ℓ never gets set to some value v (by having $\mathcal{Q}(v) = \mathsf{false}$), then the carved-out $\mathsf{Acq}(\ell, \mathcal{Q}')$ permission gets to keep that information too.

The restriction that if the resource transfer is happening using atomic updates, then it is happening *exclusively* using atomic updates is a crucial ingredient that allows FSL to have as powerful CAS rules as it does. As we have seen in (ACQ-SPLIT), if there are concurrent readers, they have to take disjoint parts of the resource associated with the location they are reading from. However, concurrent CAS accesses do not behave like regular loads, and we know that at most one out of all concurrent CAS-es can be successful. Therefore, it is perfectly ok that the successful CAS takes over all the resources associated with the accessed location. This idea is reflected by RMWAcq permissions being duplicable:

$$\mathsf{RMWAcq}(\ell, \mathcal{Q}) \iff \mathsf{RMWAcq}(\ell, \mathcal{Q}) * \mathsf{RMWAcq}(\ell, \mathcal{Q})\,. \tag{RMW-SPLIT}$$

Let us now turn our attention to the rules governing atomic updates. Since the rules are somewhat intricate, we will first look at a simplified version of the rule for the strongest type of the CAS instruction, the **acq_rel** CAS. We are going to be very imprecise, omitting many technical details, and focusing on the intuition behind the rule. The general structure of the rule is

$$\frac{\begin{array}{ll}\mathcal{Q}(v) & \Rrightarrow A * T \\ P * T & \Rrightarrow \mathcal{Q}(v')\end{array}}{\{\mathsf{RMW}(\ell, \mathcal{Q}) * P\}\ \mathsf{CAS}_{\mathbf{acq_rel}}(\ell, v, v')\ \left\{\begin{array}{ll} a. & (a = v \wedge A) \\ \vee & (a \neq v \wedge \mathsf{RMW}(\ell, \mathcal{Q}) * P)\end{array}\right\}},$$

where $\mathsf{RMW}(\ell, \mathcal{Q}) = \mathsf{Rel}(\ell, \mathcal{Q}) * \mathsf{RMWAcq}(\ell, \mathcal{Q}) * \mathsf{Init}(\ell)$.

The precondition consists of $\mathsf{RMW}(\ell, \mathcal{Q})$, which gives us the permission to execute CAS on the location ℓ, and the resource P which we want to transfer away upon a successful CAS operation.

[1] RMW stands for read-modify-write, which is another common term referring to atomic updates.

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.

A way to visually imagine the split of $\mathcal{Q}(v)$ into the A and T components is to think of the A part as the resource that is being acquired by the currently executing CAS, and T as the part of the resource which remains in the invariant and "flows down" the release sequence chain to be picked up by some later reader.

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.

Now that we have a good enough picture of the rule's structure, we are ready to look at the full rule.

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

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

(CAS-AR)

There are three technicalities that make the rule look bloated compared to the simplified version discussed above. First, the rule acknowledges that the Rel and RMWAcq permissions do not necessarily have to share the same mapping from values to assertions. Second, the normalizability requirement for the resource P appears. Just as in the rules from Section 3.1.3, this technical requirement causes no difficulties in practice. Third, the rule takes into account that the CAS instruction generates a read event of a specified type σ when the atomic update fails. Thus, the rule requires us to establish what happens in that case. That is the purpose of the $\left\{\mathsf{UPD}(\ell, \mathcal{Q}_{\mathtt{rel}}, \mathcal{Q}_{\mathtt{acq}}) * P\right\} [\ell]_{\sigma} \left\{a.\, a \neq v \rightarrow R\right\}$ premise.

The interesting addition to the simplified rule is the appearance of the pure formula φ. Instead of assertions A and T, the rule 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. Simply stated, this feature of the rule allows us to carry over into the postcondition any additional logical facts we are able to obtain while reestablishing the invariant.

The 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 weaker, the $\triangle$ and $\triangledown$ modalities take over in order to ensure that proper fences have been placed.

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.

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

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

(CAS-REL)

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.

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

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

(CAS-ACQ)

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

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

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

(CAS-RLX)

The last CAS rule (CAS-$\bot$) 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

that would be acquired by a successful CAS operation.

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

$$\frac{\begin{array}{c}\mathcal{Q}(v) * P \Rightarrow \mathsf{false} \\ \left\{\mathsf{UPD}(\ell, \mathcal{Q}_{\texttt{rel}}, \mathcal{Q}_{\texttt{acq}}) * P\right\} [\ell]_{\boldsymbol{\sigma}} \left\{a.\, a \neq v \rightarrow R\right\} \\ \boldsymbol{\tau} \in \{\mathbf{rlx}, \mathbf{rel}, \mathbf{acq}, \mathbf{acq_rel}\} \qquad \sigma \in \{\mathbf{acq}, \mathbf{rlx}\}\end{array}}{\left\{\mathsf{UPD}(\ell, \mathcal{Q}_{\texttt{rel}}, \mathcal{Q}_{\texttt{acq}}) * P\right\} \mathsf{CAS}_{\boldsymbol{\tau},\boldsymbol{\sigma}}(\ell, v, v') \left\{a.\, a \neq v \wedge R\right\}} \quad (\text{CAS-}\bot)$$

3.1.6. Examples Using CAS Rules

Spin Lock

We will illustrate the application of CAS rules by looking at an implementation of locks, which can be abstractly specified by

$$\begin{array}{rcl} & \{R\}\, \text{new_lock}()\, \{x.\, \text{Lock}(x, R)\} & \\ \{\text{Lock}(x, R)\} & \text{lock}(x) & \{R * \text{Lock}(x, R)\} \\ \{R * \text{Lock}(x, R)\} & \text{unlock}(x) & \{\text{Lock}(x, R)\} \\ \text{Lock}(x, R) & \iff & \text{Lock}(x, R) * \text{Lock}(x, R)\,. \end{array}$$

If we have ownership of some resource R, we can create a lock which protects that resource. Creating a lock transfers the ownership of the resource to the lock, and we are given a predicate which says that there is a lock which protects resource R. To get the resource out of the lock, we have to call the lock function, and once we no longer need the resource we put it back in the lock by calling unlock. It is important that the predicate $\text{Lock}(x, R)$ can be freely duplicated, so that multiple threads can use the lock concurrently.

In Fig. 3.4 we have a verified implementation of a lock module, realizing the above abstract specification.

First thing to note is that the $\text{Lock}(x, R)$ predicate is realized as a permission to access an already initialized atomic location x. That atomic location is governed by the protocol described by $\mathcal{Q}_R$, which asks for ownership of R when x is set to 1 and empty ownership when x is set to 0. Duplicability of $\text{Lock}(x, R)$ follows directly from (REL-SPLIT), (RMW-SPLIT), and (INIT-SPLIT).

Function new_lock allocates a new location, initializes it to 1, and returns the allocated location. In the verification, the (A-AT-RMW), (W-REL), and (VAL) rules are used.

Function unlock just sets x to 1, marking the lock as unlocked. The verification uses the (W-REL) for the store instruction, which will generate an extra $\mathsf{Init}(x)$. The extra $\mathsf{Init}(x)$ is absorbed back into $\text{Lock}(x, R)$ using (INIT-SPLIT).

The most interesting part of the verification happens in the lock function which repeatedly tries to acquire the lock by performing a CAS on x. Within the loop, we use the (CAS-ACQ) rule. In the case CAS is not successful, we fall back to (R-RLX) with the help of (RMW-SPLIT). Once CAS succeeds (i.e., x gets updated from 1 to 0), we take out the resource R from the lock.

Connecting the simple use of (CAS-ACQ) in the verification of the lock function to the complicated-looking formal statement of the rule, we observe that we instantiated the

$$\mathcal{Q}_R(v) = (v = 0 \land \mathsf{emp}) \lor (v = 1 \land R)$$

$$\mathrm{Lock}(x, R) = \mathsf{Rel}(x, \mathcal{Q}_R) * \mathsf{RMWAcq}(x, \mathcal{Q}_R) * \mathsf{Init}(x)$$

new_lock() :
$\{R\}$
  let $x = \textbf{alloc}()$ **in**
$\{R * \mathsf{Rel}(x, \mathcal{Q}_R) * \mathsf{RMWAcq}(x, \mathcal{Q}_R)\}$
    $[x]_{\mathbf{rel}} = 1;$
$\{\mathsf{Rel}(x, \mathcal{Q}_R) * \mathsf{RMWAcq}(x, \mathcal{Q}_R) * \mathsf{Init}(x)\}$
    x
$\{x.\, \mathrm{Lock}(x, R)\}$

lock(x) :
$\{\mathrm{Lock}(x, R)\}$
  repeat
    $\mathbf{CAS}_{\mathbf{acq},\mathbf{rlx}}(x, 1, 0)$
$\left\{\begin{array}{l} v.\, (v = 0 \land \mathrm{Lock}(x, R)) \\ \lor (v = 1 \land R * \mathrm{Lock}(x, R)) \end{array}\right\}$
  end;
$\{R * \mathrm{Lock}(x, R)\}$

unlock(x) :
$\{R * \mathrm{Lock}(x, R)\}$
  $[x]_{\mathbf{rel}} = 1$
$\{\mathrm{Lock}(x, R) * \mathsf{Init}(x)\}$
$\{\mathrm{Lock}(x, R)\}$

Figure 3.4.: Implementation and verification of a lock.

various parameters as follows: $\mathcal{A} = \lambda z.\, R$, $\mathcal{T} = \lambda z.\, \mathsf{emp}$, and we simply do not utilize the pure formula φ (i.e., we set $\varphi(z)$ to be a tautology carrying no additional information for every z).

Fetch-and-add

Compare-and-swap is a primitive which allows us to build various other kind of atomic update operations. One commonly used atomic operation is fetch-and-add which atomically adds a specified value to the value stored at a certain location. The goal of this example is to show how an operation such as fetch-and-add can be implemented using CAS, and how to give a very flexible specification to such an operation. We will first look at the implementation, before explaining the specification in detail.

The implementation, shown on the left side of Fig. 3.5, implements fetch-and-add of access type τ, which adds v to the value stored at location x. Fetch-and-add works by repeatedly reading the value stored at x, and then attempting to use CAS to update the value of x from the one which was previously read, to the value incremented by v. Somewhat strange-looking machinations of incrementing the value observed by the CAS at the end of the loop, and then decrementing that value once we get out of the loop are just a technicality for dealing with how our language handles loops. (We make sure that we exit the loop even when the CAS successfully updates from 0, and then we actually return the value which was read by the CAS.)

The specification, shown on the right side of Fig. 3.5, looks rather intricate, but it allows us to do some quite interesting things with fetch-and-add.

```
fetch_and_add_τ(x, v) :
let r = repeat
  let t = [x]_rlx in
  let u = CAS_τ,rlx(x, t, t + v) in
  if u ≠ t then 0 else u + 1
end
in r − 1
```

$$\frac{\forall t.\,(P \iff \mathcal{P}_{\text{send}}(t) * \mathcal{P}_{\text{keep}}(t)) \qquad \forall t.\left(\begin{array}{l}\{\mathsf{UPD}(x, \mathcal{Q}_{\mathtt{rel}}, \mathcal{Q}_{\mathtt{acq}}) * \mathcal{P}_{\text{send}}(t)\} \\ \mathbf{CAS}_{\tau,\mathbf{rlx}}(x, t, t+\mathrm{v}) \\ \left\{\begin{array}{l} y.\ (y = t \wedge \mathcal{R}(t)) \\ \vee\ (y \neq t \wedge \mathsf{UPD}(\ell, \mathcal{Q}_{\mathtt{rel}}, \mathcal{Q}_{\mathtt{acq}}) * \mathcal{P}_{\text{send}}(t))\end{array}\right\}\end{array}\right)}{\{\mathsf{UPD}(\ell, \mathcal{Q}_{\mathtt{rel}}, \mathcal{Q}_{\mathtt{acq}}) * P\}\ \ \mathtt{fetch_and_add}_{\tau}(x, v)\ \ \{y.\,\mathcal{R}(y) * \mathcal{P}_{\text{keep}}(y)\}}$$

$$\tau \in \{\mathbf{rlx}, \mathbf{rel}, \mathbf{acq}, \mathbf{acq_rel}\}$$

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

Figure 3.5.: Fetch-and-add implemented using compare-and-swap.

Before executing the fetch-and-add, the precondition gives us a permission to atomically access x, 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 ℓ from t to $t + v$. We need to prove that upon successful update we can send away $\mathcal{P}_{\text{send}}(t)$ and acquire $\mathcal{R}(t)$, and if the CAS fails we have to reestablish the same resources we had before the CAS.

After executing the fetch-and-add, in the postcondition we get $\mathcal{R}(y) * \mathcal{P}_{\text{keep}}(y)$, with y being the value stored at the location x prior to the update taking place. $\mathcal{R}(y)$ is what we acquired by updating x, while $\mathcal{P}_{\text{keep}}(y)$ is the part we kept from the original resource P we had in the precondition. Note that (as a consequence of the second premise) the acquired resource $\mathcal{R}(y)$ is guaranteed to have a proper modality with respect to the access mode τ.

Proving this specification correct is rather simple. All the heavy-lifting is done by the second premise which makes sure that the loop invariant is reestablished upon a failed CAS, and that we obtain all the necessary resources once the CAS succeeds.

Using the fetch-and-add specification boils down to deciding how we want to split the resource we have for each particular value, and then applying the CAS rule to satisfy the second precondition of the specification. Note that the only non-trivial part is establishing what happens upon a successful CAS. Establishing that we have the same resources upon a failed CAS follows immediately from (R-RLX) and (RMW-ACQ-SPLIT).

Examples using the fetch-and-add specification can be seen in Section 4.2.2.

3.1.7. Ghost State

The last feature we are going to look at in the syntactical overview of FSL is the *ghost state* (Jensen and Birkedal, 2012; Dinsdale-Young et al., 2013; Ley-Wild and Nanevski, 2013; Turon et al., 2014), a very useful feature of program logics, 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 the 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 γ 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.

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

In FSL, the most important feature of ghost state is the 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.

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

This feature is invaluable when reasoning at the points in the program where synchronization might not yet have happened.

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 the 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.

3.1.8. Examples Using the Ghost State

Concurrent Updates

As we saw in Section 2.4.3, only one out of two concurrent CAS instructions can be successful. Interestingly, even to establish such a basic property we need to use the ghost state.

Figure 3.6 contains a verification of a small program which contains two concurrent CAS instructions. The program has two non-atomic locations, a and b, and an atomic location x, initially set to 0. Two threads are concurrently trying to update the value of x from 0 to 1 using CAS. After the CAS-es have finished, the left thread stores the value observed by the CAS in a, and the right thread stores its value in b.

Clearly, it is not possible for both a and b to be set to 0 at the end of the program, as that would mean both CAS-es were successful, which we know cannot happen. How do we prove that using FSL?

$$\mathcal{Q}_\gamma(v) := (v = 0 \land \boxed{\gamma : \bullet}) \lor (v = 1 \land \mathsf{emp})$$

$$\mathsf{UPD}(\ell, \mathcal{Q}_\gamma, \mathcal{Q}_\gamma) := \mathsf{Rel}(\ell, \mathcal{Q}_\gamma) * \mathsf{RMWAcq}(\ell, \mathcal{Q}_\gamma) * \mathsf{Init}(\ell)$$

$$\{\mathsf{emp}\}$$
$$\textbf{let } a = \textbf{alloc}() \textbf{ in}$$
$$\{\mathsf{Uninit}(a)\}$$
$$\textbf{let } b = \textbf{alloc}() \textbf{ in}$$
$$\{\mathsf{Uninit}(a) * \mathsf{Uninit}(b)\}$$
$$x = \textbf{alloc}()$$
$$\left\{\mathsf{Uninit}(a) * \mathsf{Uninit}(b) * \mathsf{Rel}(x, \mathcal{Q}_\gamma) * \mathsf{RMWAcq}(x, \mathcal{Q}_\gamma) * \boxed{\gamma : \bullet}\right\}$$
$$[x]_{\mathbf{rlx}} = 0;$$
$$\{\mathsf{Uninit}(a) * \mathsf{Uninit}(b) * \mathsf{UPD}(x, \mathcal{Q}_\gamma, \mathcal{Q}_\gamma)\}$$

$$\begin{array}{c|c}
\{\mathsf{Uninit}(a) * \mathsf{UPD}(x, \mathcal{Q}_\gamma, \mathcal{Q}_\gamma)\} & \{\mathsf{Uninit}(b) * \mathsf{UPD}(x, \mathcal{Q}_\gamma, \mathcal{Q}_\gamma)\} \\
\textbf{let } u = \textbf{CAS}_{\mathbf{rlx},\mathbf{rlx}}(x, 0, 1) \textbf{ in} & \textbf{let } v = \textbf{CAS}_{\mathbf{rlx},\mathbf{rlx}}(x, 0, 1) \textbf{ in} \\
\left\{\begin{array}{l} u.\ \mathsf{Uninit}(a) * \mathsf{UPD}(x, \mathcal{Q}_\gamma, \mathcal{Q}_\gamma) \\ * ((u = 0 \land \boxed{\gamma : \bullet}) \lor (u = 1 \land \mathsf{emp})) \end{array}\right\} & \left\{\begin{array}{l} v.\ \mathsf{Uninit}(b) * \mathsf{UPD}(x, \mathcal{Q}_\gamma, \mathcal{Q}_\gamma) \\ * ((v = 0 \land \boxed{\gamma : \bullet}) \lor (v = 1 \land \mathsf{emp})) \end{array}\right\} \\
[a]_{\mathrm{na}} = u & [a]_{\mathrm{na}} = v \\
\left\{\begin{array}{l} ((a \mapsto 0 * \boxed{\gamma : \bullet}) \lor (a \mapsto 1 * \mathsf{emp})) * \\ \mathsf{UPD}(x, \mathcal{Q}_\gamma, \mathcal{Q}_\gamma) \end{array}\right\} & \left\{\begin{array}{l} ((b \mapsto 0 * \boxed{\gamma : \bullet}) \lor (b \mapsto 1 * \mathsf{emp})) * \\ \mathsf{UPD}(x, \mathcal{Q}_\gamma, \mathcal{Q}_\gamma) \end{array}\right\}
\end{array}$$

$$\{\exists v, v' \in \{0, 1\}.\, a \mapsto v * b \mapsto v' \land \neg(v = 0 \land v' = 0)\}$$

Figure 3.6.: Verification showing that two concurrent CAS instructions cannot both succeed. The ghost resource uses a PCM in which $\bullet \oplus \bullet$ is undefined.

We are facing a problem because both CAS-es are relaxed, so there will be no synchronization between the two threads, which means that we will be unable to transfer any non-ghost resources between the two threads. However, due to (GHOST-MOD), we can freely transfer ghosts.

When allocating x we choose the protocol governing it such that in order to set x to 0 we have to have some ghost token $\bullet$, which we then transfer away, and when updating x from 0 to 1 we get that token back. We choose the ghost token such that it is not composable by itself (i.e., $\bullet$ belongs to a partial commutative monoid $(M, \oplus)$ such that $\bullet \oplus \bullet$ is undefined), so that more than one token cannot exist at the same time. We can get the initial ghost token by using (GHOST-INTRO) before we allocate x. (NB: we really have to introduce the ghost resource *before* we allocate x because we need to know what γ is in order to use $\mathcal{Q}_\gamma$ as the parameter of the permission assertions governing the use of x.)

With the choice of the protocol for x as described above, we proceed into the two threads. After executing their respective CAS-es, each thread either has a successful CAS, reading 0 and obtaining the ghost token, or a failed CAS, reading 1 and obtaining no additional resources.

$$\mathcal{Q}(v) = (v = 0 \land \mathsf{emp}) \lor (v \neq 0 \land \boxed{\gamma : \bullet})$$

$$\left\{\mathsf{Rel}(x, \mathcal{Q}) * \mathsf{Acq}(x, \mathcal{Q}) * \mathsf{Rel}(y, \mathcal{Q}) * \mathsf{Acq}(y, \mathcal{Q}) * \boxed{\gamma : \bullet}\right\}$$
$$[x]_{\mathbf{rlx}} = 0;$$
$$\left\{\mathsf{Rel}(x, \mathcal{Q}) * \mathsf{Acq}(x, \mathcal{Q}) * \mathsf{Init}(x) * \mathsf{Rel}(y, \mathcal{Q}) * \mathsf{Acq}(y, \mathcal{Q}) * \boxed{\gamma : \bullet}\right\}$$
$$[y]_{\mathbf{rlx}} = 0;$$
$$\left\{\mathsf{Rel}(x, \mathcal{Q}) * \mathsf{Acq}(x, \mathcal{Q}) * \mathsf{Init}(x) * \mathsf{Rel}(y, \mathcal{Q}) * \mathsf{Acq}(y, \mathcal{Q}) * \mathsf{Init}(y) * \boxed{\gamma : \bullet}\right\}$$

$$\begin{array}{c||c}
\left\{\mathsf{Acq}(x, \mathcal{Q}) * \mathsf{Init}(x) * \mathsf{Rel}(y, \mathcal{Q}) * \boxed{\gamma : \bullet}\right\} & \{\mathsf{Acq}(y, \mathcal{Q}) * \mathsf{Init}(y) * \mathsf{Rel}(x, \mathcal{Q})\} \\
[x]_{\mathbf{rlx}}; & \mathbf{repeat}\ [y]_{\mathbf{rlx}}\ \mathbf{end}; \\
\left\{\begin{array}{l} v.\ \mathsf{Rel}(y, \mathcal{Q}) * \boxed{\gamma : \bullet} * \\ ((v = 0 \land \mathsf{emp}) \lor (v \neq 0 \land \boxed{\gamma : \bullet})) \end{array}\right\} & \left\{\mathsf{Rel}(x, \mathcal{Q}) * \boxed{\gamma : \bullet}\right\} \\
\left\{v.\ \mathsf{Rel}(y, \mathcal{Q}) * \boxed{\gamma : \bullet} \land v = 0\right\} & [x]_{\mathbf{rlx}} = 1 \\
[y]_{\mathbf{rlx}} = 1 & \{\mathsf{Rel}(x, \mathcal{Q})\} \\
\{\mathsf{Rel}(y, \mathcal{Q})\} &
\end{array}$$
$$\{\mathsf{true}\}$$

Figure 3.7.: Using ghosts, FSL can prove absence of load buffering. The ghost resource uses a PCM in which $\bullet \oplus \bullet$ is undefined.

At the end of the program we get to conclude that a and b cannot both be set to 0, as that would mean we have two tokens present, i.e., we would have $\boxed{\gamma : \bullet} * \boxed{\gamma : \bullet}$, which is, according to (GHOST-SPLIT), a contradiction.

Load Buffering

As the final example, we will revisit load buffering once again. In 2.5 we discussed the reasons behind the decision to forbid the load buffering behavior in our version of the C11 model, and now we will see that FSL is strong enough to expose that feature of the model.

Figure 3.7 contains a verification of a load buffering variant. The program in question has two locations x and y, both accessed atomically, and at the beginning of the program we set both of them to 0. The left thread reads x and then sets y to 1. The right thread reads y repeatedly until it sees the value 1, after which it sets x to 1. Note that executions in which the left thread sees the value 1 when reading x are of the same shape as depicted in Fig. 2.10. Therefore, the left thread will never read 1, because our model forbids load-buffering-type behavior.

How do we reason about it in FSL, without going down into the nitty-gritty details of the underlying model?

We start by assuming that we have some non-duplicable ghost token $\bullet$, and that both x

and y are governed by the same protocol: when setting either of the locations to 1, we have to provide the ghost token (which means that when we read 1 we get the token out), and we can always freely set them to 0. With this, initializing x and y to 0 poses no trouble, and as expected, the interesting things start happening once we enter the two threads.

We split the resources so that the left thread gets the permission to read from x and write to y, while the right thread gets to write to x and read from y. The ghost token is non-duplicable, so it cannot be given to both threads, and we give it to the left thread.

The left thread starts by reading from x. If it reads 0, it acquires no new resources, and if it reads 1 it gets the ghost token. However, the thread already has the token, and obtaining another copy of a non-duplicable token would lead into a contradiction. Therefore, we can conclude that the value read has to be 0! The left thread then finishes its execution by setting y to 1. It can do so by giving up the ownership of the token.

The thread on the right starts by repeatedly reading from y. It receives no new ownership as long as it reads 0, and once it sees the value 1 it acquires the ghost token. Now, owning the token it can set x to 1 by giving up the ownership of the token.

The proof establishes that the value read by the thread on the left has to be 0, meaning that the load-buffering behavior does not happen in this program. Note that all the accesses in the program are relaxed, so we could not have done this kind of reasoning without the help of the ghost state, because we had to rely on the (GHOST-MOD) property to transfer the resources between the threads.

This example demonstrates that the (GHOST-MOD) property depends on the irreflexivity of $(\mathtt{po} \cup \mathtt{rf})^+$ in the underlying model. In other words, FSL (at least with the ghost state as presented) cannot be sound under models which allow certain forms of load buffering.

3.2. Semantics

3.2.1. Heaps as Models of Assertions

In general, assertions in separation logics are modeled by a partial commutative monoid (Calcagno et al., 2007). Classically, semantics is given by partial functions, called *heaps*, which map allocated locations to values they hold. Together with disjoint union as the monoid operations, heaps form a partial commutative monoid.

To model FSL assertions, we will need much more information than can be given by heaps mapping locations to values. For each location we have to know if it is treated as an atomic or a non-atomic location. Non-atomic locations have partial permissions attached to them, and atomic locations have permissions for executing atomic loads, stores, and CAS-es. There are two modalities which also need to be modeled somehow. Finally, alongside the regular locations, there is the ghost state. In order to model everything expressible by FSL assertions, we are going to build up a more generalized notion of heaps which will map locations to resources rich enough to record all the information we need.

Non-atomic Resources

To model non-atomic locations we need the resources which can differentiate between uninitialized and initialized locations, and if a location is initialized it has to record the value to which the location is set, as well as which part of the ownership permission is associated with that location.

Definition 3.3 (Non-atomic resource) A *non-atomic* resource is a distinct symbol $\mathbb{U}$, or a triple $\mathsf{NA}_P[v, p]$ consisting of a value $v \in \mathsf{Val}$, permission structure P, and a non-empty permission $p \in P$. We denote the class of all non-atomic resources with NA. Composition of non-atomic resources is given by

$$\mathsf{NA}_P[v, p] \oplus \mathsf{NA}_{P'}[v', p'] := \begin{cases} \mathsf{NA}_P[v, p \oplus p'] & \text{if } v = v',\ P = P', \text{ and } p \oplus p' \text{ is defined,} \\ \textbf{undefined} & \text{otherwise.} \end{cases}$$

Composition of $\mathbb{U}$ with any resource is undefined.

The intent is to use $\mathbb{U}$ to mark locations as uninitialized, and $\mathsf{NA}_P[v, p]$ to say that we hold ownership part p of a location which has a value v. The composition operation in permission structures is lifted[1]to the notion of composition of non-atomic resources, which will give rise to the (NA-SPLIT) equivalence.

For the sake of notational convenience, in the rest of the chapter we write $\mathsf{NA}[v, p]$ instead of $\mathsf{NA}_P[v, p]$, assuming that the permission structure P can be inferred from the permission p.

Atomic Resource

Modeling atomic locations gets more intricate because we have to figure out how to represent $\mathsf{Rel}(\ell, \mathcal{Q})$, $\mathsf{Acq}(\ell, \mathcal{Q})$, and $\mathsf{RMWAcq}(\ell, \mathcal{Q})$ assertions, and we have to do it in a way that will support equivalence such as (ACQ-SPLIT), and (RMW-ACQ-SPLIT). Those assertions contain a mapping from values to assertions, so it seems like we will have to find a way to refer to the semantics of assertions while defining it. As shown by Birkedal et al. (2010) this kind of a problem can be resolved by employing advanced category-theoretic construction, but for the sake of simplicity, we are going to resort to a simpler solution.

In order to avoid any potential circularity in our definitions, our resources are not going to contain any kind of semantic structures related to $\mathcal{Q}$ mappings. Instead, we are going to use syntactical assertions to represent $\mathcal{Q}$. More precisely, in order the get Rel, Acq, and RMWAcq to have the properties we want them to support we are going to store syntactical assertions up to the notion of syntactical equivalence defined below.

Definition 3.4 (Syntactical equivalence of FSL assertions) The binary relation $\sim\ \subseteq \mathsf{Assn} \times \mathsf{Assn}$ is the smallest relation satisfying the following properties for all FSL assertions $P, P', Q, Q', R \in \mathsf{Assn}$, mappings $\mathcal{Q}, \mathcal{Q}' \colon \mathsf{Val} \to \mathsf{Assn}$, and locations $\ell \in \mathsf{Loc}$:

[1]The same symbol is used to represent composition of permissions and composition of resources, relying on context to disambiguate which operation is being referred to. We will continue to use $\oplus$ for various related notions of composition, without drawing any more attention to it.

(1) $P \sim P$;
(2) if $P \sim Q$, then $Q \sim P$;
(3) if $P \sim Q$ and $Q \sim R$, then $P \sim R$;
(4) if $P \sim P'$ and $Q \sim Q'$, then $P \to Q \sim P' \to Q'$;
(5) if $P \sim P'$, then $\forall x.\, P \sim \forall x.\, P'$;
(6) if $P \sim P'$ and $Q \sim Q'$, then $P * Q \sim P' * Q'$;
(7) if $\mathcal{Q}(v) \sim \mathcal{Q}'(v)$ for all $v \in \mathsf{Val}$, then $\mathsf{Rel}(\ell, \mathcal{Q}) \sim \mathsf{Rel}(\ell, \mathcal{Q}')$;
(8) if $\mathcal{Q}(v) \sim \mathcal{Q}'(v)$ for all $v \in \mathsf{Val}$, then $\mathsf{Acq}(\ell, \mathcal{Q}) \sim \mathsf{Acq}(\ell, \mathcal{Q}')$;
(9) if $\mathcal{Q}(v) \sim \mathcal{Q}'(v)$ for all $v \in \mathsf{Val}$, then $\mathsf{RMWAcq}(\ell, \mathcal{Q}) \sim \mathsf{RMWAcq}(\ell, \mathcal{Q}')$;
(10) if $P \sim P'$, then $\triangle P \sim \triangle P'$;
(11) if $P \sim P'$, then $\triangledown P \sim \triangledown P'$;
(12) $P * Q \sim Q * P$;
(13) $(P * Q) * R \sim P * (Q * R)$;
(14) $\mathsf{false} * \mathsf{false} \sim \mathsf{false}$; and
(15) $P * \mathsf{emp} \sim P$.

Clearly, the relation we just defined is indeed an equivalence relation.

Lemma 3.1 *The relation $\sim$ is an equivalence relation.*

PROOF Follows immediately from the properties (1)-(3) in Definition 3.4. □

Apart from the first three conditions which ensure that $\sim$ is an equivalence relation, the rest of the conditions list the most important syntactical criteria for assertion equivalence, especially regarding reasoning about the separating conjunction. Later, in Proposition 3.1 we will see that the syntactical equivalence really does imply semantic equivalence.

Now that we have the notion of syntactical equivalence, we can proceed with defining the atomic resources.

In contrast to the non-atomic resources, the atomic resources will not record the value stored at the location they represent. This is so due to the concurrent nature of atomic locations, i.e., when reading from an atomic location the reader, in principle, cannot be sure which value will be observed, since it might have several different writers from which it might be reading. Therefore, neither the assertions nor the resources which talk about atomic locations explicitly mention which values will be observed. The reader can infer the set of observable values by consulting the relevant Acq or RMWAcq permission, as we have seen in various examples from the previous section.

Definition 3.5 (Atomic resource) *Atomic resource* is a quadruple $\mathsf{AT}[\mathcal{R}, \mathcal{A}, \rho, \iota]$ consisting of two mappings $\mathcal{R}, \mathcal{A} \colon \mathsf{Val} \to \mathsf{Assn}/_{\sim}$, and two flags $\rho, \iota \in \{0, 1\}$. We denote the

class of all atomic resources with AT. The composition of atomic resources is defined by

$$
\begin{aligned}
&\mathsf{AT}[\mathcal{R}, \mathcal{A}, \rho, \iota] \oplus \mathsf{AT}[\mathcal{R}', \mathcal{A}', \rho', \iota'] \\
&:= \begin{cases}
\mathsf{AT}[\mathcal{R} \vee \mathcal{R}', \mathcal{A} * \mathcal{A}', 0, \max\{\iota, \iota'\}] & \text{if } \rho = \rho' = 0, \\
\mathsf{AT}[\mathcal{R} \vee \mathcal{R}', \mathcal{A}, 1, \max\{\iota, \iota'\}] & \text{if } \rho = 1, \rho' = 0, \text{ and} \\
& \forall v.\, (\mathcal{A}'(v) = [\mathsf{emp}] \vee \mathcal{A}(v) = \mathcal{A}'(v) = [\mathsf{false}]), \\
\mathsf{AT}[\mathcal{R} \vee \mathcal{R}', \mathcal{A}', 1, \max\{\iota, \iota'\}] & \text{if } \rho = 0, \rho' = 1, \text{ and} \\
& \forall v.\, (\mathcal{A}(v) = [\mathsf{emp}] \vee \mathcal{A}(v) = \mathcal{A}'(v) = [\mathsf{false}]), \\
\mathsf{AT}[\mathcal{R} \vee \mathcal{R}', \mathcal{A}, 1, \max\{\iota, \iota'\}] & \text{if } \rho = \rho' = 1, \text{ and } \mathcal{A} = \mathcal{A}', \\
\textbf{undefined} & \text{otherwise},
\end{cases}
\end{aligned}
$$

where $[P]$ denotes the equivalence class of P according to $\sim$, $\mathcal{R} \vee \mathcal{R}' = \lambda v.\, \mathcal{R}(v) \wedge \mathcal{R}'(v)$, $\mathcal{A} * \mathcal{A}' = \lambda v.\, \mathcal{A}(v) * \mathcal{A}'(v)$, $[P] \vee [Q] = [P \vee Q]$, and $[P] * [Q] = [P * Q]$.

An atomic resource has four components, but it can be seen as having three logical parts.

1. The release part, $\mathcal{R}$, which is there to model Rel assertions.
2. The acquire part, consisting of the mapping $\mathcal{A}$ and the flag ρ. $\mathcal{A}$ represents Acq assertions when ρ is set to 0, and RMWAcq assertions when ρ is set to 1.
3. The initialization flag ι which tells us if the resource is representing an initialized or uninitialized location (0 for uninitialized and 1 for initialized).

With this intuition in mind, let us look at the atomic resource composition.

The simplest part is the composition of the initialization flags. Unsurprisingly, if at least one of the operands says that the represented location is initialized, then it is initialized.

Composition of the acquire part is the most complicated, and it comes in three distinct flavors.

The first case is a composition of two Acq resources (i.e., both ρ flags are set to 0). The resulting resource is still an Acq resource with ρ flag 0, and the two $\mathcal{A}$ mappings combined by the separating conjunction. This composition perfectly matches the (ACQ-SPLIT) equivalence.

In the second case we are composing an Acq and an RMWAcq resource (i.e., one of the ρ flags is 1, and the other is 0). Mirroring (RMW-ACQ-SPLIT), the definition of the resource composition asks that for every value v, either both of the $\mathcal{A}$ mappings map to false, or the one belonging to the Acq resource maps to emp. The composition simply "absorbs" the Acq resource, leaving the RMWAcq operand intact.

The last case is a composition of two RMWAcq resources (i.e., both ρ flags are set to 1). In this case the composition goes through if and only if the $\mathcal{A}$ components of the two operands are equal, and the result is again the same resource, following the intuition behind (RMW-SPLIT).

Finally, the release part is composed by taking the disjunction of the $\mathcal{R}$ components of the two operands. Defining the composition of the release part like this takes into account the (REL-SPLIT) equivalence, and gives us a bit more flexibility.

One important consequence of taking the release part composition to be more flexible than strictly necessary to later derive (REL-SPLIT) is existence of the neutral element for the atomic resource composition. The neutral element is $\mathsf{AT}[\lambda v.\, \mathsf{false}, \lambda v.\, \mathsf{emp}, 0, 0]$. Keep note of it, it will come in handy later when defining the assertion semantics.

Remark When defining the atomic resource composition, we use disjunction and separating conjunction on equivalence classes of assertions, and define it by selecting an arbitrary representative from each class, making a disjunction or separating conjunction of the chosen representatives and proclaiming the equivalence class of the newly constructed assertion to be the disjunction or separating conjunction of the two classes. This is well defined because Definition 3.4 (specifically conditions (4) and (6)) ensures that the result does not depend on which representatives were selected. (Recall that the $P \vee Q$ is treated as syntactic sugar for $(P \to \mathsf{false}) \to Q)$).

Resource Heap

With atomic and non-atomic resources under our belt, we can define resource heaps as mappings from locations to the newly defined resource classes. The resource heaps will not be enough to model all the FSL assertions—specifically ghost assertions and modalities—but are enough to provide semantics for the rest of the logic. They will also be used as the key building blocks in the final semantical structure for FSL assertions.

Definition 3.6 (Resource heap) A finite partial function $h\colon \mathsf{Loc} \rightharpoonup_{\mathit{fin}} \mathsf{NA} \cup \mathsf{AT}$ is called a *resource heap*. We denote the class of all resource heaps with $\mathcal{H}$. Resource heap composition is defined by

$$h \oplus h' \coloneqq \lambda \ell. \begin{cases} h(\ell) & \text{if } \ell \in \mathrm{dom}(h) \setminus \mathrm{dom}(h'), \\ h'(\ell) & \text{if } \ell \in \mathrm{dom}(h') \setminus \mathrm{dom}(h), \\ h(\ell) \oplus h'(\ell) & \text{if } \ell \in \mathrm{dom}(h') \cap \mathrm{dom}(h) \text{ and } h(\ell) \oplus h'(\ell) \text{ defined}, \\ \textbf{undefined} & \text{otherwise.} \end{cases}$$

As expected, the resource heaps are composed by pointwise composition of resources. It is important to keep in mind that the composition of an atomic and a non-atomic resource is always undefined.

Resource heaps, together with the composition operation, form a suitable structure for modeling separation logics, i.e., a partial commutative monoid.

Lemma 3.2 $(\mathcal{H}, \oplus)$ *is a partial commutative monoid with neutral element $\emptyset$, i.e., $\oplus$ is associative and commutative, and $h \oplus \emptyset = \emptyset \oplus h = h$ for every $h \in \mathcal{H}$.*

Remark (on relation with RSL semantics) Those interested in connection of FSL to its immediate predecessor RSL should note that the resource heaps are precisely the semantic structure used to model RSL assertions. This should not come as a surprise, as FSL inherits all of RSL's assertions and adds modalities and ghost assertions.

Remark (for set theorists) You might have noticed that NA, AT, and $\mathcal{H}$ are not sets, but rather proper classes, which does make using the notation $h\colon \mathsf{Loc} \rightharpoonup_{\mathit{fin}} \mathsf{NA} \cup \mathsf{AT}$ and calling $(\mathcal{H}, \oplus)$ a partial commutative monoid somewhat dubious. However, none of it detracts from $\mathcal{H}$ being a suitable semantic structure for separation logic assertions. (It might also be worth noting that each resource heap is indeed a set.) It is, of course, possible to present semantics of FSL without appealing to proper classes, but doing so would be unnecessarily cumbersome without providing any additional clarity. For this reason I hope my set-theorist colleagues will excuse slight abuses of notation and terminology committed here and in some future statements.

Ghost Resources and Ghost Heaps

Before defining FSL heaps, we have to define the resources for modeling ghosts. Similarly to the non-atomic resources, ghost resources simply lift monoid structures, which allows us to to consider elements coming from different monoids as being of the same type of resource.

Definition 3.7 (Ghost resource) A *ghost resource* is a pair $\mathsf{GHOST}_M[g]$ consisting of a partial commutative monoid M, and $g \in M$. We denote the class of all ghost resources with GHOST. The composition of ghost resources is defined by

$$\mathsf{GHOST}_M[g] \oplus \mathsf{GHOST}_{M'}[g'] := \begin{cases} \mathsf{GHOST}_M[g \oplus g'] & \text{if } M = M' \text{ and } g \oplus g' \text{ is defined,} \\ \textbf{undefined} & \text{otherwise.} \end{cases}$$

We will commonly identify a resource $\mathsf{GHOST}_M[g]$ with the monoid element g, assuming the monoid can be inferred from the element.

As one might expect, ghost heaps are mappings from locations to ghost resources, composable pointwise.

Definition 3.8 (Ghost heap) A finite partial function $h\colon \mathsf{Loc} \rightharpoonup_{\mathit{fin}} \mathsf{GHOST}$ is called a *ghost heap*. We denote the class of all resource heaps with $\mathcal{G}$. Ghost heap composition is defined by

$$h \oplus h' := \lambda \ell. \begin{cases} h(\ell) & \text{if } \ell \in \mathrm{dom}(h) \setminus \mathrm{dom}(h'), \\ h'(\ell) & \text{if } \ell \in \mathrm{dom}(h') \setminus \mathrm{dom}(h), \\ h(\ell) \oplus h'(\ell) & \text{if } \ell \in \mathrm{dom}(h') \cap \mathrm{dom}(h) \text{ and } h(\ell) \oplus h'(\ell) \text{ defined,} \\ \textbf{undefined} & \text{otherwise.} \end{cases}$$

FSL Heaps

We now have all the necessary ingredients to define the semantic structure used to model FSL assertions.

Definition 3.9 (FSL Heap) The class of FLS heaps is

$$\boldsymbol{\mathcal{H}} := \{(h_\circ, h_\vartriangle, h_\triangledown, h_g) \in \mathcal{H} \times \mathcal{H} \times \mathcal{H} \times \mathcal{G} \mid h_\circ \oplus h_\vartriangle \oplus h_\triangledown \text{ is defined}\}\,.$$

A quadruple $(h_\circ, h_\vartriangle, h_\triangledown, h_g) \in \boldsymbol{\mathcal{H}}$ is called an *FSL heap* or simply *heap*. The four components of an FSL heap are named the *normal heap* ($h_\circ$), the *release heap* ($h_\vartriangle$), the *acquire heap* ($h_\triangledown$) and the *ghost heap* (h_g). FSL heap composition is defined by

$$(h_\circ, h_\vartriangle, h_\triangledown, h_g) \oplus (h_\circ{}', h_\vartriangle{}', h_\triangledown{}', h_g{}') := \begin{cases} (h_\circ \oplus h_\circ{}', h_\vartriangle \oplus h_\vartriangle{}', h_\triangledown \oplus h_\triangledown{}', h_g \oplus h_g{}') & \text{if } h_\circ \oplus h_\circ{}' \oplus h_\vartriangle \oplus h_\vartriangle{}' \oplus h_\triangledown \oplus h_\triangledown{}' \text{ and} \\ & h_g \oplus h_g{}' \text{ are defined,} \\ \textbf{undefined} & \text{otherwise.} \end{cases}$$

Instead of having a single heap function attaching resources to locations, FSL uses three resource heaps and one ghost heap. Having separate heaps for regular resources and ghost resources is not unexpected, but why keep three resource heaps? The three separate resource heaps will enable us to model the $\vartriangle$ and $\triangledown$ modalities. The idea is to have the normal heap represent the “usable” resources (i.e., those not under any of the modalities), while the release and acquire heaps model the resources protected by $\vartriangle$ and $\triangledown$ modalities, respectively.

Assertion semantics

With all the semantic structures in place, we can finally proceed with defining the semantics of FSL assertions.

Definition 3.10 (FSL assertion semantics) When a heap $\mathbf{h} \in \boldsymbol{\mathcal{H}}$ models an FSL assertion P (notation: $\mathbf{h} \models P$) is given by the following structural recursion on P.

$$\begin{array}{rll}
\mathbf{h} \models \mathsf{false} & \text{never} & \\
\mathbf{h} \models \mathsf{emp} & \text{iff} & \mathbf{h} = (\emptyset, \emptyset, \emptyset, \emptyset) \\
\mathbf{h} \models \mathsf{Uninit}(\ell) & \text{iff} & \mathbf{h} = (\{(\ell, \mathbb{U})\}, \emptyset, \emptyset, \emptyset) \\
\mathbf{h} \models \ell \overset{p}{\mapsto} v & \text{iff} & \mathbf{h} = (\{(\ell, \mathsf{NA}[v, p])\}, \emptyset, \emptyset, \emptyset) \\
\mathbf{h} \models \mathsf{Rel}(\ell, \mathcal{Q}) & \text{iff} & \mathbf{h} = (\{(\ell, \mathsf{AT}[\lambda v.\, [\mathcal{Q}(v)], \lambda v.\, \mathsf{emp}, 0, 0])\}, \emptyset, \emptyset, \emptyset) \\
\mathbf{h} \models \mathsf{Acq}(\ell, \mathcal{Q}) & \text{iff} & \mathbf{h} = (\{(\ell, \mathsf{AT}[\lambda v.\, \mathsf{false}, \lambda v.\, [\mathcal{Q}(v)], 0, 0])\}, \emptyset, \emptyset, \emptyset) \\
\mathbf{h} \models \mathsf{RMWAcq}(\ell, \mathcal{Q}) & \text{iff} & \mathbf{h} = (\{(\ell, \mathsf{AT}[\lambda v.\, \mathsf{false}, \lambda v.\, [\mathcal{Q}(v)], 1, 0])\}, \emptyset, \emptyset, \emptyset) \\
\mathbf{h} \models \mathsf{Init}(\ell) & \text{iff} & \mathbf{h} = (\{(\ell, \mathsf{AT}[\lambda v.\, \mathsf{false}, \lambda v.\, \mathsf{emp}, 0, 1])\}, \emptyset, \emptyset, \emptyset) \\
\mathbf{h} \models \boxed{\gamma : g} & \text{iff} & \mathbf{h} = (\emptyset, \emptyset, \emptyset, \{(\gamma, g)\}) \\
\mathbf{h} \models P \to Q & \text{iff} & \mathbf{h} \not\models P \text{ or } \mathbf{h} \models Q \\
\mathbf{h} \models P * Q & \text{iff} & \mathbf{h} = \mathbf{h}_1 \oplus \mathbf{h}_2, \text{ for some } \mathbf{h}_1, \mathbf{h}_2 \in \boldsymbol{\mathcal{H}} \\
& & \text{such that } \mathbf{h}_1 \models P \text{ and } \mathbf{h}_2 \models Q \\
\mathbf{h} \models \forall x.\, P & \text{iff} & \mathbf{h} \models P[v/x], \text{ for every } v \in \mathsf{Val} \\
\mathbf{h} \models \vartriangle P & \text{iff} & \mathbf{h} = (\emptyset, h, \emptyset, h_g), \text{ for some } h \in \mathcal{H} \text{ and } h_g \in \mathcal{G} \\
& & \text{such that } (h, \emptyset, \emptyset, h_g) \models P \\
\mathbf{h} \models \triangledown P & \text{iff} & \mathbf{h} = (\emptyset, \emptyset, h, h_g), \text{ for some } h \in \mathcal{H} \text{ and } h_g \in \mathcal{G} \\
& & \text{such that } (h, \emptyset, \emptyset, h_g) \models P
\end{array}$$

The logical connectives (including the separating conjunction), quantifiers, and constants false and emp are interpreted in the standard way.

The assertions which reference atomic and non-atomic locations are modeled by atomic and non-atomic resources, respectively, in the normal part of the heap. Note how the atomic assertions use only the specific part of the resource (e.g., RMWAcq assertions utilize the second and the third component), while setting the unused components to be equal to the corresponding components of the neutral element for atomic resource composition.

The release heap is used to model the $\triangle$ modality by saying that if we move the release heap to the position of the normal heap, the resulting heap should satisfy the assertion under the modality. Analogously, the acquire heap is used to provide the semantics of the $\triangledown$ modality.

Let us immediately note that the syntactic equivalence used to define the atomic resources does indeed conform with the semantic equivalence.

Proposition 3.1 *Let P and Q be FSL assertions. If $P \sim Q$, then $P \iff Q$.*

Also, as expected, all the equivalences mentioned in Section 3.1 hold.

Proposition 3.2 *Equivalences* (NA-SPLIT), (REL-SPLIT), (ACQ-SPLIT), (RMW-SPLIT), (RMW-ACQ-SPLIT), (INIT-SPLIT), (GHOST-SPLIT), *and* (GHOST-MOD) *universally hold.*

Another important feature of the assertion semantics is distributivity of modalities over the separating conjunction.

Proposition 3.3 *For every two assertions P and Q, $\triangle(P * Q) \iff \triangle P * \triangle Q$ and $\triangledown(P * Q) \iff \triangledown P * \triangledown Q$.*

Without this property it would be impossible to handle programs which acquire resources by executing multiple relaxed loads and then place a single acquire fence afterwards; or conversely programs which place a single release fence and then use several relaxed stores to release resources which were set up before the fence.

It might also be interesting to note that our semantics treats nested modalities in a fairly trivial way. Any assertion which talks about non-ghost resources (i.e., any assertion which cannot simply ignore the modalities) becomes unsatisfiable under nested modalities.

Proposition 3.4 *Let P be an assertion. Then either*

$$\triangle\triangle P \iff \triangle\triangledown P \iff \triangledown\triangle P \iff \triangledown\triangledown P \iff \mathsf{false}$$

or there exist a finite set G of pairs of locations and ghost resources such that

$$P \iff \mathop{\circledast}_{(\gamma,g)\in G} \boxed{\gamma : g},$$

in which case $P \iff \triangledown P \iff \triangle P$.

Recalling the intuition behind the modalities will help us see why is this a sensible semantics for nested modalities. Assertion $\triangle P$ intuitively means that the resource P has been prepared to be released (by a release fence) and can now be safely sent away using a relaxed store. Conversely, the intuitive meaning of $\triangledown P$ is that we've obtained ownership of P via a relaxed load and we have to wait for an acquire fence before synchronization completes and the resource P becomes usable. Since transferring the information about which fences are executed or expected to be executed from one thread to another is completely useless, it makes perfect intuitive sense to declare nested modalities unsatisfiable.

From a more formal standpoint, being able to use the rules such as (R-RLX) and (W-RLX) to send and receive assertions containing modalities would create soundness problems similar to the one described in Section 3.1.4. Therefore, it would be good if nested modalities never appeared while verifying programs, and having nested modalities unsatisfiable makes sure that they cannot be introduced in preconditions of program specifications.

Looking at the inference rules, there are several points at which nested modalities could be introduced. For example, what is there to stop us from introducing nested modalities via the (F-REL) rule? The normalizability condition, which we are finally going to define does the trick.

Definition 3.11 (Normalizability) An assertion P is *normalizable* if and only if for every $\mathbf{h} \models P$, there exists $h \in \mathcal{H}$, $h_g \in \mathcal{G}$, and $\mathbf{h}' \in \boldsymbol{\mathcal{H}}$ such that $\mathbf{h} = (h, \emptyset, \emptyset, h_g) \oplus \mathbf{h}'$ and $(h, \emptyset, \emptyset, h_g) \models P$.

Normalizability is a semantic property stating that if a heap satisfies a normalizable assertion, then there is some subheap with empty release and acquire parts which also satisfies the assertion. It does more than simply forbidding nested modalities (e.g., $\neg\mathsf{Uninit}(\ell)$ is not normalizable), but it is not at all cumbersome in practice as the following syntactic criterion for normalizability demonstrates.

Proposition 3.5 (Syntactic criterion for normalizability) *Assertions* false *and* emp *are normalizable. For every* $\ell, \gamma \in \mathsf{Loc}$, $v \in \mathsf{Val}$, *permission* p, *and ghost resource* g, $\mathsf{Uninit}(p)$, $\ell \overset{p}{\mapsto} v$, $\mathsf{Init}(\ell)$, *and* $\boxed{\gamma : g}$ *are normalizable.* $\mathsf{Rel}(\ell, \mathcal{Q})$, $\mathsf{Acq}(\ell, \mathcal{Q})$, *and* $\mathsf{RMWAcq}(\ell, \mathcal{Q})$ *are normalizable for every* $\ell \in \mathsf{Loc}$ *and every* $\mathcal{Q} \colon \mathsf{Val} \to \mathsf{Assn}$. *If* P *and* Q *are normalizable assertions, then* $P \wedge Q$, $P \vee Q$, *and* $P * Q$ *are normalizable. If* $P[v/x]$ *is normalizable for every* $v \in \mathsf{Val}$, *then* $\forall x.P$ *and* $\exists x.P$ *are normalizable.*

The above result is telling us that as long as we are describing resources using only positive logical connectives and quantifiers, we are guaranteed to remain in the fragment of normalizable assertions. In practice, that is all we will ever need, since assertions containing negation (such as $\neg\mathsf{Uninit}(\ell)$) are not particularly useful.[1]

Apart from normalizability, another semantic condition we see imposed on assertions in some inference rules (specifically, (R-ACQ) and (R-RLX)) is precision. It is a standard notion (O'Hearn, 2007), stating that an assertion uniquely determines a subheap on which it holds.

[1]Note that $\neg\mathsf{Uninit}(\ell)$ does not simply mean "location ℓ is not uninitialized". It is satisfied by literally any heap different from $(\{(\ell, \mathbb{U})\}, \emptyset, \emptyset, \emptyset)$, which makes $\neg\mathsf{Uninit}(\ell)$ useless for specifying resource ownership.

Definition 3.12 (Precision) An assertion P is *precise* if and only if for all heaps $\mathbf{h}_1, \mathbf{h}'_1, \mathbf{h}_2, \mathbf{h}'_2$, if $\mathbf{h}_1 \models P$, $\mathbf{h}_2 \models P$, and $\mathbf{h}_1 \oplus \mathbf{h}'_1 = \mathbf{h}_2 \oplus \mathbf{h}'_2$, then $\mathbf{h}_1 = \mathbf{h}_2$ and $\mathbf{h}'_1 = \mathbf{h}'_2$.

We will see how precision is used in Section 3.3.3; particularly in the proof of Theorem 3.11.

3.2.2. Semantics of Triples – Heap-annotated Executions

With the assertion semantics pinned down, it is time to move on to the semantics of triples.

As we mentioned before, informally the triple $\{P\}\,E\,\{Q\}$ means that if the program E starts running in a state satisfying the precondition P, then it will not fail, and if it terminates, the resulting state will satisfy the postcondition Q. However, the execution model we are working with is not operational, and as such, the notion of "program state" is not well defined. Instead, we have execution graphs, and we have to find the way to connect our intuitive understanding of triples with the semantics based on execution graphs.

The way we are going to resolve this conundrum is by making the notion of resource ownership front and center of the definition of the triple semantics. We will think of programs as starting with some resources, and as the program executes, those resources "flow" down the execution via the po edges, being mutated by the nodes of the execution (which represent actions taken by the program). When a fork happens (i.e., a parallel composition), the available resources will split into two parts, each of the two new threads inheriting a part of the total resource. Additionally, synchronization can be used to transfer the resources between the threads. Since the synchronization always happens as a consequence of a load operation, we will also allow resources to move along the rf edges (and use the modalities to ensure nothing illicit happens).

In this view, there is no global state of a program. Instead, each thread owns and manages a separate part of the total resource, and at each point during the execution the available resources have to be enough for the action described by the particular node to be performed (e.g., a node labeled $\mathtt{W}_{\mathrm{na}}(x, v)$ attempting to write the value v to the location x, should have exclusive ownership of the location x). We can now recast the intuitive understanding of $\{P\}\,E\,\{Q\}$ to mean that if the program E owns resources satisfying P at the beginning of its execution, then it will not fail, and if it terminates, the resources owned by the program at the end of its execution will satisfy Q.

Annotated executions

To formalize the above idea, we start by labeling po and rf edges with heaps, which represent resources. The intuitive understanding should be that every node has on its disposal resources annotated on its incoming edges, and the effects of the node are represented by the resources coming out on the outgoing nodes. For technical purposes we also attach a virtual "sink edge" to every node of the graph. The reason for this addition will soon become clear as we continue formalizing the idea of heap-annotated executions.

Definition 3.13 (Heap annotations) Let $G = (r, \mathcal{A}, \mathsf{lab}, \mathtt{po}, \mathtt{rf}, \mathtt{mo})$ be an RC11 execution. The set $\mathcal{E}(G) = \{[\mathtt{po}, a, b] \mid \mathtt{po}(a, b)\} \cup \{[\mathtt{rf}, a, b] \mid \mathtt{rf}(a, b)\} \cup \{[\mathtt{sink}, a] \mid a \in \mathcal{A}\}$

is called the set of *annotatable edges*, and a mapping $hmap\colon \mathcal{E}(G) \to \boldsymbol{\mathcal{H}}$ is called a *heap annotation* of the execution G.

We also define $\mathrm{fst}([-,a,-]) \coloneqq a$, $\mathrm{fst}([\texttt{sink},a]) \coloneqq a$, and $\mathrm{snd}([-,-,b]) \coloneqq b$.

Remark (The "omitted component" notation) In the Definition 3.13 above, we used "$-$" instead of naming the component which should be there. We do this to simplify the notation without introducing unnecessary variable names which will not be used within the definition. The omitted component notation is used in situations when the interpretation of the fields marked by "$-$" can be straightforwardly deduced from the context. For example, $\mathrm{fst}([-,a,-]) = a$ should be interpreted as $\forall x,y.\,\mathrm{fst}([x,a,y]) = a$.

We will continue to use this notation heavily throughout the rest of the chapter in order to avoid making the exposition more cumbersome than it has to be.

Of course, just assigning whichever heap we want to each edge does not at all capture our idea of incoming edges to a node representing available resources, which then get affected by the node. In order to further formalize our intuition, we have to declare which annotations are actually valid. Our notion of validity will be the requirement that the annotations around each node follow a kind of Kirchhoff's law (i.e., what comes in, comes out), modulo the effect of the node. The precise requirements are given in the following (rather long) definition, which specifies the exact validity conditions which need to be satisfied at a node, depending on its label.

Definition 3.14 (Annotation validity) Let $G = (r, \mathcal{A}, \mathsf{lab}, \mathsf{po}, \texttt{rf}, \texttt{mo})$ be an execution, and $g\colon \mathcal{A} \to \mathcal{G}$ a function mapping the nodes of the execution to ghost heaps. We say that a heap annotation $hmap\colon \mathcal{E}(G) \to \boldsymbol{\mathcal{H}}$ is *locally valid* at a node $x \in \mathcal{A}$, with respect to the *ghost allocation scheme* g if

$$
\begin{aligned}
&\mathsf{lab}(x) = \texttt{skip} \land \\
&\bigoplus_{\mathsf{po}(a,x)} hmap([\mathsf{po},a,x]) \oplus (\emptyset,\emptyset,\emptyset,g(x)) = \bigoplus_{\mathsf{po}(x,b)} hmap([\mathsf{po},x,b]) \oplus hmap([\texttt{sink},x]) \\
&\qquad\qquad\qquad\qquad\qquad\lor \\
&\quad \mathsf{lab}(x) = \texttt{A}(\ell) \land \exists \mathbf{h}_a. \\
&\quad (\mathbf{h}_a = (\{(\ell,\mathbb{U})\},\emptyset,\emptyset,\emptyset) \lor \exists \mathcal{Q},\rho\,.\mathbf{h}_a = (\{(\ell,\mathsf{AT}[\mathcal{Q},\mathcal{Q},\rho,0])\},\emptyset,\emptyset,\emptyset)) \land \\
&\quad hmap([\mathsf{po},-,x]) \oplus \mathbf{h}_a \oplus (\emptyset,\emptyset,\emptyset,g(x)) = hmap([\mathsf{po},x,-]) \land \\
&\quad hmap([\texttt{sink},x]) = \boldsymbol{\emptyset} \\
&\qquad\qquad\qquad\qquad\qquad\lor \\
&\quad \mathsf{lab}(x) = \texttt{W}_{\mathrm{na}}(\ell,v) \land \exists \mathbf{h}_F.\,(hmap([\mathsf{po},-,x]) = (\{(\ell,\mathbb{U}])\},\emptyset,\emptyset,\emptyset) \oplus \mathbf{h}_F \lor \\
&\quad hmap([\mathsf{po},-,x]) = (\{(\ell,\mathsf{NA}[-,\mathbb{1}])\},\emptyset,\emptyset,\emptyset) \oplus \mathbf{h}_F \land \\
&\quad hmap([\mathsf{po},x,-]) = (\{(\ell,\mathsf{NA}[v,\mathbb{1}])\},\emptyset,\emptyset,\emptyset) \oplus \mathbf{h}_F \oplus (\emptyset,\emptyset,\emptyset,g(x))) \land \\
&\quad \bigoplus_{\texttt{rf}(x,b)} ([\texttt{rf},x,b]) \oplus hmap([\texttt{sink},x]) = \boldsymbol{\emptyset} \\
&\qquad\qquad\qquad\qquad\qquad\lor \\
&\quad \mathsf{lab}(x) = \texttt{R}_{\mathrm{na}}(\ell,v) \land \exists \mathbf{h}_F.\,hmap([\mathsf{po},-,x]) = (\{(\ell,\mathsf{NA}[v,-])\},\emptyset,\emptyset,\emptyset) \oplus \mathbf{h}_F \land \\
&\quad hmap([\mathsf{po},x,-]) = hmap([\mathsf{po},-,x]) \oplus (\emptyset,\emptyset,\emptyset,g(x)) \land \\
&\quad hmap([\texttt{rf}[-,x]) = hmap([\texttt{sink},x]) = \boldsymbol{\emptyset}
\end{aligned}
$$

$$\vee$$

$$\begin{aligned}
&\exists \tau \in \{\mathbf{rel}, \mathbf{rlx}\}.\, \mathsf{lab}(x) = \mathtt{W}_\tau(\ell, v) \land \\
&\exists \mathbf{h}_F, h_\circ, h_\triangle, h_g, h_{ng}, h'_{ng}, \mathcal{Q}.\, g(x) = h_{ng} \oplus h'_{ng} \land \\
&hmap([\mathsf{po}, -, x]) = (\{(\ell, \mathsf{AT}[\mathcal{Q}, \lambda v.\, \mathsf{emp}, 0, -])\}, \emptyset, \emptyset, \emptyset) \oplus (h_\circ, h_\triangle, \emptyset, h_g) \oplus \mathbf{h}_F \land \\
&(\tau = \mathbf{rlx} \rightarrow h_\circ = \emptyset) \land \\
&hmap([\mathsf{po}, x, -]) = (\{(\ell, \mathsf{AT}[\mathcal{Q}, \lambda v.\, \mathsf{emp}, 0, 1])\}, \emptyset, \emptyset, h_{ng}) \oplus \mathbf{h}_F \land \\
&\bigoplus_{\mathtt{rf}(x,b)} hmap([\mathtt{rf}, x, b]) \oplus hmap([\mathtt{sink}, x]) = (\emptyset, \emptyset, h_\circ \oplus h_\triangle, h_g \oplus h'_{ng}) \land \\
&\bigoplus_{\mathtt{rf}(x,b)} hmap([\mathtt{rf}, x, b]) \oplus hmap([\mathtt{sink}, x]) \models \triangledown \mathcal{Q}(v)
\end{aligned}$$

$$\vee$$

$$\begin{aligned}
&\exists \tau \in \{\mathbf{acq}, \mathbf{rlx}\}.\, \mathsf{lab}(x) = \mathtt{R}_\tau(\ell, v) \land \\
&\exists \mathbf{h}_F, \mathbf{h}_a, h_\triangledown, h_g, \mathcal{Q}. \\
&hmap([\mathsf{po}, -, x]) = (\{(\ell, \mathsf{AT}[\lambda v.\, \mathsf{false}, \mathcal{Q}, 0, 1])\}, \emptyset, \emptyset, \emptyset) \oplus \mathbf{h}_F \land \\
&hmap([\mathtt{rf}, -, x]) \models \triangledown \mathcal{Q}(v) \land hmap([\mathtt{rf}, -, x]) = (\emptyset, \emptyset, h_\triangledown, h_g) \land \\
&hmap([\mathsf{po}, x, -]) = (\{(\ell, \mathsf{AT}[\lambda v.\, \mathsf{false}, \mathcal{Q}[v := \mathsf{emp}], 0, 1])\}, \emptyset, \emptyset, g(x)) \oplus \mathbf{h}_F \oplus \mathbf{h}_a \land \\
&(\mathbf{h}_a = (\emptyset, \emptyset, h_\triangledown, h_g) \lor (\mathbf{h}_a = (h_\triangledown, \emptyset, \emptyset, h_g) \land \tau = \mathbf{acq})) \land \\
&hmap([\mathtt{sink}, x]) = (\{(\ell, \mathsf{AT}[\lambda v.\, \mathsf{false}, \lambda u.\, \mathbf{if}\ u = v\ \mathbf{then}\ \mathcal{Q}(v)\ \mathbf{else}\ \mathsf{emp}, 0, 1])\}, \emptyset, \emptyset, \emptyset) \land \\
&\mathsf{precise}(\mathcal{Q}(v)) \land \mathsf{normalizable}(\mathcal{Q}(v))
\end{aligned}$$

$$\vee$$

$$\begin{aligned}
&\exists \tau \in \{\mathbf{acq_rel}, \mathbf{rel}, \mathbf{acq}, \mathbf{rlx}\}.\, \mathsf{lab}(x) = \mathtt{U}_\tau(\ell, v, v') \land \\
&\exists \mathbf{h}_F, \mathbf{h}_t, \mathbf{h}_a, h_\circ, h_\triangle, h_\triangledown, h_g, h'_g, h_{ng}, h'_{ng}, \mathcal{R}, \mathcal{Q}.\, g(x) = h_{ng} \oplus h'_{ng} \land \\
&hmap([\mathsf{po}, -, x]) = (\{(\ell, \mathsf{AT}[\mathcal{R}, \mathcal{Q}, 1, 1])\}, \emptyset, \emptyset, \emptyset) \oplus (h_\circ, h_\triangle, \emptyset, h_g) \oplus \mathbf{h}_F \land \\
&(\tau \in \{\mathbf{rlx}, \mathbf{acq}\} \rightarrow h_\circ = \emptyset) \land \\
&hmap([\mathtt{rf}, -, x]) \models \triangledown \mathcal{Q}(v) \land hmap([\mathtt{rf}, -, x]) = (\emptyset, \emptyset, h_\triangledown, h'_g) \oplus \mathbf{h}_t \land \\
&\bigoplus_{\mathtt{rf}(x,b)} hmap([\mathtt{rf}, x, b]) \oplus hmap([\mathtt{sink}, x]) = (\emptyset, \emptyset, h_\circ \oplus h_\triangle, h_g \oplus h'_{ng}) \oplus \mathbf{h}_t \land \\
&\bigoplus_{\mathtt{rf}(x,b)} hmap([\mathtt{rf}, x, b]) \oplus hmap([\mathtt{sink}, x]) \models \triangledown \mathcal{R}(v') \land \\
&hmap([\mathsf{po}, x, -]) = (\{(\ell, \mathsf{AT}[\mathcal{R}, \mathcal{Q}, 1, 1])\}, \emptyset, \emptyset, h_{ng}) \oplus \mathbf{h}_F \oplus \mathbf{h}_a \land \\
&(\mathbf{h}_a = (\emptyset, \emptyset, h_\triangledown, h'_g) \lor (\mathbf{h}_a = (h_\triangledown, \emptyset, \emptyset, h'_g) \land \tau \in \{\mathbf{acq}, \mathbf{acq_rel}\})) \land \\
&hmap([\mathsf{po}, x, -]) \oplus \bigoplus_{\mathtt{rf}(x,b)} hmap([\mathtt{rf}, x, b]) \oplus hmap([\mathtt{sink}, x])\ \text{defined}
\end{aligned}$$

$$\vee$$

$$\begin{aligned}
&\mathsf{lab}(x) = \mathtt{F}_{\mathbf{rel}} \land \exists \mathbf{h}_F, h_\circ. \\
&hmap([\mathsf{po}, -, x]) = (h_\circ, \emptyset, \emptyset, \emptyset) \oplus \mathbf{h}_F \land \\
&hmap([\mathsf{po}, x, -]) = (\emptyset, h_\circ, \emptyset, g(x)) \oplus \mathbf{h}_F
\end{aligned}$$

$$\vee$$

$$\begin{aligned}
&\mathsf{lab}(x) = \mathtt{F}_{\mathbf{acq}} \land \exists \mathbf{h}_F, h_\triangledown. \\
&hmap([\mathsf{po}, -, x]) = (\emptyset, \emptyset, h_\triangledown, \emptyset) \oplus \mathbf{h}_F \land \\
&hmap([\mathsf{po}, x, -]) = (h_\triangledown, \emptyset, \emptyset, g(x)) \oplus \mathbf{h}_F\,.
\end{aligned}$$

A set of nodes $V \subseteq \mathcal{A}$ is *validly annotated* by heap annotation *hmap* if there exists a ghost allocation scheme g such that *hmap* is locally valid at every node $x \in V$ with respect to g, and for every two nodes $x, x' \in V$, if $\mathrm{dom}(g(x)) \cap \mathrm{dom}(g(x')) \neq \emptyset$, then $x = x'$.

An execution is *validly annotated* by heap annotation *hmap* if its set of nodes is validly annotated by *hmap*.

Before delving into the main points of the validity definition, let us first demystify the appearance of the ghost allocation scheme, which we never mentioned during the motivating discussion prior to giving the formal definition. The ghost allocation scheme is simply a technical tool which allows nodes to create new ghost resources if those resources are needed. We need the nodes to be able to create ghost resources in order to support the (GHOST-INTRO) inference rule. We also require each node to allocate "fresh" ghost resources, which is formally achieved by the requirement that different nodes have to introduce ghost heaps with disjoint domains.

In validity definition for some types of nodes we can see that the existing ghosts are split into h_g and h'_g, and newly allocated ghosts are split into h_{ng} and h'_{ng}. This is done when we need to keep track which existing ghost are coming from po edges and which are coming from rf edges, and similarly which newly allocated ghosts are placed onto outgoing po edges and which ones end up on rf edges.

Another feature universally present in the definition is the appearance of the heap named $\mathbf{h}_F$, which comes into the observed node on the incoming po edge, and leaves undisturbed on the outgoing po edge. That heap represents the frame, i.e., resources which are present in the environment, but are irrelevant to the actions taken by the node.

With the ghost allocation scheme and frames out of the way, let us look in detail at each case of the validity definition. In the discussion below, we will ignore frames and the ghost resources created by the nodes, and focus on what happens to the resources affected by the actions of the node.

When reading the annotation validity definition, it is important to keep the inference rules from Section 3.1 in mind and to notice how the local validity conditions and the corresponding inference rules reflect each other. To visualize this connection, Fig. 3.8 provides a simplified visual summary of the local validity conditions, annotating edges with corresponding syntactical assertions instead of the semantical heaps.

Skip nodes are the simplest, as they do not modify the incoming resources in any way, nor do they require any specific resources to be present. All that a skip node does, is to collect all the resources from the incoming po edges and distribute them among the outgoing po and sink edges. The only interesting feature of the skip nodes is that they are the only kind of nodes which could possibly have multiple incoming or outgoing po edges (in cases when the node corresponds to thread fork or join). Therefore, we have to explicitly sum over all the incoming/outgoing po edges, instead of relying on uniqueness of the po edges, as we will do in all the other cases.

The allocation nodes take the incoming resource and add a new uninitialized location to it (the location can be either non-atomic or atomic). The requirement that the incoming resource does not contain the location which is being initialized is derivable from the implicit assumption that all the mentioned heap compositions are defined.

Non-atomic stores require the incoming resource to include a full non-atomic ownership of the location being written to (either the uninitialized location, or full permission of the location set to some value). After executing, the value of the location is changed to the

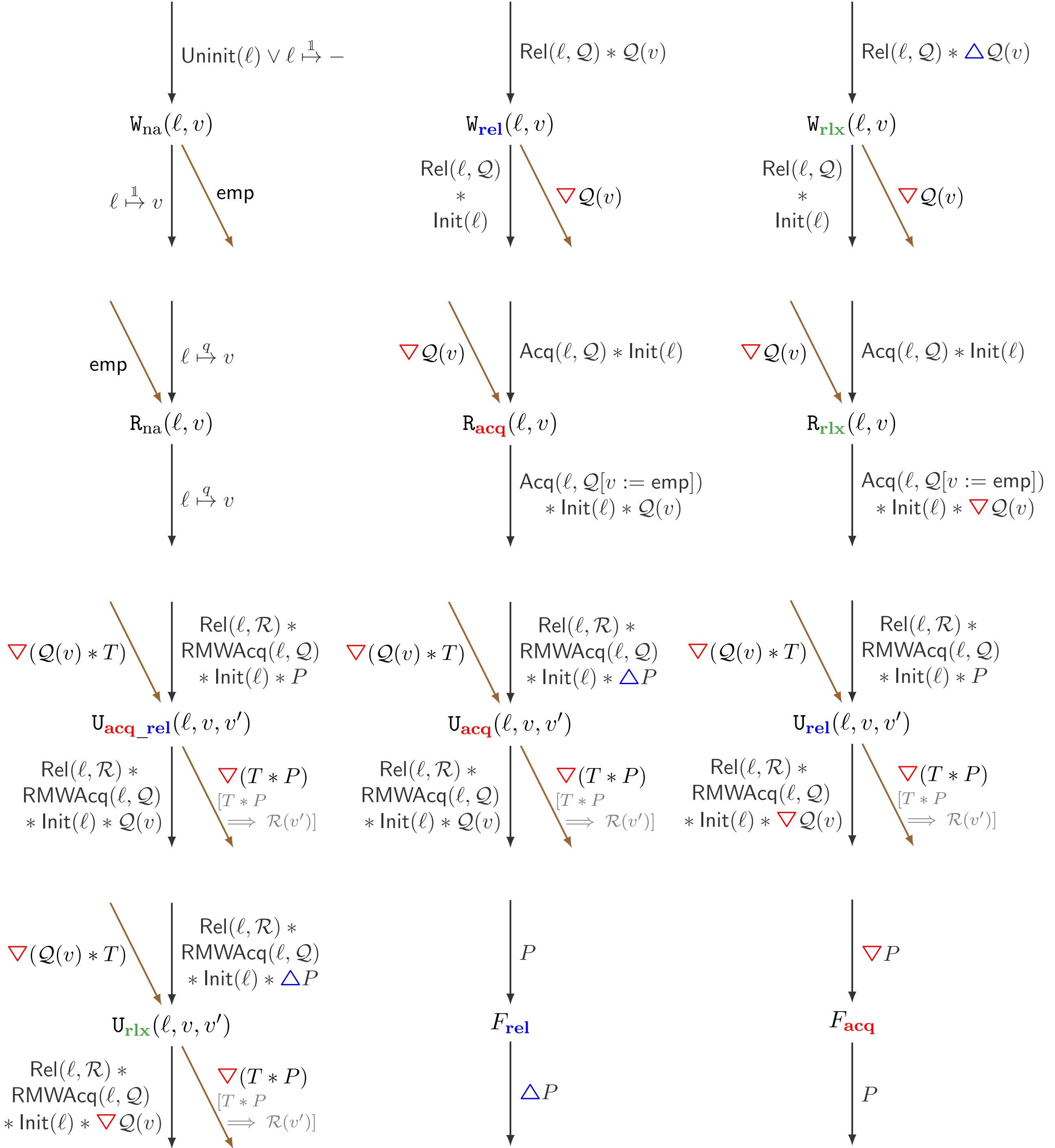


Figure 3.8.: Simplified illustration of local validity conditions for store, load, update and fence nodes, with assertions used for annotations instead of the corresponding heaps. po edges are depicted vertically, and the rf edges are depicted slanted to one side. For simplicity, potential multiple outgoing rf edges are compressed into one, and sink edges, frame heaps, and ghost introductions are left out from the picture.

value being stored, and the rest of the resources remain unaffected. Non-atomic accesses cannot cause synchronization, and as such there cannot be any inter-thread resource transfer via non-atomic stores; this is reflected in the requirement that the outgoing read and sink edges carry only empty resources on them.

Non-atomic loads need to have some fractional part of the total non-atomic location whose value is being read, and the value recorded in the resource should match the one observed by the node. Again, as the non-atomic accesses cannot be used for resource transfer, the incoming `rf` edge has to carry the empty resource.

Atomic stores are the first intricate case we are encountering. Here, the incoming resource has to contain two important parts: a release resource attached to the accessed location, and a resource which is to be transferred after the execution of the store. Note how the resource which is being transferred moves from the incoming po edge, and gets distributed over the outgoing `rf` and `sink` edges. As the resource gets transferred the normal and release components of the incoming resource get converted into the acquire component of the outgoing resource. The resource which is about to be transferred has to have the acquire component empty (as the acquire components are never transferable), and if the store is relaxed, the normal component has to be empty too (since only release stores are able to transfer normal resources).

Here, we can finally illustrate the usefulness of having a sink edge. A store could transfer away some resource, but there is no guarantee that there there will be corresponding readers who will end up picking up the entire resource which the writer made available. In order to avoid messy accounting of seemingly disappearing resources, we will utilize the sink edge to store the resources which have not been claimed by any readers.

Atomic loads have two incoming edges: the po edge carrying the permission to execute the atomic load, and the `rf` edge carrying the resource transferred from the corresponding store. The validity definition requires that the resource on the `rf` edge has to agree with the acquire permission present on the incoming po edge. The atomic load node combines the resources coming in from both incoming edges and places them on the outgoing po edge, with two important alterations: (1) permission to obtain the same resource by reading the same value from the same location is permanently lost (the lost permission gets placed on the sink edge); and (2) if the load is of acquire type, it is allowed to take the acquire part of the resource obtained via the rf edge and move it into the normal position at the outgoing po edge.

The update nodes have the most complicated validity criterion, because they have to combine both the management of the incoming resources from two sources (the `rf` and po edges), as well as handling the dissemination of resources that are being transferred away. The definition follows the structure of annotation validity of atomic stores and loads (making sure that the resources on `rf` edges respect the permissions provided by the incoming po edge), with one crucial addition—the update event does not have to take out the whole resource from the incoming `rf` edge and place it on the outgoing po edge; it can take a part of the resource for the outgoing po edge, and send the rest on the outgoing `rf` (and `sink`) edges.

Finally, we come to the fence nodes whose only job is to manipulate the components of

the heap arriving at the node. A release fence takes the normal heap and moves it to the release position, and an acquire fence moves the acquire heap to the normal position.

The Meaning of Triples

Now that we have introduced the concept of heap-annotated executions, and hammered down what kind of annotations are considered valid, we should utilize it to give the formal meaning to FSL triples. However, before we can pin down the definition for the semantics of triples, there are a couple more issues which we need to iron out:

1. We are interested in giving our triples a meaning which makes sense in the context of concurrency, i.e., a triple $\{P\}\, E\, \{Q\}$ is telling us something about the behavior of the program E, not in isolation, but within and arbitrary well-behaved context. So, we have to make sure that our semantics talks not only about the executions of E, but it also has to refer to the context, and we have to be able to specify what we mean by a well-behaved context.

2. We still have to connect our intuitive idea of programs being executed step-by-step, with the semantics which simply provides us with ready-made execution graphs.

To resolve the first challenge, instead of simply looking at the executions of E, we are going to embed them in a wider context, and annotate those bigger executions.

Definition 3.15 (Contextual executions) A tuple $G = (r, \mathcal{A}_{\text{prg}} \uplus \mathcal{A}_{\text{ctx}}, \mathsf{lab}, \mathtt{po}, \mathtt{rf}, \mathtt{mo})$ is a *contextual execution* of a program E if the following conditions hold:

- $(r, \mathcal{A}_{\text{prg}}, \mathsf{lab}|_{\mathcal{A}_{\text{prg}}}, [\mathcal{A}_{\text{prg}}]; \mathtt{po}; [\mathcal{A}_{\text{prg}}])$ is a pre-execution representing E,
- there exists r' such that $(r', \mathcal{A}_{\text{prg}} \uplus \mathcal{A}_{\text{ctx}}, \mathsf{lab}, \mathtt{po}, \mathtt{rf}, \mathtt{mo})$ is a consistent execution,
- there exists a unique $a \in \mathcal{A}_{\text{ctx}}$ such that $\mathtt{po}(a, \mathtt{po}_{\text{min}}(\mathcal{A}_{\text{prg}}))$, and
- there exists a unique $b \in \mathcal{A}_{\text{ctx}}$ such that $\mathtt{po}(\mathtt{po}_{\text{max}}(\mathcal{A}_{\text{prg}}), b)$.

If $\mathcal{A}_{\text{ctx}} = \{a, b\}$ and $\mathsf{lab}(a) = \mathsf{lab}(b) = \mathtt{skip}$, we say that G is a *minimal contextual execution.*

The first condition in the above definition formalizes the idea that the nodes belonging to $\mathcal{A}_{\text{prg}}$ represent the program E, while the rest of the execution nodes ($\mathcal{A}_{\text{ctx}}$) represent the wider context. We are utilizing the layered approach taken by the C11 semantics (executions build on top of pre-executions)—by exploiting the notion of pre-execution to formalize the idea of embedding E in a wider context, we are able to ensure that the nodes of $\mathcal{A}_{\text{prg}}$ really represent events generated by E, while still allowing the program to communicate with the wider context by allowing $\mathtt{rf}$ edges to cross between $\mathcal{A}_{\text{prg}}$ and $\mathcal{A}_{\text{ctx}}$.

Note that when looking at the contextual executions we are not interested in the return value of the entire execution, but only in its consistency (as stated in the second condition above). The return value we are interested in is the return value associated with the

program E, and we identify that value by looking at the part of the whole execution which is representing the program E. It might seem like there is a problem with this way of identifying the return value of E because we are doing that on the level of pre-executions, and pre-executions have completely unconstrained load events; thus it looks like we might end up with arbitrary return values for E. Luckily, the pre-execution of E is extracted from a larger consistent execution, and that is what ensures that the load events of E are not obtaining arbitrary values.

The remaining two conditions state that the execution of E is connected to the wider context by a unique incoming and a unique outgoing po edge. This clear delineation between the program nodes and the context nodes allows us to find a natural place for the resources representing triples' precondition and postcondition. The precondition goes on the incoming po edge, and the postcondition should appear on the outgoing po edge.

The definition of contextual executions does not contain the solution to the question of what we mean by a "well-behaved" context. We will come back to that point later.

The second challenge, connecting the idea of step-by-step execution to the notion of executions as graphs requires a bit of subtlety. The idea is to imagine nodes of the execution being "activated" one by one as the program executes—at each execution step, one of the nodes in the graph gets added to the set of already "active" ones.

That is a good idea, but is there a sensible order in which we should be activating the nodes of our execution? Luckily for us, there is—recall that $(\mathsf{po} \cup \mathtt{rf})^+$ is irreflexive, i.e., it is a partial order; it also provides a very sensible activation order. If at each activation step we activate a $(\mathsf{po} \cup \mathtt{rf})^+$-minimal node among the non-activated ones, we will never end up in some kind of a strange situation where, for example, we are attempting to activate a load node that is reading from an unactivated store node.

To formalize this step-by-step activation idea, we start by looking at the sets which are $(\mathsf{po} \cup \mathtt{rf})^+$-prefix-closed, i.e., sets of nodes such that taking a step back along a po edge or an $\mathtt{rf}$ edge does not take us outside the set.

Definition 3.16 (Prefix-closed sets) Let $(r, \mathcal{A}, \mathsf{lab}, \mathsf{po}, \mathtt{rf}, \mathtt{mo})$ be an execution and $V \subseteq \mathcal{A}$. The *prefix* of V is the set

$$\mathsf{Pre}(V) \coloneqq \{a \in \mathcal{A} \mid \exists b \in V.\, \mathsf{po}(a, b) \vee \mathtt{rf}(a, b)\}\,.$$

The set V is *prefix-closed* if $\mathsf{Pre}(V) \subseteq V$.

As we proceed along the execution, we will be building an increasing prefix-closed set whose nodes will have to be validly annotated (think of it as the set of executed actions), which means that eventually the entire execution will be annotated (here it once more becomes important that our executions are always finite structures where infinite executions are represented by their "unfinished" prefixes). As we are adding new nodes which need to be validly annotated, we are effectively getting new constraints on the heap annotation of the execution's edges.

Keeping on with the idea of heap annotations representing resources flowing down the execution graph, we do not want just any valid annotation to be used at each step as we continue growing our prefix-closed set. Intuitively, what we need is the annotations

to be kept constant on all the edges constrained by validity conditions of the nodes in our prefix-closed set, and adding a new node to it should only affect the annotations of previously unconstrained edges.

At this point we need to be a little bit careful when talking about validity conditions of the nodes constraining the annotations of the edges. Strictly speaking, the annotation validity definition constrains every edge coming in or out of the node, but that is not what we were intuitively talking about. What we have in mind is that as we proceed down the execution and keep annotating the edges coming in and out of the nodes we added to our prefix-closed set of annotated nodes, we imagine the resources flowing down the execution and once the resources have traveled along a particular edge, then the annotation of that edge needs to be set in stone. What we need is a notion that captures this idea.

Before giving the formal definition, let us summarize the important intuitive notions we want to capture.

- When the action of a certain node is being executed, that node should know which resources are available for it on its incoming po edges, but the action of the node is what decides the resources that go on the outgoing po edges.
- The readers are deciding what goes on their incoming rf edges, because they are the ones that know what resources they need in order to achieve what they want to do. The writers cannot possibly know who is going to end up reading from them, so they should have no business deciding the annotations of their outgoing rf edges.
- The sink edges are not real edges of the execution, but are there to help us with accounting. Therefore, changing the annotations of sink edges when needed is always ok.

This means that when being added to the set of executed actions, the node decides on the annotations of its outgoing po and incoming rf edges, and once those annotations are set, they should never change. We formalize this idea in the following definition.

Definition 3.17 (Annotation responsibility) Let $G = (r, \mathcal{A}, \mathsf{lab}, \mathsf{po}, \mathtt{rf}, \mathtt{mo})$ be an execution. The set of edges a node $a \in \mathcal{A}$ is *responsible for annotating* is

$$\mathsf{Resp}(a) = \{e \in \mathcal{E}(G) \mid e = [\mathsf{po}, a, -] \vee e = [\mathtt{rf}, -, a]\}\,.$$

For a set of nodes $V \subseteq \mathcal{A}$ we denote $\mathsf{Resp}(V) = \bigcup_{a \in V} \mathsf{Resp}(a)$.

We are now finally ready to define the concept in terms of which the semantics of FSL triples will be given.

Definition 3.18 (Configuration safety) Let $G = (r, \mathcal{A}_{\text{prg}} \uplus \mathcal{A}_{\text{ctx}}, \mathsf{lab}, \mathsf{po}, \mathtt{rf}, \mathtt{mo})$ be a contextual execution of some program (with $\mathcal{A}_{\text{prg}}$ representing the actions of the program). For a set of nodes $V \subseteq \mathcal{A}_{\text{prg}} \uplus \mathcal{A}_{\text{ctx}}$, a heap annotation $hmap\colon \mathcal{E}(G) \to \boldsymbol{\mathcal{H}}$, a class of heaps $\boldsymbol{\mathcal{Q}} \subseteq \boldsymbol{\mathcal{H}}$, and a natural number $n \in \mathbb{N}$, we define the safety predicate $\mathsf{safe}^n_G(V, hmap, \mathcal{A}_{\text{prg}}, \mathcal{A}_{\text{ctx}}, \boldsymbol{\mathcal{Q}})$ recursively on n as follows:

- $\mathsf{safe}^0_G(V, hmap, \mathcal{A}_{\text{prg}}, \mathcal{A}_{\text{ctx}}, \boldsymbol{\mathcal{Q}})$ holds if V is validly annotated by $hmap$;
- $\mathsf{safe}^{n+1}_G(V, hmap, \mathcal{A}_{\text{prg}}, \mathcal{A}_{\text{ctx}}, \boldsymbol{\mathcal{Q}})$ holds if all of the following conditions hold:
  - for all $a \in \mathcal{A}_{\text{ctx}} \setminus V$ such that $\mathsf{Pre}(\{a\}) \subseteq V$, and all $hmap' \colon \mathcal{E}(G) \to \boldsymbol{\mathcal{H}}$ such that $hmap|_{\mathsf{Resp}(V)} = hmap'|_{\mathsf{Resp}(V)}$, if $V \cup \{a\}$ is validly annotated by $hmap'$ then $\mathsf{safe}^n_G(V \cup \{a\}, hmap', \mathcal{A}_{\text{prg}}, \mathcal{A}_{\text{ctx}}, \boldsymbol{\mathcal{Q}})$;
  - for all $a \in \mathcal{A}_{\text{prg}} \setminus V$ such that $\mathsf{Pre}(\{a\}) \subseteq V$, there exists $hmap' \colon \mathcal{E}(G) \to \boldsymbol{\mathcal{H}}$ such that $hmap|_{\mathsf{Resp}(V)} = hmap'|_{\mathsf{Resp}(V)}$, the set $V \cup \{a\}$ is validly annotated by $hmap'$ and $\mathsf{safe}^n_G(V \cup \{a\}, hmap', \mathcal{A}_{\text{prg}}, \mathcal{A}_{\text{ctx}}, \boldsymbol{\mathcal{Q}})$;
  - if $\mathtt{po}_{\max}(\mathcal{A}_{\text{prg}}) \in V$, then $hmap([\mathtt{po}, \mathtt{po}_{\max}(\mathcal{A}_{\text{prg}}), -]) \in \boldsymbol{\mathcal{Q}}$

Intuitively, the safety predicate is telling us, given a validly annotated configuration, for how many additional steps can we continue to validly annotate the execution. The set V is representing the nodes whose edges have already been annotated, and we are proceeding to take n more annotation steps from there. Each annotation step consists of picking the next node to be annotated (among those that are immediate successors of the nodes from V according to the $\mathtt{po} \cup \mathtt{rf}$ relation), and annotating the edges connected with that node.

If we are taking no more steps (i.e., $n = 0$), then we are not annotating any more nodes, and our only requirement is that the set V is validly annotated.

If we are taking $n+1$ steps, then the definition separates in two different cases, depending on if the next node we are adding belongs to the context ($\mathcal{A}_{\text{ctx}}$), or is it one of the nodes belonging to the program ($\mathcal{A}_{\text{prg}}$).

For the steps in the context, we are simply assuming that any valid annotation of the newly annotated node keeps the configuration safe for n more steps. This is how we formalize the idea that the context is "well-behaved". Simply stated, the context can be arbitrary, so we have to rely on it to be structured in such a way that it will not create any issues.

For the program steps, establishing safety for $n+1$ steps boils down to showing that we can validly annotate the next node in such a way that we remain safe for n more steps. Here, we do not assume safety, but instead, since the node is belonging to our program, we have to guarantee that it behaves correctly.

Finally, we require that if the annotation has reached the $\mathtt{po}$ edge exiting the program and going into the context, then the heap annotation of that edge has to belong to the class of heaps $\boldsymbol{\mathcal{Q}}$. This requirement is there to enable us to talk about the meaning of the postcondition in the definition of the semantics of the FSL triples.

Definition 3.19 (Semantics of triples) The FSL triple $\{P\}\, E\, \{y.\, Q\}$ holds if and only if for all contextual executions $G = (r, \mathcal{A}_{\text{prg}} \uplus \mathcal{A}_{\text{ctx}}, \mathsf{lab}, \mathtt{po}, \mathtt{rf}, \mathtt{mo})$ of the program E, for all prefix-closed set of nodes $V \subseteq \mathcal{A}_{\text{ctx}}$, for all heap-annotations $hmap \colon \mathcal{E}(G) \to \boldsymbol{\mathcal{H}}$ validly annotating V, for all assertions $R \in \mathsf{Assn}$, if $hmap([\mathtt{po}, -, \mathtt{po}_{\min}(\mathcal{A}_{\text{prg}})]) \models P * R$,

then for all $n \in \mathbb{N}$, $\mathsf{safe}^n_G(V, hmap, \mathcal{A}_{\text{prg}}, \mathcal{A}_{\text{ctx}}, \boldsymbol{\mathcal{Q}})$, where

$$\boldsymbol{\mathcal{Q}} = \begin{cases} \{\mathbf{h} \oplus \mathbf{h}_{FR} \oplus \mathbf{h}' \mid \mathbf{h} \models Q(r),\ \mathbf{h}_{FR} \models R, \text{ and } \mathbf{h} \oplus \mathbf{h}_{FR} \oplus \mathbf{h}' \text{ is defined}\} & \text{if } r \neq \bot\,, \\ \boldsymbol{\mathcal{H}} & \text{if } r = \bot\,. \end{cases}$$

In broad strokes, a triple $\{P\}\,E\,\{y.\,Q\}$ holds if any contextual execution of the program E, in which the program starts with resources satisfying the assertion P, can be safely annotated for arbitrarily many steps, and if the program terminates, the resources at the end of the program's execution must satisfy the assertion $Q(r)$, where r is the return value of the program. Formally, the requirement that the program starts with certain resources is expressed by requiring those resources to be annotated on the po edge entering the program nodes from the context nodes, and the requirement that certain resources are there at the end of the program's execution is expressed by having those resources annotated on the po edge leaving the program nodes and entering the context.

In the definition we do not ask that the resources on the incoming po edge satisfy only the precondition P, but the assertion $P * R$, for some arbitrary R. This way we can bake-in the frame rule directly into the definition of the FSL triple semantics—the resource R represents the frame.

We also do not simply ask that the resources on the outgoing po edge satisfy the postcondition $Q(r)$, but instead the heap annotated on the outgoing po edge is composed of three parts: the heap $\mathbf{h}$, satisfying $Q(v)$; the heap $\mathbf{h}_{FR}$, satisfying R; and an arbitrary heap $\mathbf{h}'$. The point of the heaps $\mathbf{h}$ and $\mathbf{h}_{FR}$ is clear—they correspond to the postcondition and the frame—but what is the point of $\mathbf{h}'$; why are we allowing some resources to be "forgotten" in the postcondition? We allow resources to be forgotten due to a technical issue which appears in the soundness proof of some rules[1]. We will return to the discussion of this choice in the definition in Section 3.3.3, once we had a chance to see the technicality in the soundness proof which is addressed by this weakening of the requirement on the postcondition.

Lastly, note that the requirement that the heap annotated on the outgoing po edge satisfies the postcondition is only there for the terminating executions of the program E (i.e., the executions which actually return a value, and not the $\bot$ symbol). This is in concordance with the notion that the FSL triples only state that the terminating programs satisfy the postcondition, while the only requirement for non-terminating programs is that they do not go wrong, which is here formalized by asking that the contextual executions of the program can be safely annotated for an arbitrary number of steps. The following section will demonstrate that the notion of configuration safety corresponds well with the intuitive notion of program "not going wrong".

[1] More specifically, the rules which mention the normalizability in their precondition, i.e., (F-REL), (W-REL), (R-ACQ), (CAS-AR), and (CAS-REL)

3.3. Soundness

In this section we will establish the soundness of the FSL proof rules which were presented in Section 3.1, but we are interested in more than just the soundness of the proof rules. We want to establish that a valid FSL triple $\{P\}\, E\, \{Q\}$ tells us something substantial about the program E. Specifically, we want to establish that by proving $\{P\}\, E\, \{Q\}$ we also prove that the program E is well-formed (in the sense of Definition 2.15), i.e., the program does not contain data-races, uninitialized reads, and faulty memory accesses.

We will start the section off by investigating some general properties of well-annotated executions, which will enable us to prove the well-formedness of the programs for which an FSL triple can be established. At the end of the section we will go over the proofs of soundness of the FSL proof rules.

The goal is to present the proofs in enough detail so that all the key moments of the proofs are pointed out and understandable, but without getting bogged down into heavy formality. The full formal proofs are available in the Coq development, and that enables us to focus on a higher-level picture.

3.3.1. Properties of Validly Annotated Executions

First, let us turn our attention towards exploring validly annotated executions. The properties we explore here will help us establish that FSL triples can be proven to hold only of well-formed programs.

Independent Heap Compatibility

The first property we focus on can be thought of as a very important sanity check of our semantics definitions so far. This whole time we have been thinking of resources flowing through the graph along the po and rf edges, telling a story of threads owning specific resources, operating on them, and transferring the ownership of those resources among them. If this intuition holds any water, it should better be the case that two threads executing concurrent events own non-conflicting resources. Here, by "non-conflicting resources" we mean that the heaps representing those resources should be composable.

The formal concept corresponding to the concurrent events would be to look at the nodes in the graph which are not related by the happens-before (hb) relation, but we will play a bit looser here and actually look at a larger set of nodes—the nodes not related by the $(\mathtt{po} \cup \mathtt{rf})^+$ relation. We will consider two edges in the graph to be independent if it is not possible to reach from the endpoint of one of the edges to the starting point of the other edge via a path consisting of po and rf edges.

Definition 3.20 (Independent edges) Let G be an execution, and $e_1, e_2 \in \mathcal{E}(G)$ edges of the execution G. We say that e_1 and e_2 are *independent edges* if

$$\neg\left((\mathtt{po} \cup \mathtt{rf})^*(\mathrm{snd}(e_1), \mathrm{fst}(e_2)) \vee (\mathtt{po} \cup \mathtt{rf})^*(\mathrm{snd}(e_2), \mathrm{fst}(e_1))\right)$$

We say that a set $T \subseteq \mathcal{E}(G)$ is a set of *pairwise independent* edges if every two edges $e_1, e_2 \in T$ are independent edges.

If our intuitive notion of resources flowing through the graph is sensible, then a theorem stating that the heaps annotated on two independent edges are compatible should hold.

Theorem 3.1 *Let V be a prefix-closed set of nodes in an execution G. If V is validly annotated by hmap, and $e_1, e_2 \in \mathcal{E}(G)$, $e_1 \neq e_2$, are two independent edges such that* $\mathrm{fst}(e_1), \mathrm{fst}(e_2) \in V$, *then the heap composition $hmap(e_1) \oplus hmap(e_2)$ is defined.*

This exact theorem comes as a corollary of a more general theorem about the heap compatibility of an arbitrary set of pairwise independent edges.

Theorem 3.2 (Inependent heap compatibility) *Let V be a prefix-closed set of nodes in an execution G, and let $T \subseteq \mathcal{E}(G)$ be a set of pairwise independent edges, such that* $\mathrm{fst}(e) \in V$ *for every $e \in T$. If V is validly annotated by hmap, then the heap composition* $\bigoplus_{e \in T} hmap(e)$ *is defined.*

Proof The idea is to prove this theorem by the induction on a value telling us how “deep” in the graph the set T is.

Let A be a set of nodes of the execution G. We define the depth of A to be the number of nodes that precede the nodes in A according to the $(\mathtt{po} \cup \mathtt{rf})^*$ relation. Formally:

$$\begin{aligned} \mathsf{Pre}_0(A) &:= A \\ \mathsf{Pre}_{n+1}(A) &:= \mathsf{Pre}(\mathsf{Pre}_n(A)) \end{aligned}$$

and

$$\mathbf{D}(A) := \left| \bigcup_{n \in \mathbb{N}} \mathsf{Pre}_n(A) \right| .$$

For a set of edges we define its depth to be the depth of the set of the first nodes of the edges from the set:

$$\mathbf{D}(T) := \mathbf{D}(\{\mathrm{fst}(e) \mid e \in T\}) .$$

The proof will proceed by induction on $\mathbf{D}(T)$.

Let T be an arbitrary set of pairwise independent edges, such that for every set of pairwise independent edges T' for which $\mathbf{D}(T') < \mathbf{D}(T)$ the composition $\bigoplus_{e \in T'} hmap(e)$ is defined.

If $T = \emptyset$, we obviously have nothing to prove.

If T is non-empty, look at an edge $e \in T$ such that $\mathbf{D}(\{e\}) \geqslant \mathbf{D}(\{e'\})$ for every $e' \in T$.

If $\mathbf{D}(\{e\}) = 1$, then $T = \{e\}$, because only the initial edge of the execution has no preceding nodes, and we still have nothing to prove.

In the case $\mathbf{D}(\{e\}) > 1$, define

$$\begin{aligned} T_e &:= \{x \in T \mid \mathrm{fst}(x) = \mathrm{fst}(e)\} \\ \mathrm{incoming}(e) &:= \left\{e' \in \mathcal{E}(G) \mid \mathrm{snd}(e') = \mathrm{fst}(e)\right\} \end{aligned}$$

and consider the set

$$T' := (T \setminus T_e) \cup \mathrm{incoming}(e) .$$

Essentially, we obtain the set T' by taking out an edge e of maximal depth (and all the edges that share the same first node), from the set T and replacing it with all the edges "incoming" into e (i.e., edges whose second node is the same as the first node of e).

Note that the set T' is a set of pairwise independent edges:

- Any pair of edges from $T \setminus T_e$ is independent because T is pairwise independent.
- Any pair of edges from $\text{incoming}(e)$ is independent because they share the second node, so their independence follows from irreflexivity of $(\mathsf{po} \cup \mathtt{rf})^+$.
- For $e_1 \in T \setminus T_e$ and $e_2 \in \text{incoming}(e)$, if it were possible to arrive from e_1 to e_2 via a $(\mathsf{po} \cup \mathtt{rf})^*$ path, that would contradict independence of T; and if it were possible to arrive from e_2 to e_1 via a $(\mathsf{po} \cup \mathtt{rf})^*$ path, then e would not be of maximal depth in T.

Also note that $\mathbf{D}(T') < \mathbf{D}(T)$, because $\bigcup_{n\in\mathbb{N}} \mathsf{Pre}_n(T') = \bigcup_{n\in\mathbb{N}} \mathsf{Pre}_n(T) \setminus \{\text{fst}(e)\}$. Therefore, by the inductive hypothesis, we have that the heap composition

$$\bigoplus_{x\in T'} \mathit{hmap}(x) = \bigoplus_{x\in T\setminus T_e} \mathit{hmap}(x) \oplus \bigoplus_{x\in \text{incoming(e)}} \mathit{hmap}(x)$$

is defined.

Now, if we could prove that given a node a of the execution, and a heap $\mathbf{h}$, if the composition of the heap annotations on the edges incoming to a is composable with $\mathbf{h}$, then the composition of the heaps annotated on the outgoing nodes from a is also composable with $\mathbf{h}$, the proof would be completed. (Just set $a = \text{fst}(e)$ and $\mathbf{h} = \bigoplus_{x\in T\setminus T_e} \mathit{hmap}(x)$.)

The above property can be proven by (somewhat tedious, and not particularly interesting) case analysis on the local validity definition. The details of the proof can be seen in the Coq development, so we will avoid going over it here. □

Memory Safety

We are now turning our attention to the three properties which are needed for program well-formedness (Definition 2.15). We will go over them in the order of increasing proof complexity, and the first one on the menu is memory safety.

Theorem 3.3 *Every validly annotated execution is memory safe.*

Proof Let G be an execution validly annotated by hmap. Let a be a node in g which is accessing a location ℓ, i.e. $\mathsf{lab}(a) = \mathtt{R}_{_}(\ell, -)$, $\mathsf{lab}(a) = \mathtt{W}_{_}(\ell, -)$, or $\mathsf{lab}(a) = \mathtt{U}_{_}(\ell, -, -)$. We need to prove that there is a node allocating ℓ which is $\mathtt{hb}$-before a.

From the annotation validity definition, we see that ℓ is in the domain of the normal heap of $\mathit{hmap}([\mathsf{po}, -, a])$.

First thing we will notice is that if ℓ is in the domain of one of the heap parts (normal, release, or acquire) of $\mathit{hmap}(e)$, for some edge $e \in \mathcal{E}(G)$, then only the following two cases are possible:

1. There exists an edge e', such that $\text{snd}(e') = \text{fst}(e)$ (we will say that e' *precedes* e in the rest of the proof) and ℓ is in the domain of one of the heap parts of $hmap(e')$.

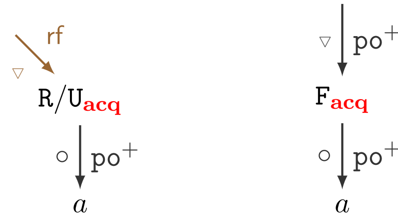
2. The first node of e is the allocation of ℓ, i.e., $\mathsf{lab}(\text{fst}(e)) = \mathtt{A}(\ell)$.

This dichotomy follows immediately by case analysis of the annotation validity definition.

The above dichotomy tells us that if we can follow the location ℓ upstream along the edges that have ℓ in the domain of their heap annotations, we will eventually reach the allocation node. Unfortunately, this only tells us that the allocation node is necessarily $(\mathtt{po} \cup \mathtt{rf})^+$-before a, and that alone is not enough to infer the happens-before relation between the allocation node and our node a. But, all is not lost! Definition 2.8 tells us what kind of $(\mathtt{po} \cup \mathtt{rf})^+$ paths between two nodes induce the $\mathtt{hb}$ relation, so what we need to do is be careful and make some accounting along the way while we track the location ℓ from its use by the access node a to its origin at the allocation node.

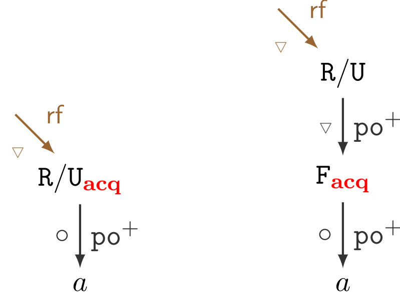
As we noted earlier, ℓ is present in the normal part of the heap annotated on $[\mathtt{po}, -, a]$. Starting from there, let us follow the location ℓ back along $\mathtt{po}$ edges as far as we can, meaning that when looking at the edges preceding e which have ℓ in the domain of their heap annotation, if possible we pick a $\mathtt{po}$ edge to continue building our path backwards toward the allocation. Clearly, if we manage to reach the allocation using only $\mathtt{po}$ edges, we established the desired happens-before relation, because $\mathtt{po} \subseteq \mathtt{hb}$. Let us therefore focus on the situation in which it is not possible to reach the allocation node via $\mathtt{po}$ edges alone.

We will start by following ℓ backwards from $[\mathtt{po}, -, a]$ as long as it stays in the domain of the normal heap annotated on the edges we are following. Once ℓ cannot be found on the normal part of the heap annotated on the preceding edges, we have to be in one of the following two cases according to the annotation validity definition:



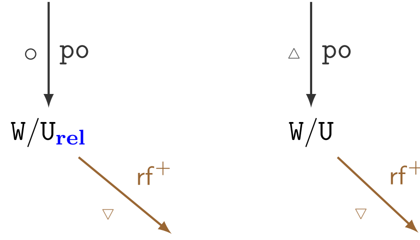
In our backwards travel, we find ℓ switching from the normal part of the heap ($\circ$) to the acquire part of the heap ($\triangledown$). It got there either after passing through a read or an update node, which has to be of **acq** or **acq_rel** kind, and we find ℓ on an $\mathtt{rf}$ edge now; or ℓ stayed on a $\mathtt{po}$ edge, but changed the heap position with the help of an acquire fence.

In the case where we encountered a fence, we can still continue to follow ℓ along $\mathtt{po}$ edges until it is forced to move to an $\mathtt{rf}$ edge. Annotation validity tells us that this can happen only by encountering an atomic read or update (of any atomic access type). We can update our visual representation of the two cases to look as follows:

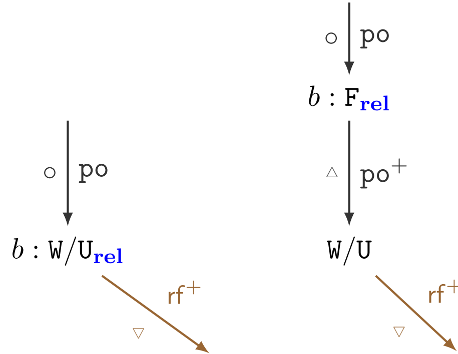


Both cases are now in the situation where we have our location ℓ present on an `rf` edge, so let us analyze the situation from there. Our strategy is to stay on `rf` edges only while we have to, and get back on a `po` edge as soon as we can.

Looking back at the validity definition, what are the situations in which we will be able to do so? It will have to be a store or an update node that moves us back to a `po` edge, and once again there are two subcases. We are either encountering a **rel** or **acq_rel** node that also shifts us back onto the normal heap; or we are seeing an atomic node (possibly **rlx**) that moves us away from the acquire heap, but this time on the release part of the heap (△). Visually, it looks like this:



In the case where ℓ is on the release part of the heap, we should continue tracking it back along the graph. From the annotation validity, we can see that it will eventually come back to the normal heap, and the switch will be helped along by a release fence, bringing our graph sketches to their final form.



Finally, in all the cases we have reached a node b such that the location ℓ is in the domain of the normal part of the heap $hmap([\mathtt{po}, -, b])$.

Notice that, combining the case analyses we have just done, we get the exact four synchronization patterns depicted in Figs. 2.1 and 2.2, meaning that there is synchronization between the points at which ℓ disappears from the normal part of the heap and the point at which it comes back to the normal heap. That synchronization, together with $\mathtt{po}^+ \subseteq \mathtt{hb}$ gives us $\mathtt{hb}(b, a)$.

Now, all we need to do is to prove that there is an allocation of ℓ which happens before b. Since ℓ is in the domain of the normal part of the heap of $hmap([\mathtt{po}, -, b])$, which is the exact same situation we initially had with $hmap([\mathtt{po}, -, a])$, we would like to proceed with the argument in the inductive fashion, but do we have some decreasing measure which we can use for the induction? Yes we do! The depth metric we used in the proof of Theorem 3.2 works here too. We have $\mathbf{D}(\{b\}) < \mathbf{D}(\{a\})$, so we can proceed by the induction on the depth of the node. □

Initialized Reads

Next, we will prove that in validly annotated executions all the loads are reading initialized locations.

Theorem 3.4 *If an execution* $(r, \mathcal{A}, \mathsf{lab}, \mathtt{po}, \mathtt{rf}, \mathtt{mo})$ *is validly annotated, then all the reads are initialized, i.e.,* $\operatorname{rng} \mathtt{rf} = \mathtt{R} \cup \mathtt{U}$*.*

Proof We need to prove that $\mathtt{rf}^{-1}$ is a total function, i.e., that every load reads from some store. Definition 2.9 (the second condition in the definition) tells us that it is enough to prove that for every read node there exists a write node which is $\mathtt{hb}$-before the read.

The main idea of the proof is exactly the same as in Theorem 3.3. We will start from the read, track back the location it is accessing, be careful which paths we take, bridge the parts where the location is not appearing on the normal part of the annotated heap by synchronizations, and proceed inductively on the depth of the node.

The main difference is that we are not looking for the allocation, but for initialization, so we will not be requiring only that the heap has the location in its domain, but that it maps the location to a resource telling us that the location is initialized. In the case of non-atomic locations it is important that the associated resource is $\mathsf{NA}[-, -]$, and not $\mathbb{U}$. In the case of atomic locations, the resource should be $\mathsf{AT}[-, -, -, 1]$, i.e., the initialization flag in the resource should be set to 1.

The rest of the proof boils down to a case analysis, similar (but slightly more technical) to the one executed in the proof of Theorem 3.3. The details can be seen in the Coq development. □

Data race freedom

Finally, we will show that data races do not happen in validly annotated executions.

Theorem 3.5 *A validly annotated execution contains no non-atomic data races.*

Proof Let us assume that we have an execution G, validly annotated by $hmap$, in which there is a non-atomic data race. According to Definition 2.14, that means that there is a location ℓ, and two events a and b in the execution G which are not related by the happens-before relation, such that both of those events are accessing the location ℓ, at least one of them is a store event, and at least one of them is non-atomic.

First thing we will note is that the annotation validity precludes a single location to be accessed with both atomic and non-atomic accesses. (This is because there is a single allocation node for each location in the execution, and that allocation node produces either atomic or non-atomic resources associated with the location. Later, non-atomic and atomic accesses require the presence of non-atomic and atomic resources, respectively.) Therefore, if one of the accesses to the location ℓ is non-atomic, then both of them are.

Let us now look at the store event. Without the loss of generality, we can say that is a, which means $\mathsf{lab}(a) = \mathtt{W}_{\mathrm{na}}(\ell, -)$. We know that it has to be a non-atomic store, because we just concluded that both a and b are non-atomic accesses.

According to the annotation validity definition $hmap([\mathtt{po}, -, a]) = (\{(\ell, \mathbb{U})\}, \emptyset, \emptyset, \emptyset) \oplus \mathbf{h}_F$ or $hmap([\mathtt{po}, -, a]) = (\{(\ell, \mathsf{NA}[-, \mathbb{1}])\}, \emptyset, \emptyset, \emptyset) \oplus \mathbf{h}_F$, for some heap $\mathbf{h}_F$. The important thing to note here is that ℓ being mapped to $\mathbb{U}$ or $\mathsf{NA}[-, \mathbb{1}]$ makes $hmap([\mathtt{po}, -, a]) \oplus \mathbf{h}$ undefined for any heap $\mathbf{h}$ which has ℓ in the domain of any of its components.

Since $hmap([\mathtt{po}, -, b])$ also has to have ℓ in the domain (of its normal component), then by Theorem 3.1 we can conclude that either $(\mathtt{po} \cup \mathtt{rf})^+(a, b)$ or $(\mathtt{po} \cup \mathtt{rf})^+(b, a)$.

We now want to employ a strategy similar to what we have done in the proofs of Theorems 3.3 and 3.4, that is to start at the $[\mathtt{po}, -, b]$ edge and follow the location ℓ along the edges in whose heap annotation it appears, until we hit the node a. Note that we are indeed guaranteed to hit the node a. In the case $(\mathtt{po} \cup \mathtt{rf})^+(a, b)$ we need to follow the location ℓ upstream along the graph, and in the case $(\mathtt{po} \cup \mathtt{rf})^+(b, a)$ we have to follow it downstream. In either case, we will be forced to eventually move along the $[\mathtt{po}, -, a]$ edge by Theorem 3.1 and irreflexivity of $(\mathtt{po} \cup \mathtt{rf})^+$ (and the fact that the execution graphs are finite).

There is one major difference in this proof compared to the ones we have seen so far. Namely, the depth metric is of no use because we are not guaranteed to track the location backwards along the graph. We need to find something else over which we can do the induction.

The trick is to specify the conditions which make a path behave like the one we constructed in the proof of Theorem 3.3 (synchronizations between the points where ℓ disappears from and comes back to the normal heap component of the heap annotations). Let us call those kind of paths "hb-preserving" paths. Then we establish that given an hb-preserving path p with a beginning (in the case $(\mathtt{po} \cup \mathtt{rf})^+(b, a)$) or the end (in the case $(\mathtt{po} \cup \mathtt{rf})^+(a, b)$) at node b, p either already connects a and b, or p can be extended to a longer hb-preserving path. This property can be proven by induction on the length of the path p.

At this point, the proof moves into the highly technical territory, formally specifying the hb-preserving paths and doing the induction, which once again requires a lengthy case analysis of the annotation validity definition. A reader interested in the details (especially on how to structure the proof to make all the technicalities manageable) should consult the Coq development. □

3.3.2. The Adequacy Theorem

Having explored the behavior of well-annotated executions, we are finally ready to prove the main adequacy theorem of the FSL proof system. The theorem states that a program for which some specification (with no assumptions on the precondition) can been proven to hold, does not go wrong, and the established postcondition holds of the result for every terminating execution of the program.

Theorem 3.6 (Adequacy) *Let E be a program, and Q a mapping from values to assertions such that $\{\mathsf{true}\}\, E\, \{y.\, Q\}$ holds. Then, E is a well-formed program, and if E can terminate with a return value r, then $Q(r)$ is satisfiable.*

Proof In order to prove that E is well-formed, we need to establish that it is memory safe, has no uninitialized reads, and is not racy. To do that, we have to prove that the corresponding properties hold of all the consistent executions of E.

Note that every consistent execution of E can be transformed into a minimal contextual execution of E, simply by adding the two requisite skip nodes into the graph. Vice versa, if we have a minimal contextual execution of E, removing the trivial context (i.e., the two skip nodes of which the entire context consists) leaves us with an execution of E.

$\{\mathsf{true}\}\, E\, \{y.\, \mathcal{Q}\}$ implies that every minimal contextual execution of E can be validly annotated, which enables us to apply Theorems 3.3 to 3.5 to conclude that E has to be well formed.

In the case where E terminates with some result r, the definition of triple semantics guarantees that a part of the heap annotated on the final po edge in the corresponding minimal contextual execution satisfies the assertion $\mathcal{Q}(r)$. □

Note that the adequacy theorem talks only about closed programs, i.e., the programs which have been fully verified and are not running inside some wider non-verified context. If we immerse a verified program inside a non-verified context, the program might go wrong due to the interactions with the unverified context.

3.3.3. Soundness of the Inference Rules

What remains to be proven is the soundness of the proof rules presented back in Section 3.1. Before looking into the proofs of specific inference rules, let us take a moment to observe the general structure of a soundness proof of an inference rule.

In general, a proof rule consists of some conditions under which the rule is valid, and a triple $\{P\}\, E\, \{y.\, Q\}$ which we need to prove holds whenever the conditions of the rule are satisfied. Definition 3.19 gives the meaning of triples in terms of the safety predicate, saying that given a contextual execution validly annotated up to some point (including the edge carrying the heap satisfying the precondition P), then it is safe to continue annotating the execution for arbitrarily many more steps. The proof that the triple $\{P\}\, E\, \{y.\, Q\}$ holds proceeds by induction on the number of annotation steps; essentially, assuming that the edges around n nodes have been annotated so far, we have to show that we can safely (as given by Definition 3.18) annotate one more node. There are two cases to consider here:

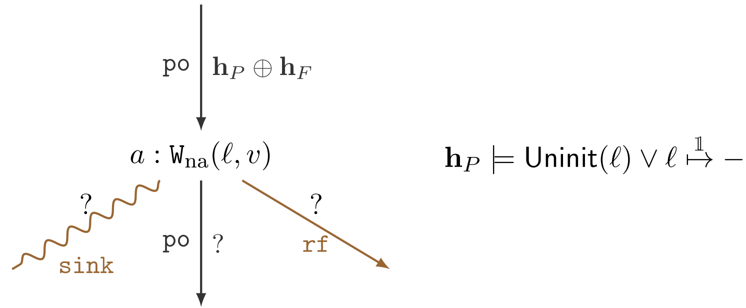
1. The node we are trying to annotate belongs to the context, in which case we do not actually have anything to prove, since the configuration safety is defined in a way that assumes the context to be well-behaved.

2. The node we are trying to annotate is a program node, and then we have to prove that there actually is a way to extend the existing annotation to cover the edges the newly added node is responsible for annotating, and we have to ensure that the annotations follow Definition 3.14. Furthermore, if the new node is the last node of the program, then the outgoing po edge has to be annotated by a heap satisfying the postcondition Q.

We will now see what the soundness proof looks like on a simple example by proving the soundness of the inference rules for non-atomic stores.

Theorem 3.7 *The inference rule* (W-NA) *holds universally.*

Proof Assuming that there is some contextual execution of $[\ell]_{\text{na}} = v$, and that some prefix-closed set V of nodes from that execution have been validly annotated so far (more precisely, the edges for which those nodes are responsible have been annotated), we have to show that we can annotate one more node. As discussed above, we can focus on the case when the new node is a program node, and not a context node. In this particular case, the program is $[\ell]_{\text{na}} = v$, and that program's execution consists of a single node. The situation we are in looks like this:



We are looking at a node a whose label is $\mathsf{lab}(a) = \mathtt{W}_{\text{na}}(\ell, v)$. Since $[\mathtt{po}, -, a]$ is the edge entering from the context into the execution of the program, we know that $\mathit{hmap}([\mathtt{po}, -, a]) = \mathbf{h}_P \oplus \mathbf{h}_F$, for some $\mathbf{h}_P$ satisfying the precondition and some frame heap $\mathbf{h}_F$.

We now need to define a new heap annotation function hmap' which will assign the same value to as hmap to all the edges except those a is responsible for (and the sink node coming out of a), i.e., those marked by '?' in the sketch, and then prove that the hmap' validly annotates a as well as all the nodes that have been validly annotated so far. The

annotation validity definition does not provide us with much choice, and we have to assign

$$\begin{aligned} hmap'([\mathsf{po}, a, -]) &:= (\{(\ell, \mathsf{NA}[v, \mathbb{1}])\}, \emptyset, \emptyset, \emptyset) + \mathbf{h}_F \\ hmap'([\mathtt{rf}, a, -]) &:= \emptyset \\ hmap'([\mathtt{sink}, a]) &:= \emptyset \end{aligned}$$

With those assignments, it is a trivial matter to check that all the validity conditions have been fulfilled.

The edge $[\mathsf{po}, a, -]$ is the edge leaving the program and entering the context, so we also need to prove that it has a heap satisfying the postcondition on it. Indeed

$$(\{(\ell, \mathsf{NA}[v, \mathbb{1}])\}, \emptyset, \emptyset, \emptyset) \models \ell \overset{\mathbb{1}}{\mapsto} v$$

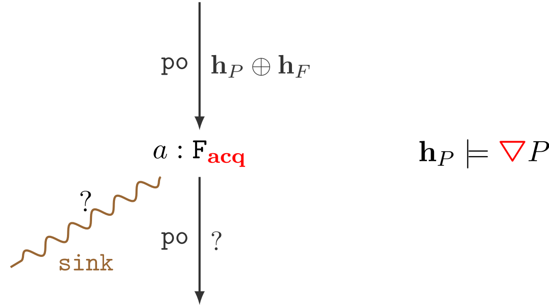
which completes our proof. □

The proof we have just seen boils down to simply defining the annotations the way the annotation validity asks us to do, and everything immediately falls into place. This proof pattern repeats itself for many other inference rules (atomic stores, allocations, and fences), so we will not go over most of those proofs.

Before moving on to the more intricate proofs, we will take time to look at the (F-REL) and (F-ACQ) rules. They provide us with a great opportunity to point out where and how the normalizability requirement gets used in the soundness proofs. The soundness of the two fence rules follows the exact same pattern, with the only difference that one needs the normalizability and the other does not. Let us see what is going on.

Theorem 3.8 *The inference rule* (F-ACQ) *holds universally.*

PROOF To prove the rule sound, we need to validly annotate the following simple execution.



Unfolding the definition of $\mathbf{h}_P \models \nabla P$, we see that

$$\mathbf{h}_P = (\emptyset, \emptyset, h, h_g)$$

for some resource heap h and ghost heap h_g, such that

$$(h, \emptyset, \emptyset, h_g) \models P\,.$$

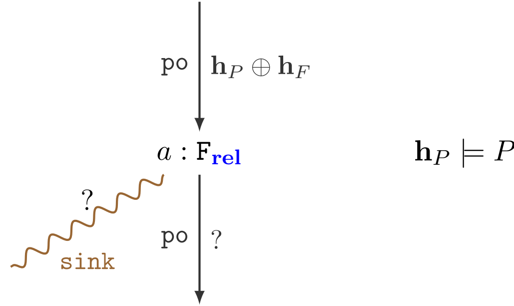
From here, it is immediately clear that setting

$$hmap'([\texttt{po}, a, -] := (h, \emptyset, \emptyset, h_g) \oplus \mathbf{h}_F$$
$$hmap'([\texttt{sink}, a] := \boldsymbol{\emptyset}$$

satisfies the validity conditions. □

Theorem 3.9 *The inference rule* (F-REL) *holds universally.*

Proof Analogously to the proof of Theorem 3.8, we need to validly annotate the following execution.



We need to put a heap satisfying $\triangle P$ on the $\texttt{po}(a, -)$ edge, i.e., a heap of the form $(\emptyset, h, \emptyset, h_g)$, such that $(h, \emptyset, \emptyset, h_g) \models P$. Unfortunately, from knowing only $\mathbf{h} \models P$ we cannot guarantee the existence of such h and h_g.

This is where the normalizability requirement comes in to save the day. Since P is normalizable and $\mathbf{h} \models P$, Definition 3.11 tells us that there exist h, h_g, and $\mathbf{h}'$ such that $\mathbf{h} = (h, \emptyset, \emptyset, h_g) \oplus \mathbf{h}'$ and $(h, \emptyset, \emptyset, h_g) \models P$.

We can now annotate

$$hmap'([\texttt{po}, a, -] := (\emptyset, h, \emptyset, h_g) \oplus \mathbf{h}' \oplus \mathbf{h}_F$$
$$hmap'([\texttt{sink}, a]) := \boldsymbol{\emptyset}$$

Notice that the edge $[\texttt{po}, a, -]$ holding the triple's postcondition now has an "extra" heap ($\mathbf{h}'$) on it. This is not a problem because we defined the triple semantics in such a way to be able to "forget" a part of the heap in the postcondition. □

Remark (On alternative to the normalizability condition) Note that we could have asked for a stricter condition on P. We could have asked that if $(h_\circ, h_\triangle, h_\triangledown, h_g) \models P$, then it has to be $h_\triangle = h_\triangledown = \emptyset$. This restriction would let us have the more traditional definition of the triple semantics, where we would not be allowing parts of the heap to be forgotten by the postcondition.

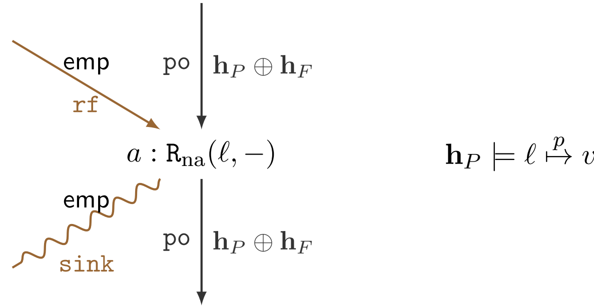
This stricter version of normalizability would not create any issues in practical applications, since the assertions we would be interested in transferring between threads

would fall under that definition as well. In fact, any extensions of FSL where forgetting parts of the heap in the postcondition might be problematic (e.g., if we want to reason about deallocation) would have to make a switch to the stricter normalizability criterion. Luckily, changing between the two versions of normalizability would be a very simple process, as the soundness proof works just as well with either version.

With the nice and easy proofs out of the way, let us tackle the rules that do most of the heavy lifting—the rules regarding loads and atomic updates.

Theorem 3.10 *The inference rule* (R-NA) *holds universally.*

PROOF We begin with the sketch of the annotations that not only exist so far, but that we would have to see after we finish annotating the node corresponding to the store instruction.



$\mathbf{h}_P \models \ell \overset{p}{\mapsto} v$

Seems easy enough. The heap $\mathbf{h}_P$ does indeed satisfy the postcondition and everything checks out. However, the devil is in the details, and we hit a significant roadblock when looking at the annotation validity of node a.

The annotations we sketched above are good only if $\mathsf{lab}(a) = \mathtt{R}_{\text{na}}(\ell, v)$, but we are not a priori given that the read event generated by our load instruction actually reads the value v. That is something we need to prove.

Proving that we do indeed have $\mathsf{lab}(a) = \mathtt{R}_{\text{na}}(\ell, v)$ is rather technical, but the idea is to utilize the resource-tracking technique which worked well in the proofs of Theorems 3.3 to 3.5. This time we are following the $\{(\ell, \mathsf{NA}[v, -]])\}$ resource until we reach the write node which set the value. By annotation validity rules that write node has to have the label $\mathtt{W}_{\text{na}}(\ell, v)$, and by Definition 2.9 we also get that the write node we reached is also $\mathtt{rf}^{-1}(a)$, thus establishing that the node a really does read the value v. □

The soundness proof of the non-atomic load rule (R-NA) had some subtleties to it, but in the end it came down to proving that the read actually reads the correct value. The atomic load rules are even more intricate, because they involve ownership transfer of resources along the rf edges, which has to respect the conditions given by the atomic heap resources carrying the Acq permissions. In order to reason about the ownership transfer, we have to first prove a lemma telling us about the structure of Rel and Acq permissions flowing through a validly annotated execution.

In the lemma statement below we are abusing the notation a bit, and for a FSL heap $\mathbf{h} = (h_\circ, h_\triangle, h_\triangledown, h_g)$ and a location ℓ we write $\mathbf{h}(\ell) := h_\circ(\ell) \oplus h_\triangle(\ell) \oplus h_\triangledown(\ell)$.

Lemma 3.3 *Let V be a prefix-closed set of nodes in a consistent execution G; let hmap be a heap annotation validly annotating the set V; and let $T \subseteq \mathcal{E}(G)$ be a set of pairwise independent edges, such that $\mathrm{fst}(e) \in V$ for every $e \in T$. If location ℓ is such that $\left(\bigoplus_{e \in T} hmap(e)\right)(\ell) = \mathsf{AT}[-, \mathcal{Q}, v, -]$, then for every node $w \in V$ such that $\mathsf{lab}(w) = \mathtt{W}_{-}(\ell, -)$ and $hmap([\mathsf{po}, -, w])(\ell) = \mathsf{AT}[\mathcal{R}, -, -, -]$, the following implication[1] holds:*

$$\mathcal{R}(v) \implies \mathcal{Q}(v) * \mathsf{true}\,.$$

Proof This is another proof heavy in technicalities, but the main idea is easy to understand. The structure of the proof is similar to the proof of Theorem 3.2. We are performing induction over the edge depth of the set $T \cup \{[\mathsf{po}, -, w]\}$, with the inductive step being replacing a deepest edge with all its immediate predecessor edges.

Instead of looking at the sum of the heaps annotated on the edges in the set (as we did in the proof of Theorem 3.2) we are going to look at release and acquire permissions associated with the location ℓ. Also, instead of going back all the way to the beginning of the execution, we are going to stop once we encounter the node which allocates the location ℓ.

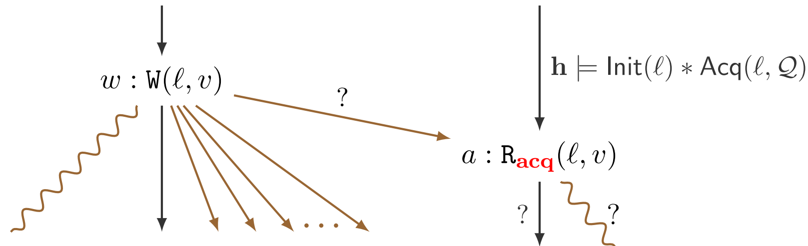
Due to the way the composition of atomic resources works (Definition 3.5), the release permission can only be weakened as we go upstream along the graph, while the acquire part is becoming an ever-growing $*$-conjunction of permissions we collect along the way.

Eventually, we reach the allocation node, and the annotation validity tells us that the release and acquire permissions generated at the allocation point have to be the same. From there we are able to conclude that the implication claimed by the lemma holds. □

Stated in more intuitive terms, Lemma 3.3 tells us that (in a validly annotated execution) if a certain reader r is attempting to acquire some resources from a writer w, then it is indeed the case that w released the resources r wants to pick up (and maybe some more). With this property, we will be able to clear the major hurdle in establishing the soundness of atomic load rules.

Theorem 3.11 *The inference rules* (R-ACQ) *and* (R-RLX) *hold universally.*

Proof As always, we start with the image showcasing the situation we need to annotate. Here, we are actually dealing with two different cases, depending on whether the load node is reading from a store or from an atomic update. The interesting case is the read from a store.



[1]Technically, $\mathcal{R}(v)$ and $\mathcal{Q}(v)$ are equivalence classes of assertions, but Proposition 3.1 enables us to simply take any representative of a class when using the equivalence classes in formulas.

We are trying to annotate the edges around the node a, which is reading from the node w, and w might have multiple other readers reading from it. Our job is to find out what to put on the edges marked by question marks.

The tricky part is to figure out how to annotate the $[\mathtt{rf}, w, r]$ edge without messing up the annotation validity of w. Once that is figured out, everything else falls into place, by simply making the choices forced upon us by the definition of annotation validity at a. But, how do we decide which heap should go on $[\mathtt{rf}, w, r]$?

According to the annotation validity requirements for a, we have to set the new annotation mapping, $hmap'$ such that $hmap'([\mathtt{rf}, w, a]) \models \triangledown\mathcal{Q}(v)$. To find a heap satisfying $\triangledown\mathcal{Q}(v)$ we are going to utilize the lemma we have just proven.

Applying Lemma 3.3 for location ℓ, the set $\{[\mathtt{po}, -, a]\}$, and the writer node w, we can deduce that w has indeed released enough resources. In particular, we know that the annotation mapping $hmap$ (the annotations established prior to a being included in the set of annotated nodes) satisfies

$$\mathbf{h} \coloneqq hmap([\mathtt{sink}, w]) \oplus \bigoplus_{\mathtt{rf}(w,x)} hmap([\mathtt{rf}, w, x]) \models \triangledown\mathcal{Q}(v) * \mathsf{true}\,.$$

So, $\mathbf{h} = \mathbf{h}_q \oplus \mathbf{h}'$, where $\mathbf{h}_q \models \triangledown\mathcal{Q}(v)$, and we are going to move the $\mathbf{h}_q$ heap to the $[\mathtt{rf}, w, a]$ edge, leaving $\mathbf{h}'$ on the other $[\mathtt{rf}, w, -]$ edges and the $[\mathtt{sink}, w]$ edge. Seems good, but how do we know we will not end up messing up the validity of other readers reading from w?

This is the point where the precision requirement comes into play. Due to the theorem assumption that $\mathcal{Q}(v)$ is precise (see Definition 3.12), and the validity requirement which ensure that all the heaps on the other $\mathtt{rf}$ edges from w to previously annotated nodes are also precise, we get to conclude that the heap $\mathbf{h}_q$ has to be located entirely on the $[\mathtt{sink}, w]$ edge, i.e., $hmap([\mathtt{sink}, w]) = \mathbf{h}_q \oplus \mathbf{h}''$ for some heap $\mathbf{h}''$. We can now define $hmap'$ by setting

$$hmap'([\mathtt{rf}, w, a]) \coloneqq \mathbf{h}_q\,,$$
$$hmap'([\mathtt{sink}, w]) \coloneqq \mathbf{h}''\,,$$

annotating the rest of the edges coming out of a as required by the validity definition, and leaving all the other edges annotated as they were by $hmap$. This concludes the case when a reads from a store event.

If a is reading from an atomic update, we can conclude that there will be no resource transfer. Theorem 3.1 and Definition 3.5 ensure that $\mathcal{Q}(v) = \mathsf{emp}$ in this case, which trivializes the work we need to do when annotating the edges around a. □

Interestingly enough, the most complicated rules—the ones regarding CAS-es—do not have the most complicated proofs; at least not conceptually.

Theorem 3.12 *The inference rules* (CAS-AR), (CAS-REL), (CAS-ACQ), (CAS-RLX), *and* (CAS-⊥) *hold universally.*

PROOF There are a couple of important properties that significantly simplify the proof compared to the one we saw when dealing with the atomic load rules:

- When there are atomic updates accessing some location, then (in a validly annotated execution) atomic loads from that location cannot transfer any ownership. This is the same property which trivialized the case of reading from an atomic update in the proof of Theorem 3.11.
- There cannot be multiple atomic updates reading from the same node. That is forbidden by the fourth condition in Definition 2.9.

This means that in the case of a successful CAS, we have a pretty straightforward job annotating the atomic update node generated by the CAS. We do not have to worry about messing up the validity of the other readers; we can simply take the entire resource released by the node we are reading from, and do with it as we please. All we have to take care of is distributing the heaps among the edges we need to annotate in the way dictated by the local validity definition, and we are done. □

We will close the chapter by giving a quick overview of soundness of the rules from Section 3.1.1, and the (GHOST-INTRO) rule.

The (VAL) rule boils down to validly annotating a `skip` node in a perfectly straightforward fashion.

(CONSEQ), (FRAME), (DISJ), and (∃-INTRO) follow directly from Definition 3.19.

The rest of the standard rules are proven by exploiting the assumptions about the triples which hold for the constituent parts of the program. Perhaps the only interesting moment in all those proofs is at the end of the proof of (PAR), where we need to annotate the skip node representing the two threads joining. At that point we need to establish that the heaps annotated on the incoming `po` edges are composable. We prove that by invoking Theorem 3.1.

The proof of (GHOST-INTRO) consists of taking a consistent annotations provided by $\{P\}\, C\, \{Q\}$ in the premise of the rule, and introducing the new ghost where appropriate to satisfy $\{P\}\, C\, \left\{Q * \exists\gamma.\, \boxed{\gamma : g}\right\}$. Adding a new ghost is a trivial matter, as the facilities for doing so are baked into the annotation validity definition.

4. Case Study: Verification of Atomic Reference Counter

4.1. Algorithm

Atomic reference counter (ARC) (Rust Team, 2015a) is a part of the standard library of the Rust programming language (Rust Team, 2015b), providing 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. Even though ARC is a rather simple algorithm, it uses many interesting C11 features, such as relaxed memory accesses and memory fences. ARC also heavily relies on release sequences to ensure synchronization among concurrent threads. This makes ARC an ideal example of application of FSL to a real-world algorithm.

Our ARC implementation is given in Fig. 4.1 and consists of four functions: `new`, `read`, `drop`, and `clone`. To gain a basic understanding of the algorithm, we will first ignore the access types and fences.

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 which 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 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[1] (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 a separation-logic style, as in Fig. 4.2. In this specification, $\mathsf{ARC}(a, v)$ represents the permission to run functions which access the ARC object a. This permission is created by the function `new`, duplicated by `clone`, and destroyed by `drop`.

[1] Keep in mind that `fetch_and_add` returns the value stored at the location prior to the modification done by the function.

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

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

read(a){
  return a.data;
}

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

Figure 4.1.: Atomic reference counter implementation.

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

Figure 4.2.: ARC specification in separation logic.

4.1.1. 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. 4.2 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.

Since FSL is designed to ensure (among other things) memory safety, formalizing the ARC specification in FSL (and proving it, of course), will provie us with the desired correctness guarantees.

Before proceeding with the formal verification, let us get some understanding of why is ARC correct from the standpoint of programing language models. We will first take a quick look at the classic sequentially conisistent case, and then consider the C11 model.

Sequential Consinstency

From the perspective of 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

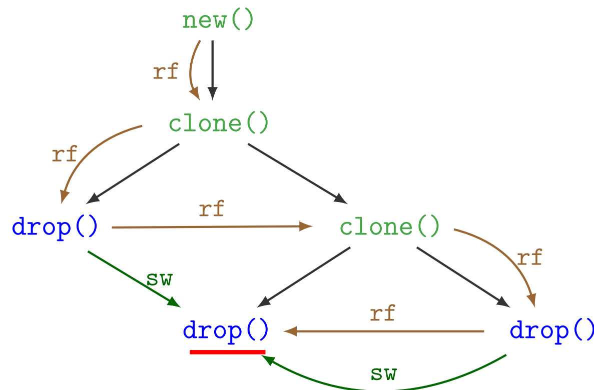


Figure 4.3.: An example execution of the ARC algorithm.

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

In the C11 model, the reasoning becomes significantly more complex.

Looking back at the ARC algorithm in Fig. 4.1 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. In this discussion, for the sake of simplicity, we will treat `fetch_and_add` instruction as if it is emitting a single atomic update event.

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. 4.3. 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`. (In Rust, this is ensured by `drop` being called as the destructor of the ARC object.)

In a single run of the ARC algorithm, there will be a single chain of atomic updates terminating with the decrement from one to zero, which is then followed by the acquire fence. Because of synchronization through release sequences (Fig. 2.2), the acquire fence will synchronize with all the release updates in the chain, i.e., all other drop functions happen before the one which does the deallocation.

4.2. Verification of ARC

The implementation shown in Fig. 4.1 uses some features not present in the programming language assumed by FSL.

- Function `new` allocates and returns a structure with two fields. This can be simulated by allocating two locations, and then returning a single value encoding the addresses

of the allocated locations. (The solution implemented in the Coq development.) For the sake of presentation simplicity, we will not bother with those details.

- Function `drop` uses `fetch_and_add` as decrement by adding -1 to `a.count`. Formally, the values in our programming language are natural numbers, which does not include -1. This could be solved by either modifying the programming language, or by making `fetch_and_add` more general by allowing it to make arbitrary modifications (instead of just the addition). The latter approach is taken in the Coq development.

- Function `drop` uses memory deallocation, which is not only not present in the programming language we considered, but is not even present in the C11 model as presented in Chapter 2. To go around this restriction, we treat the call to the `free` function as a no-operation, and our specification will explicitly mention the memory being leaked. We discuss a way for handling deallocation in Section 4.2.3.

Having cleared up potential confusion between the presented ARC implementation and the formalities behind FSL, we can proceed with the verification.

4.2.1. Defining the ARC Predicate

We rely on the ghost state to do the accounting of ARC instances and ensuring that the deallocation happens only when all the ARC objects have been dropped. To use the ghost state we need to describe a partial commutative monoid which will represent our ghost resources.

Lemma 4.1 (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 denoting that we have 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-SPLIT)) having more than one would lead to a contradiction.

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

Definition 4.1 (ARC invariant) For location x, value v, and ghost locations γ and δ, we define the mapping from values to assertions

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

where $x \overset{0}{\mapsto} v \equiv \mathsf{emp}$.

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 enables us to reconstitute the full permission after all the ARC objects have been dropped. How this happens will become clear in Section 4.2.2.

Lastly, the least complicated part of the invariant, the ghost state attached to the ghost location δ counts how much of the access permission to x is shared by the still active ARC objects. This will be used in Section 4.2.2 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.

Definition 4.2 (ARC Predicate) For ghost locations γ and δ, we define

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

where $\mathsf{U}(x, \mathcal{Q}) \equiv \mathsf{RMWAcq}(x, \mathcal{Q}) * \mathsf{Rel}(x, \mathcal{Q}) * \mathsf{Init}(x)$ and $x \overset{0}{\mapsto} v \equiv \mathsf{emp}$.

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 keep 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 $a.\texttt{count}$.

4.2.2. The ARC Correctness Theorem

The following theorem contains the formal correctness statement for ARC.

Theorem 4.1 (Correctness of ARC) *With the* $\mathsf{ARC}_{\gamma,\delta}$ *defined as in Definition 4.2, the following holds.*

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

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.`data`. If we were modeling deallocation, having the full permission for a location would be enough to deallocate it.

An additional thing of note is that we prove 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.

In what follows, we are going to discuss the main points of the proof of Theorem 4.1 for each of the functions from the ARC algorithm. Full formal proofs are available in the Coq formalization.

Function `new`

In Fig. 4.4 you can see a simplified version of the proof for the function `new`.

At the beginning, we have to introduce two ghosts (γ and δ) 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-RMW) rule. Naturally, we will choose the mapping defined in Definition 4.1 as the invariant governing the `a.count` location.

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

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

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

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

$$\{\mathsf{emp}\}$$
$$\texttt{a = alloc();}$$
$$\left\{\begin{array}{c}\mathsf{RMWAcq}(\texttt{a.count}, \mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}) * \mathsf{Rel}(\texttt{a.count}, \mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}) * \\ \mathsf{Uninit}(\texttt{a.data}) * \boxed{\gamma : 0^-} * \boxed{\delta : 0^-}\end{array}\right\}$$
$$\texttt{[a.data]}_{\text{na}}\ \texttt{=}\ v\texttt{;}$$
$$\left\{\begin{array}{c}\mathsf{RMWAcq}(\texttt{a.count}, \mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}) * \mathsf{Rel}(\texttt{a.count}, \mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}) * \\ \texttt{a.data} \overset{1}{\hookrightarrow} v * \boxed{\gamma : 0^-} * \boxed{\delta : 0^-}\end{array}\right\}$$
$$\Downarrow \text{(using (GHOST-SPLIT), and } \texttt{a.data} \overset{0}{\hookrightarrow} v \iff \mathsf{emp})$$
$$\left\{\begin{array}{c}\mathsf{RMWAcq}(\texttt{a.count}, \mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}) * \mathsf{Rel}(\texttt{a.count}, \mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}) * \\ \texttt{a.data} \overset{1}{\hookrightarrow} v * \texttt{a.data} \overset{0}{\hookrightarrow} v * \boxed{\gamma : 0^-} * \boxed{\gamma : 0^+} * \boxed{\delta : 1^-} * \boxed{\delta : 1^+}\end{array}\right\}$$
$$\Downarrow \text{(using (GHOST-MOD), and } \mathsf{emp} \iff \triangle\mathsf{emp})$$
$$\left\{\begin{array}{c}\mathsf{RMWAcq}(\texttt{a.count}, \mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}) * \mathsf{Rel}(\texttt{a.count}, \mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}) * \\ \texttt{a.data} \overset{1}{\hookrightarrow} v * \boxed{\gamma : 0^+} * \boxed{\delta : 1^+} * \triangle\left(\texttt{a.data} \overset{0}{\hookrightarrow} v * \boxed{\gamma : 0^-} * \boxed{\delta : 1^-}\right)\end{array}\right\}$$
$$\texttt{[a.count]}_{\text{rlx}}\ \texttt{= 1;}$$
$$\left\{\begin{array}{c}\mathsf{RMWAcq}(\texttt{a.count}, \mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}) * \mathsf{Rel}(\texttt{a.count}, \mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}) * \mathsf{Init}(\texttt{a.count}) \\ \texttt{a.data} \overset{1}{\hookrightarrow} v * \boxed{\gamma : 0^+} * \boxed{\delta : 1^+}\end{array}\right\}$$
$$\texttt{return a;}$$
$$\{\mathsf{ARC}_{\gamma,\delta}(\mathsf{a}, v)\}$$

Figure 4.4.: Function `new`: FSL proof

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 $\texttt{a.data} \overset{0}{\hookrightarrow} v \iff \mathsf{emp} \iff \triangle\mathsf{emp}$.

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

Before releasing $\texttt{a.data} \overset{0}{\hookrightarrow} v * \boxed{\gamma : 0^-} * \boxed{\delta : 1^-}$, 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 (because we need it as a part of the $\mathsf{ARC}_{\gamma,\delta}$ predicate).

We can now finally release the required resource, and what we are left with is exactly the ARC predicate from Definition 4.2, with the existentially quantified q set to be 1.

Function `read`

Verifying `read` is trivial. The predicate from Definition 4.2 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`.

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. 3.5) 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 $(q > 0 \wedge \boxed{\delta : q^+} * \boxed{\delta : 0^-} \implies \mathsf{false})$, and we can use the (CAS-$\bot$) rule to conclude that the value 0 could not have been read.

Now that we know the value read is not 0, we need to somehow produce an additional ARC predicate.

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 predicate we already have, allows us to "produce" an additional ARC predicate. In order to do that, we have to use the following three equivalences:

$$\texttt{a.data} \overset{q}{\mapsto} v \iff \texttt{a.data} \overset{\frac{q}{2}}{\mapsto} v * \texttt{a.data} \overset{\frac{q}{2}}{\mapsto} v\,,$$
$$\boxed{\gamma : (1-q)^+} * \boxed{\gamma : 1^+} \iff \boxed{\gamma : \left(1-\tfrac{q}{2}\right)^+} * \boxed{\gamma : \left(1-\tfrac{q}{2}\right)^+}, \text{ and}$$
$$\boxed{\delta : q^+} \iff \boxed{\delta : \tfrac{q}{2}^+} * \boxed{\delta : \tfrac{q}{2}^+}.$$

Using those equivalences, it is easy to see that the implication

$$\mathsf{ARC}_{\gamma,\delta}(\texttt{a.data}, v) * \boxed{\gamma : 1^+} \implies \mathsf{ARC}_{\gamma,\delta}(\texttt{a.data}, v) * \mathsf{ARC}_{\gamma,\delta}(\texttt{a.data}, v)$$

holds.

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.

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. We are now left with two distinct cases.

First case is when 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

$$\mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}(c) * \texttt{a.data} \overset{q}{\mapsto} v * \boxed{\gamma : (1-q)^{+}} * \boxed{\delta : q^{+}} \implies \mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}(c-1)$$

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 (through 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 to have $q + f = 1$, and in that case we have

$$\boxed{\gamma : (1-q)^{+}} * \boxed{\delta : q^{+}} * \boxed{\gamma : f^{-}} * \boxed{\delta : (1-f)^{-}} \implies \boxed{\gamma : 0^{-}} * \boxed{\delta : 0^{-}},$$

establishing the $\mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}(0)$ invariant.

While establishing the $\mathcal{Q}_{\gamma,\delta,v,\texttt{a.data}}(0)$ 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 $\texttt{a.data} \overset{q}{\mapsto} v * \triangledown \texttt{a.data} \overset{f}{\mapsto} v$ 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 using the (F-ACQ) rule, the resource we own is transformed into

$$\texttt{a.data} \overset{q}{\mapsto} v * \texttt{a.data} \overset{f}{\mapsto} v \iff \texttt{a.data} \overset{q+f}{\mapsto} v \iff \texttt{a.data} \overset{1}{\mapsto} v\,.$$

The last equivalence holds because we know $q + f = 1$, as proven earlier.

With this, the proof of Theorem 4.1 is concluded.

4.2.3. Dealing With Deallocation

The FSL's soundness proof 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 mostly a 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.

The biggest interventions in the soundness proof presented in Section 3.3 would come from the way in which the RC11 model would be modified to handle deallocation. Assuming an extension of the model which would require that whenever a location is beaing allocated for what is not the first allocation of that location, the allocaton in question has to happen after (in terms of the `hb` relation) the last deallocation; none of the key results from Section 3.3 would be affected, but aditional technical case analysis would be needed in the proofs.

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 Section 4.2.2).

Part II.

Multi-Execution Models

5. Promising Semantics

Here we provide an overview of the promising semantics (Kang et al., 2017). We will restrict our attention to the fragment of the semantics modeling only relaxed accesses, since the logic presented in Chapter 6 deals only with the relaxed accesses. The presentation of the model is kept at a high-level, without going into many technical specifics, as the full formal (and very detailed) description of the model considered here can be found in (Kang et al., 2017, §2)

The promising semantics is an operational semantics that interleaves execution of the threads of a program. Relaxed behavior is introduced in two ways:

- The memory is a pool of timestamped messages, and each thread maintains a "view" thereof. A thread may read any value that is not older than the latest value observed by the thread for the given location; in particular, this may well not be the latest value written to that particular location. Timestamps and views model *non-multi-copy-atomicity*: writes performed by one thread do not become simultaneously visible by all other threads.
- The operational semantics contains a non-standard step: at any point a thread can nondeterministically *promise* a write, provided that, at every point before the write is actually performed, the thread can *certify* the promise, that is, execute the write by running on its own from the current state. Promises are used to enable load-store reordering.

5.1. Memory and Messages

Formally, the semantics keeps track of writes and promises in a *global configuration*, $gconf = \langle M, P \rangle$, where M is a memory and $P \subseteq M$ is the *promise memory*. We denote by $gconf.\mathtt{M}$ and $gconf.\mathtt{P}$ the components of $gconf$.

Memory is a finite sets of messages, where a *message* is a tuple $\langle x :_i v, t \rangle$, with $x \in \mathsf{Loc}$ being the location of the message, $v \in \mathsf{Val}$ its value, $i \in \mathsf{Tid}$ its originating thread, and $t \in \mathsf{Time}$ its *timestamp*. Time is an infinite set of timestamps, densely totally ordered by $\leqslant$, with a minimum element, 0, and with no maximal element. We denote by $m.\mathtt{loc}$, $m.\mathtt{tid}$, $m.\mathtt{val}$, $m.\mathtt{time}$ the components of a message m.

A global configuration $gconf$ evolves in two ways. First, a message can be "promised" and be added both to $gconf.\mathtt{M}$ and $gconf.\mathtt{P}$. Second, a message can be written, in which case it is either added to $gconf.\mathtt{M}$, or removed from $gconf.\mathtt{P}$ (if it was promised before).

Remark Here we deviate slightly from Kang et al. (2017, §2). Instead of keeping a separate promise memory for each thread, we keep the information about which thread

issued which message in the memory. The change is inconsequential for the promising semantics, but makes handling the semantics of the logic in Section 6.2 easier.

5.2. Threads

A *thread state* is a pair $TS = \langle \sigma, V \rangle$, where σ is the internal state of the thread and V is a *view*. We denote by $TS.\sigma$ and $TS.V$ the components of TS.

Thread internal state The internal state σ consists of a thread store (denoted $\sigma.\mu$) that assigns values to local registers and a statement to execute (denoted $\sigma.s$). The transitions of the thread internal state are labeled with *memory actions* (i.e., reads and writes) and are given by an ordinary sequential semantics.

Views Thread views are used to enforce coherence, that is, the existence of a per-location total order on writes that reads respect. A view is a function $V : \mathsf{Loc} \to \mathsf{Time}$, which records how far the thread has seen in the history of each location. To ensure that a thread does not read stale messages, its view restricts the messages the thread may read—a thread may read only messages whose timestamps are not smaller than its current view—and is increased whenever a thread observes a new message. We denote the set of all views by View.

The interaction between threads and memory The interaction between a thread and the message storage is given in terms of transitions of *thread configurations*. Thread configurations are tuples $\langle TS, gconf \rangle$, where TS is a thread state, and $gconf$ is a global configuration. We label the transitions with $\beta \in \{\mathrm{NP}, \mathrm{prom}\}$ in order to distinguish whether they involve promises or not. A thread i can:

- Make an internal transition with no effect on the storage subsystem.

$$\frac{\sigma \xrightarrow{\text{internal}} \sigma'}{\langle\langle \sigma, V \rangle, gconf \rangle \xrightarrow{\mathrm{NP}}_i \langle\langle \sigma', V \rangle, gconf \rangle} \qquad (\textsc{internal})$$

- Read the value v from location x, when there is a matching message in memory that is not outdated according to the thread's view. Note that this allows a thread to read writes promised by other threads. It then updates its view accordingly, setting the timestamp for the location x to the timestamp of the observed message.

$$\frac{\sigma \xrightarrow{\mathsf{R}(x,v)} \sigma' \qquad \langle x :_{_} v, t \rangle \in gconf.M \qquad V(x) \leqslant t}{\langle\langle \sigma, V \rangle, gconf \rangle \xrightarrow{\mathrm{NP}}_i \langle\langle \sigma', V[x{:=}t] \rangle, gconf \rangle} \qquad (\textsc{read})$$

- Write the value v to location x. Here, the thread picks a timestamp greater than the one of its current view for the message it adds to memory (or removes from the

promise set) and updates its view to the chosen timestamp.

$$\frac{\sigma \xrightarrow{\mathsf{W}(x,v)} \sigma' \qquad V(x) < t \qquad \langle x :_i v, t\rangle \notin M}{\langle\langle \sigma, V\rangle, \langle M, P\rangle\rangle \xrightarrow{\text{NP}}_i \langle\langle \sigma', V[x{:=}t]\rangle, \langle M \uplus \{\langle x :_i v, t\rangle\}, P\rangle\rangle} \quad (\textsc{write})$$

$$\frac{\sigma \xrightarrow{\mathsf{W}(x,v)} \sigma' \qquad \langle x :_i v, t\rangle \in P \qquad V(x) < t}{\langle\langle \sigma, V\rangle, \langle M, P\rangle\rangle \xrightarrow{\text{NP}}_i \langle\langle \sigma', V[x{:=}t]\rangle, \langle M, P \setminus \{\langle x :_i v, t\rangle\}\rangle\rangle} \quad (\textsc{fulfill-promise})$$

- Nondeterministically promise a write by adding a message to both *gconf.M* and *gconf.P*.

$$\frac{\langle x :_i v, t\rangle \notin M}{\langle TS, \langle M, P\rangle\rangle \xrightarrow{\text{prom}}_i \langle TS, \langle M \uplus \{\langle x :_i v, t\rangle\}, P \uplus \{\langle x :_i v, t\rangle\}\rangle\rangle} \quad (\textsc{promise})$$

5.3. Constraining Promises

Now that we have described how threads and promises interact with memory, we can present the certification condition for promises, which is essential to avoid out-of-thin-air behaviors. Accordingly, we define another transition system, $\Longrightarrow$, on top of the previous one, which enforces that the memory remains "consistent", i.e., all the promises that have been made can be certified.

A thread configuration $\langle TS, gconf\rangle$ is called *consistent* with respect to $i \in \mathsf{Tid}$ if thread i can fulfill its promises by executing on its own, or more formally if

$$\langle TS, gconf\rangle \xrightarrow{\text{NP}}{}^{*}_{i} \langle TS', gconf'\rangle$$

for some TS' and $gconf'$ such that

$$\{m \in gconf.P \mid m.\mathtt{tid} = i\} = \emptyset\,.$$

Certification is *local*, that is, only thread i is executing during its certification.

Note that the locality of certifications plays the crucial role in dealing with out-of-thin-air values. Consider again the LB+DEP program from Fig. 1.2. If certifications were not local, then the left thread could promise $x = 1$ and certify it by a possible run in which the right thread reads from that promise, sets y to 1; and then the left thread continues running, reads $y = 1$ which was just written by the other thread, and finally fulfills its promise. This kind of behavior is prevented by disallowing the progress of other threads during certifications runs.

On the other hand, LB and LB+FAKEDEP programs from Fig. 1.2 produce their load buffering behavior just fine. The left thread can, at the very beginning of the execution, promise a write of 1 to x because it can, running on its own from the current state, read from y (it will read 0), then write 1 to x, thereby fulfilling its promise. The certification of the promise did not require any help from other threads, so it does nor run afoul of the locality constraint.

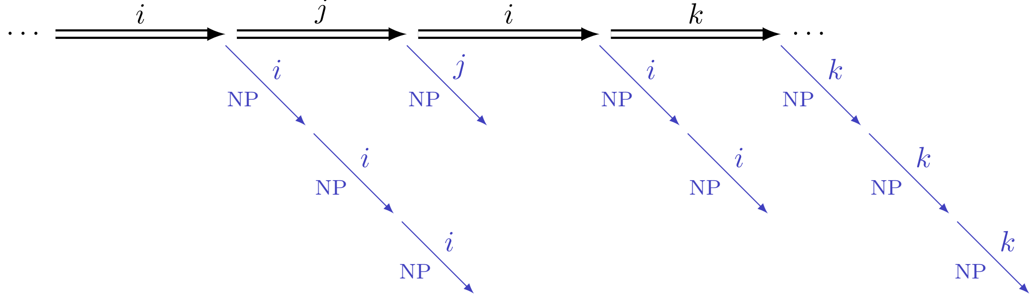


Figure 5.1.: A visual representation of a promise machine run, together with promise certifications. The certifications are local (i.e., all the steps are taken by the thread whose promises are being certified), but they can branch away from the main execution.

One might say that the locality of the promise certifications prevents threads from conspiring to produce nonsensical values.

The promise certification is precisely what makes the promising semantics a multi-execution model. During the certifications, the machine has to demonstrate that there exists *some* execution which results in the promise being fulfilled, but there is absolutely no further requirement that the actual run is in any way related to the certification run. For example, certifications are free to take branches which will later not be taken by the real run.

The thread configuration $\Longrightarrow$-transitions allow a thread to (1) take any number of non-promising steps, provided its thread configuration at the end of the sequence of step (intuitively speaking, when it gives control back to the scheduler) is consistent, or (2) take a promising step, again provided that its thread configuration after the step is consistent. The diagram in Fig. 5.1 illustrates this idea.

5.4. Full Machine

Finally, the full machine transitions simply lift the thread configuration $\Longrightarrow$-transitions to the machine level. A *machine state* is a tuple $\mathbf{MS} = \langle \mathcal{TS}, gconf \rangle$, where $\mathcal{TS}$ is a function assigning a thread state TS to every thread, and *gconf* is a global configuration. The initial state $\mathbf{MS}^0$ (for a given program) consists of the function $\mathcal{TS}^0$ mapping each thread i to its initial state $\langle \sigma_i^0, \bot \rangle$, where σ_i^0 is the thread's initial local state and $\bot$ is the zero view (all timestamps in views are 0); the initial memory M^0 consisting of one message $\langle x :_0 0, 0 \rangle$ for each location x; and the empty set of promises.

6. Weak Separation Logic

In this chapter we take a look at *weak separation logic* (Weasel), whose main feature is the ability to facilitate non-trivial reasoning about relaxed accesses without reliance on restrictions which outlaw observed load-buffering patters shown in Fig. 1.2. The program execution model under which Weasel is sound is the promising semantics presented in Chapter 5.

Weasel is a fragment of SLR (Svendsen et al., 2018), a logic which can not only reason about relaxed accesses, but also provides constructs supporting release-acquire resource transfer. For the purposes of this thesis, we constrain ourselves to the relaxed fragment only and briefly discuss the full SLR logic in Section 6.4.

The reason for presenting this particular fragment is that it covers all the contributions[1] of the thesis' author in the collaborative effort of developing SLR, while still being an interesting logic in its own right.

6.1. Syntax

6.1.1. Programming Language

Weasel deals exclusively with relaxed accesses, so the programming language will be trimmed down accordingly. We will consider the programming language containing only sequential composition, branches, loops, register assignments, and relaxed memory accesses.

Definition 6.1 (Programming language) The expressions of the language are

$$\begin{array}{rcll} e \in \mathsf{Expr} & ::= & n \in \mathbb{N} & \text{integer constants} \\ & | & r & \text{registers} \\ & | & e_1 \ op\ e_2 & \text{arithmetic,} \end{array}$$

and the statements of the programming language are given by

$$\begin{array}{rl} s \in \mathsf{Stm} ::= & \textbf{skip} \mid s_1; s_2 \mid \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \mid \textbf{while}\ e\ \textbf{do}\ s \\ & \mid r = e \mid r = [e]_{\mathbf{rlx}} \mid [e_1]_{\mathbf{rlx}} = e_2 \end{array}$$

Note that the language does not contain parallel composition. We will be looking at programs where each thread is a statement in the given programming language, and the threads are all executing concurrently, following the promising semantics discussed in Chapter 5. Essentially, we are considering the parallel composition as a top-level construct between program statements.

[1] $\mathsf{O}(x, v, t)$ and $\mathsf{W}(x, \mathcal{S})$ assertions; the semantics of $\mathsf{O}(x, v, t)$ assertions; the general structure and outline of the soundness proof; some of the example proofs using the logic.

6.1.2. The Assertions of the Logic

If we want to be able to reason about relaxed accesses in absence of any synchronization, the only property we can rely on is *coherence*, i.e., the guarantee that for each location there is an order of stores to that particular location agreed to by all of the threads. Thus, our logic contains assertions that describe this agreed-upon ordering of writes.

Definition 6.2 (Assertion grammar) The assertions of the logic are given by the following grammar

$$\begin{aligned} P, Q \in \mathsf{Assn} ::= {} & \bot \mid \top \mid P \vee Q \mid P \wedge Q \mid P \rightarrow Q \mid P * Q \\ & \mid \forall x.\, P \mid \exists x.\, P \mid \forall v.\, P \mid \exists v.\, P \mid \forall t.\, P \mid \exists t.\, P \\ & \mid \mathsf{O}(x, v, t) \mid \mathsf{W}^{\pi}(x, \mathcal{S}), \end{aligned}$$

Where $x \in \mathsf{Loc} \coloneqq \mathbb{N}$, $v \in \mathsf{Val} \coloneqq \mathbb{N}$, $t \in \mathsf{Time}$, $\mathcal{S} \subseteq \mathsf{Val} \times \mathsf{Time}$, and $\pi \in (0, 1]$.

The grammar contains the standard operators from first order logic and separation logic, and a couple of novel constructs.

The first novel assertion, $\mathsf{O}(x, v, t)$, called *observation* records the fact that location x was observed (i.e., read) to have value v at timestamp t. The timestamp is used to order it with respect to other reads from the same location. The information this assertion provides is very weak: it merely says that the owner of the assertion has observed that value, it does not imply that any other thread has ever seen it.

The other novel assertion, $\mathsf{W}^{\pi}(x, \mathcal{S})$, called *write permission*, gives the thread a permission to write to location x and records a set of writes $\mathcal{S}$ to that location. The parameter $\pi \in (0, 1]$ acts as a fractional permission. The full permission, $\pi = 1$, indicates exclusive ownership of the location x, i.e., no other thread can concurrently modify x. Thus, $\mathsf{W}^{1}(x, \mathcal{S})$ means that $\mathcal{S}$ is the exact set of writes that have been made to location x. Any $\pi < 1$ indicates shared ownership and enforces that $\mathcal{S}$ is a lower-bound on the set of writes to location x. Timestamps are used to track the order of writes to x; this is why $\mathcal{S}$ is a set of pairs consisting of the value and the timestamp of the write.

In cases where we are interested in reasoning about the ordering of writes without needing to explicitly talk about timestamps, we use the shorthand

$$\mathsf{W}^{\pi}(x, \ell) \;\equiv\; \exists t_1, ..., t_n.\, t_1 > t_2 > \cdots > t_n \wedge \mathsf{W}^{\pi}(x, \{(v_1, t_1), ..., (v_n, t_n)\}),$$

where $\ell = [v_1, \ldots, v_n]$ is a list of values. The $\mathsf{W}^{\pi}(x, \ell)$ gives the values written to x as a list, with the most recent write at the front of the list.

Below, we list some important properties of write permissions and observations. Many of them feature a *view shift* ($\Rrightarrow$) instead of a logical implication ($\Longrightarrow$). View shift can be thought of as a "generalized implication" where we get to draw conclusions not only based on the semantic interpretation of the assertions, but also using the information about which concrete machine state corresponds to the abstract state modeling the assertion. (See Section 6.2.2 for further discussion of view shifts.)

Properties of Write Permissions

Two partial write permissions can be combined into a single permission by adding the corresponding fractional permissions and taking the union of the set of performed writes:

$$\mathsf{W}^{\pi_1}(x,\mathcal{S}_1) * \mathsf{W}^{\pi_2}(x,\mathcal{S}_2) \iff \begin{cases} \mathsf{W}^{\pi_1+\pi_2}(x,\mathcal{S}_1 \cup \mathcal{S}_2) & \text{if } \pi_1+\pi_2 \leqslant 1, \\ \bot & \text{otherwise.} \end{cases} \tag{Combine-Writes}$$

Different values must have been written at different timestamps:

$$\mathsf{W}^{\pi}(x,\mathcal{S}) \wedge (v,t) \in \mathcal{S} \wedge (v',t) \in \mathcal{S} \Rrightarrow \mathsf{W}^{\pi}(x,\mathcal{S}) \wedge v = v'\,. \tag{Different-Writes}$$

Timestamps are totally ordered:

$$\mathsf{W}^{\pi}(x,\mathcal{S}) \wedge (_,t) \in \mathcal{S} \wedge (_,t') \in \mathcal{S} \implies \mathsf{W}^{\pi}(x,\mathcal{S}) \wedge (t < t' \vee t = t' \vee t' < t)\,. \tag{Writes-Ordered}$$

Relationship Between Write Permissions and Observations

Since the write permission holds a set of writes to a location, it provides us with stronger knowledge than what the observations give us. This is formalized by

$$\mathsf{W}^{\pi}(x,\mathcal{S}) \wedge (v,t) \in \mathcal{S} \Rrightarrow \mathsf{W}^{\pi}(x,\mathcal{S}) \wedge \mathsf{O}(x,v,t)\,, \tag{Write-Observed}$$

stating that we can take any pair $(v,t) \in \mathcal{S}$, and turn it into an observation $\mathsf{O}(x,v,t)$.

The information gathered in write permissions and observations can also be used to draw conclusions about the observed values and their timestamps:

$$\mathsf{W}^{1}(x,\mathcal{S}) \wedge \mathsf{O}(x,a,t) \Rrightarrow \mathsf{W}^{1}(x,\mathcal{S}) \wedge \mathsf{O}(x,a,t) \wedge (a,t) \in \mathcal{S}\,. \tag{Reads-from-Write}$$

Notice that we have to have the full write permission ($\pi = 1$) for (Reads-from-Write) to hold. Otherwise it might be the case that the observation is of a value written by some other writer. Only in the case when we have the authoritative information about all the writes performed (provided by the full write permission) can we conclude that that the observed value has to appear in the set of the written values recorded by the write permission.

6.1.3. The Rules for Relaxed Accesses

The logic contains all the standard separation logic rules, analogous to those listed in Section 3.1.1, with two notable differences

First, the consequence rule is stated in terms of view shifts, instead of in terms of logical implications.

$$\frac{P' \Rrightarrow P \qquad \{P\}\, E\, \{y.\, Q\} \qquad Q \Rrightarrow Q'}{\{P'\}\, E\, \{y.\, Q'\}} \tag{CONSEQ}$$

Second, there is no parallel composition rule, since our language does not have a parallel composition construct. Instead of the parallel composition rule, we have the correctness theorem (Theorem 6.3), telling us that we get to combine all the resources at the end of the program.

We now turn our attention to the rules governing relaxed accesses.

Relaxed Writes

To write value v (to which the value expression e_2 evaluates) to location x (to which the location expression e_1 evaluates), the thread needs to own a write permission $\mathsf{W}^\pi(x, \mathcal{S})$. The write rule updates the record of writes with the value written, timestamped with a timestamp newer than any timestamp for that location that the thread has observed so far; this is expressed by relating it to a previous timestamp that the thread has to provide through an $\mathsf{O}(x, v', t)$ assertion in the precondition.

$$\begin{array}{c} \left\{e_1 = x \wedge e_2 = v \wedge \mathsf{W}^\pi(x, \mathcal{S}) \wedge \mathsf{O}(x, v', t)\right\} \\ [e_1]_{\mathbf{rlx}} = e_2 \\ \left\{\exists t_{new}.\, \mathsf{W}^\pi(x, \{(v, t_{new})\} \cup \mathcal{S}) \wedge \mathsf{O}(x, v', t) \wedge t < t_{new}\right\} \end{array} \quad \text{(W-RLX)}$$

In practice, $\mathsf{O}(x, v', t)$ is taken to be the observation with the largest timestamp available, either among those already in the precondition, or observations which can be obtained by (Write-Observed).

Relaxed Reads

When reading from location x (to which the location expression e evaluates), the thread obtains an observation $\mathsf{O}(x, r, t_{new})$ stating that it has read the value now in register r from location x, timestamped with t_{new}. This timestamp is no older than any timestamp for that location that the thread has observed so far, expressed again by relating it to an observation $\mathsf{O}(x, v, t)$ from the precondition.

$$\begin{array}{c} \left\{e = x \wedge \mathsf{O}(x, v, t)\right\} \\ r = [e]_{\mathbf{rlx}} \\ \left\{\exists t_{new}.\, \mathsf{O}(x, v, t) \wedge \mathsf{O}(x, r, t_{new}) \wedge t \leqslant t_{new}\right\} \end{array} \quad \text{(R-RLX)}$$

There is no worry that a read will not be able to be performed due to the absence of the requisite observation assertion. Each thread can start with having the $\mathsf{O}(x, 0, 0)$ observation if necessary. This is sound because the promising machine starts with all locations initialized to 0.

Moreover, if a thread owns the exclusive write permission for a location x, then it can take advantage of the fact that it is the only writer at that location to obtain more precise information about its reads from that location—they will read the last value written to the location:

$$\begin{array}{c} \left\{e = x \wedge \mathsf{W}^1(x, \mathcal{S})\right\} \\ r = [e]_{\mathbf{rlx}} \\ \left\{\exists t.\, (r, t) = \max(\mathcal{S}) \wedge \mathsf{W}^1(x, \mathcal{S}) \wedge \mathsf{O}(x, r, t)\right\} \end{array} \quad \text{(R-RLX*)}$$

where $\max(\mathcal{S})$ denotes the greatest element in the set $\mathcal{S}$ according to the ordering of the timestamps.

6.1.4. Examples

As the examples of how the rules presented above can be used, we will look at the verification of several *litmus tests* (Mador-Haim et al., 2010, 2011; Maranget et al., 2012), small code snippets which are used as a sort of sanity checks to see if a memory model is providing expected guarantees.

CoRW

The CoRW[1] litmus test requires that the behavior suggested by the comments in the program below is not possible.

$$\begin{array}{l} a = [x]_{\mathbf{rlx}};\ \text{// reads 2} \\ [x]_{\mathbf{rlx}} = 1 \end{array} \;\Bigg\|\; [x]_{\mathbf{rlx}} = 2 \;\Bigg\|\; \begin{array}{l} b = [x]_{\mathbf{rlx}};\ \text{// reads 1} \\ c = [x]_{\mathbf{rlx}}\ \ \text{// reads 2} \end{array} \tag{CoRW}$$

The point is that the first thread sets x to 1 after first seeing the value 2, while the third thread first sees the value 1 and then the value 2 for the location x. This behavior breaks coherence because the first and the third thread disagree on the order of writes to the location x.

The proof that the forbidden behavior indeed does not happen is shown in Fig. 6.1. The first thread observes a value a and then sets x to 1, noting (according to (W-RLX)) that the timestamp of the write has to be strictly greater than the timestamp associated with the previously observed value a. The second thread only records that the value 2 was written. The third thread observes values b and c, and takes note of the ordering of their timestamps; in this case we do not get a strict inequality because we are getting it from (R-RLX). At the end of the program we have (keeping only relevant assertions)

$$\mathsf{W}^1(x, \{(0,0), (1,t_1)(2,t_2)\}) \wedge \mathsf{O}(x,a,t_a) \wedge \mathsf{O}(x,b,t_b) \wedge \mathsf{O}(x,c,t_c) \wedge t_a < t_1 \wedge t_b \leqslant t_c\,,$$

and we are interested in the potential outcome where $a = 2$, $b = 1$, and $c = 2$. Substituting those values in the above formula we have

$$\mathsf{W}^1(x, \{(0,0), (1,t_1)(2,t_2)\}) \wedge \mathsf{O}(x,2,t_a) \wedge \mathsf{O}(x,1,t_b) \wedge \mathsf{O}(x,2,t_c) \wedge t_a < t_1 \wedge t_b \leqslant t_c\,.$$

By way of (Reads-from-Write) we get

$$\begin{aligned} \mathsf{W}^1(x, \{(0,0), (1,t_1)(2,t_2)\}) \wedge \mathsf{O}(x,2,t_a) &\implies t_a = t_2 \\ \mathsf{W}^1(x, \{(0,0), (1,t_1)(1,t_2)\}) \wedge \mathsf{O}(x,1,t_b) &\implies t_b = t_1 \\ \mathsf{W}^1(x, \{(0,0), (1,t_1)(1,t_2)\}) \wedge \mathsf{O}(x,2,t_c) &\implies t_c = t_2\,, \end{aligned}$$

giving us the inequalities

$$t_2 < t_1 \wedge t_1 \leqslant t_2\,,$$

which is a contradiction. Thus we conclude that the forbidden behavior of (CoRW) cannot happen.

[1] CoRW = **Co**herence **R**ead-**W**rite, referring to the ordering of the instructions in the first thread.

$$
\begin{array}{c}
\left\{\mathsf{W}^{\frac{1}{2}}(x,\{(0,0)\})\right\} \\
a = [x]_{\mathbf{rlx}}; \\
\left\{\begin{array}{l}\exists t_a.\,\mathsf{O}(x,a,t_a) \land \\ \mathsf{W}^{\frac{1}{2}}(x,\{(0,0)\})\end{array}\right\} \\
[x]_{\mathbf{rlx}} = 1 \\
\left\{\begin{array}{l}\exists t_a, t_1.\,\mathsf{O}(x,a,t_a) \land t_a < t_1 \\ \land\, \mathsf{W}^{\frac{1}{2}}(x,\{(0,0),(1,t_1)\})\end{array}\right\}
\end{array}
\;\Bigg\|\;
\begin{array}{c}
\left\{\mathsf{W}^{\frac{1}{2}}(x,\{(0,0)\})\right\} \\
[x]_{\mathbf{rlx}} = 2 \\
\left\{\begin{array}{l}\exists t_2.\,0 < t_2 \land \\ \mathsf{W}^{\frac{1}{2}}(x,\{(0,0),(2,t_2)\})\end{array}\right\}
\end{array}
\;\Bigg\|\;
\begin{array}{c}
\{\mathsf{O}(x,0,0)\} \\
b = [x]_{\mathbf{rlx}}; \\
\{\exists t_b.\,\mathsf{O}(x,b,t_b)\} \\
c = [x]_{\mathbf{rlx}} \\
\left\{\begin{array}{l}\exists t_b, t_c.\,t_b \leqslant t_c \land \\ \mathsf{O}(x,b,t_b) \land \mathsf{O}(x,c,t_c)\end{array}\right\}
\end{array}
$$

$$
\begin{array}{c}
\left\{\begin{array}{l}\exists t_1, t_2, t_a, t_b, t_c.\,\mathsf{W}^{1}(x,\{(0,0),(1,t_1)(2,t_2)\}) \land \\ \mathsf{O}(x,a,t_a) \land \mathsf{O}(x,b,t_b) \land \mathsf{O}(x,c,t_c) \land t_a < t_1 \land 0 < t_2 \land t_b \leqslant t_c\end{array}\right\} \\
\Downarrow \text{ (using (Reads-from-Write))} \\
\{\neg(a = 2 \land b = 1 \land c = 2)\}
\end{array}
$$

Figure 6.1.: Proof of the (CoRW) example

CoWR

The CoWR[1] litmus test is similar to the previous example. Disagreement on ordering of writes is again forbidden, and the difference is that now the first thread starts by writing to the shared location, and then observes a different value from it.

$$
\begin{array}{l}
[x]_{\mathbf{rlx}} = 1; \\
a = [x]_{\mathbf{rlx}}; \ \textit{// reads 2}
\end{array}
\;\Big\|\;
[x]_{\mathbf{rlx}} = 2
\;\Big\|\;
\begin{array}{l}
b = [x]_{\mathbf{rlx}}; \ \textit{// reads 2} \\
c = [x]_{\mathbf{rlx}} \ \ \textit{// reads 1}
\end{array}
\qquad \text{(CoWR)}
$$

The proof in Figure 6.2 is very similar to the one we saw in Figure 6.1. In fact the annotations of the second and third threads is exactly the same. This time the first thread first sets x to 1 and then observes a value a, thus we get a weaker inequality on the timestamps.

The behavior we want to exclude is $a = 2$, $b = 2$, and $c = 1$. As in the previous example, (Reads-from-Write) gives us $t_a = t_2$, $t_b = t_2$, and $t_c = t_1$, providing us with inequalities

$$t_1 \leqslant t_2 \land t_2 \leqslant t_1 \,,$$

implying $t_1 = t_2$. However, using (Different-Writes) we get

$$\mathsf{W}^{1}(x,\{(0,0),(1,t_1),(2,t_2)\}) \implies 1 = 2$$

which shows that the undesired behavior does not happen.

CoRR2

Continuing with the theme of coherence tests we move onto the CoRR2[2] litmus test. Here we have two concurrent writers setting the shared location to two different values; and

[1] CoWR = **Co**herence **W**rite-**R**ead, referring to the ordering of the instructions in the first thread.

$$
\left\{\mathsf{W}^{\frac{1}{2}}(x,\{(0,0)\})\right\}\;
[x]_{\mathbf{rlx}} = 1\;
\left\{\begin{array}{l}\exists t_1.\ \wedge\\ \mathsf{W}^{\frac{1}{2}}(x,\{(0,0),(1,t_1)\})\end{array}\right\}\;
a = [x]_{\mathbf{rlx}};\;
\left\{\begin{array}{l}\exists t_a, t_1.\, \mathsf{O}(x,a,t_a) \wedge t_1 \leqslant t_a\\ \wedge\, \mathsf{W}^{\frac{1}{2}}(x,\{(0,0),(1,t_1)\})\end{array}\right\}
\;\Bigg\|\;
\left\{\mathsf{W}^{\frac{1}{2}}(x,\{(0,0)\})\right\}\;
[x]_{\mathbf{rlx}} = 2\;
\left\{\begin{array}{l}\exists t_2.\, 0 < t_2 \wedge\\ \mathsf{W}^{\frac{1}{2}}(x,\{(0,0),(2,t_2)\})\end{array}\right\}
\;\Bigg\|\;
\{\mathsf{O}(x,0,0)\}\;
b = [x]_{\mathbf{rlx}};\;
\{\exists t_b.\, \mathsf{O}(x,b,t_b)\}\;
c = [x]_{\mathbf{rlx}}\;
\left\{\begin{array}{l}\exists t_b, t_c.\, t_b \leqslant t_c \wedge\\ \mathsf{O}(x,b,t_b) \wedge \mathsf{O}(x,c,t_c)\end{array}\right\}
$$

$$
\left\{\begin{array}{l}\exists t_1, t_2, t_a, t_b, t_c.\, \mathsf{W}^1(x,\{(0,0),(1,t_1),(2,t_2)\}) \wedge\\ \mathsf{O}(x,a,t_a) \wedge \mathsf{O}(x,b,t_b) \wedge \mathsf{O}(x,c,t_c) \wedge t_1 \leqslant t_a \wedge 0 < t_2 \wedge t_b \leqslant t_c\end{array}\right\}
$$

$$\Downarrow \text{(using (Reads-from-Write))}$$

$$\{\neg(a = 2 \wedge b = 2 \wedge c = 1)\}$$

Figure 6.2.: Proof of the (CoWR) example

$$
\left\{\mathsf{W}^{\frac{1}{2}}(x,\{(0,0)\})\right\}\;
[x]_{\mathbf{rlx}} = 1\;
\left\{\exists t_1.\, \mathsf{W}^{\frac{1}{2}}(x,\{(0,0),(1,t_1)\})\right\}
\;\Big\|\;
\left\{\mathsf{W}^{\frac{1}{2}}(x,\{(0,0)\})\right\}\;
[x]_{\mathbf{rlx}} = 2\;
\left\{\exists t_2.\, \mathsf{W}^{\frac{1}{2}}(x,\{(0,0),(2,t_2)\})\right\}
$$

$$
\{\mathsf{O}(x,0,0)\}\;
a = [x]_{\mathbf{rlx}};\;
\{\exists t_a.\, \mathsf{O}(x,a,t_a)\}\;
b = [x]_{\mathbf{rlx}}\;
\{\exists t_a, t_b.\, \mathsf{O}(x,a,t_a) \wedge \mathsf{O}(x,b,t_b) \wedge t_a \leqslant t_b\}
\;\Big\|\;
\{\mathsf{O}(x,0,0)\}\;
c = [x]_{\mathbf{rlx}};\;
\{\exists t_c.\, \mathsf{O}(x,c,t_c)\}\;
d = [x]_{\mathbf{rlx}}\;
\{\exists t_c, t_d.\, \mathsf{O}(x,c,t_c) \wedge \mathsf{O}(x,d,t_d) \wedge t_c \leqslant t_d\}
$$

Figure 6.3.: Proof of the (CoRR2) example

two reader threads, each reading from the shared location twice. The test states that the reader threads should not observe the writes in different orders.

$$
[x]_{\mathbf{rlx}} = 1 \;\Big\|\; [x]_{\mathbf{rlx}} = 2 \;\Big\|\; \begin{array}{l} a = [x]_{\mathbf{rlx}};\ \text{// reads 1}\\ b = [x]_{\mathbf{rlx}}\ \ \text{// reads 2}\end{array} \;\Big\|\; \begin{array}{l} c = [x]_{\mathbf{rlx}};\ \text{// reads 2}\\ d = [x]_{\mathbf{rlx}}\ \ \text{// reads 1}\end{array} \tag{CoRR2}
$$

The proof outline is shown in Fig. 6.3. The writer threads record the writes, while the reader threads note the order of timestamps for their observed values. Once again, for the undesirable behavior ($a = 1$, $b = 2$, $c = 2$, and $d = 1$) at the end of the program, we get $t_1 \leqslant t_2$ and $t_2 \leqslant t_1$, which is not possible because t_1 and t_2 are timestamps of two distinct writes.

[2] CoRR2 = **Co**herence **R**ead-**R**ead with **2** reader threads

Random Number Generator

As our final example, look at the program

$$\begin{array}{l} a = [x]_{\mathbf{rlx}}; \\ [y]_{\mathbf{rlx}} = a + 1 \end{array} \;\Big\|\; \begin{array}{l} b = [y]_{\mathbf{rlx}}; \\ [x]_{\mathbf{rlx}} = b \end{array} \tag{RNG}$$

involving two shared locations x and y. The first thread reads a value a from the location x, and then sets y to the incremented value $a + 1$. The second thread simply copies the value it read from y into x.

Clearly, the only sensible value the first thread can read from x is 0, because it should not be able to observe its own writes from the future. This means that the second thread can only read 0 or 1 from y, depending on whether it reads the initial value or the value stored by the first thread. However, some early attempts at resolving the out-of-thin-air problem allowed the threads in the (RNG) program to observe arbitrarily large values, thus giving the example the moniker *"random number generator"*.

In Fig. 6.4 we can see the proof that the random number generator does not actually generate random numbers, but only the expected values. In the proof we are not concerned with the timestamps, but only with the values written and observed. The first thread observes a value a from x, and records the write of $a+1$ to y. The second thread observes y to have value b and records the write of b to x. At the end of the program we have

$$\mathsf{W}^1(y, [a+1, 0]) * \mathsf{W}^1(x, [b, 0]) \wedge \mathsf{O}(x, a, -) \wedge \mathsf{O}(y, b, -)\,.$$

Applying (Reads-from-Write) we get

$$\begin{aligned} \mathsf{W}^1(y, [a+1, 0]) \wedge \mathsf{O}(y, b, -) &\implies b \in \{a+1, 0\} \\ \mathsf{W}^1(x, [b, 0]) \wedge \mathsf{O}(x, a, -) &\implies a \in \{b, 0\} \end{aligned}$$

from where it is trivial to conclude

$$a = 0 \wedge (b = 0 \vee b = 1)\,.$$

6.2. Semantics

6.2.1. Semantics of Assertions

The two Weasel's novel assertions, observations ($\mathsf{O}(x, v, t)$) and write permissions ($\mathsf{W}^\pi(x, \mathcal{S})$), are what we have to focus on. Once we nail down an appropriate semantics for those assertions, the semantics can be extended to the rest of the assertions in the standard way.

The observations are pretty straightforward. We can interpret $\mathsf{O}(x, v, t)$ as saying that a message with the corresponding location, value, and timestamp exists in the memory of the promise machine's global configuration ($\mathit{gconf}.M$).

We can also try to model write permissions using the global configurations. At first glance, it seems that $\mathsf{W}^\pi(x, \mathcal{S})$) simply means that the memory $\mathit{gconf}.M \setminus \mathit{gconf}.P$ contains

$$
\begin{array}{c||c}
\{\mathsf{W}^1(y,[0]) \land \mathsf{O}(x,0,0)\} & \{\mathsf{W}^1(x,[0]) \land \mathsf{O}(y,0,0)\} \\
a = [x]_{\mathbf{rlx}}; & b = [y]_{\mathbf{rlx}}; \\
\{\mathsf{W}^1(y,[0]) \land \mathsf{O}(x,a,-)\} & \{\mathsf{W}^1(x,[0]) \land \mathsf{O}(y,b,-)\} \\
[y]_{\mathbf{rlx}} = a+1 & [x]_{\mathbf{rlx}} = b \\
\{\mathsf{W}^1(y,[a+1,0]) \land \mathsf{O}(x,a,-)\} & \{\mathsf{W}^1(x,[b,0]) \land \mathsf{O}(y,b,-)\}
\end{array}
$$

$$
\begin{array}{c}
\{\mathsf{W}^1(y,[a+1,0]) * \mathsf{W}^1(x,[b,0]) \land \mathsf{O}(x,a,-) \land \mathsf{O}(y,b,-)\} \\
\Downarrow \text{(using (Reads-from-Write))} \\
\{a \in \{b,0\} \land b \in \{a+1,0\}\} \\
\Downarrow \text{(by case analysis)} \\
\{a = 0 \land (b = 0 \lor b = 1)\}
\end{array}
$$

Figure 6.4.: Proof of the (RNG) example

all the messages mentioned by $\mathcal{S}$. We know that the messages come from executed writes; therefore they are not in the set of outstanding promises. Unfortunately, by looking at the problem a bit deeper, we see that this approach is not good enough.

Intuitively, the full write permission $\mathsf{W}^1(x,\mathcal{S})$ lists *all* the messages with the location x in $gconf.M \setminus gconf.P$, while a partial permission like $\mathsf{W}^{\frac{1}{2}}(x,\mathcal{S})$ lists *some* of the messages to the location x. We also want the equivalence

$$\mathsf{W}^1(x,\mathcal{S}_1 \cup \mathcal{S}_2) \iff \mathsf{W}^{\frac{1}{2}}(x,\mathcal{S}_1) * \mathsf{W}^{\frac{1}{2}}(x,\mathcal{S}_2)$$

to hold. On the left-hand side we have the statement saying that $\mathcal{S}_1 \cup \mathcal{S}_2$ is listing *all* the messages for the location x, and on the right-hand side there are two statements saying that $\mathcal{S}_1$ lists *some* messages and $\mathcal{S}_2$ lists *some* messages. Clearly, twice *some* does not constitute *all*; at least not without some additional information.

Weasel is a separation logic, so we cannot really attach some extra information to $\mathsf{W}^\pi(x,\mathcal{S})$, saying something about the messages listed by potential other write permissions for x. Doing so would break the core idea of separation, and we would find ourselves in all kinds of trouble when attempting to define the meaning of the separating conjunction.

Our solution is to give up on the idea of modeling write permissions using machine configurations. Instead, we model them using abstract resources defined in Definition 6.3, and leave the problem of relating those abstract resources to concrete machine states for some later time.

Definition 6.3 (Write resources) Let $\mathsf{Write} \coloneqq \mathcal{P}(\mathsf{Val} \times \mathsf{Time})$ be the powerset of the set of pairs of values and timestamps. The set

$$\mathsf{WrPerm} \coloneqq \{r \mid r\colon \mathsf{Loc} \to \{(\pi,\mathcal{S}) \in [0,1] \times \mathsf{Write} \mid \pi = 0 \to \mathcal{S} = \emptyset\}\}$$

is called the set of *write resources*. The composition of write permission resources is given by the pointwise-lifting of the partial operation

$$(\pi,\mathcal{S}) \oplus (\pi',\mathcal{S}') \coloneqq \begin{cases} (\pi+\pi', \mathcal{S} \cup \mathcal{S}') & \text{if } \pi + \pi' \leqslant 1 \\ \textbf{undefined} & \text{otherwise} \end{cases}$$

to the elements of WrPerm.

Beyond just accounting for messages in the memory, observations and write permissions provide us with information about the ordering of timestamps, as can be seen in (R-RLX) and (W-RLX). To facilitate that kind of reasoning, the semantics should also connect the observations to thread views. However, we want to avoid the mess of tying assertions with particular threads in the assertion semantics. In order to let the assertions talk about views without specifying the exact thread the view belongs to, the semantic structure modeling Weasel's assertions contains a nondescript view which should be thought of as "the current thread's view". The connection of the views mentioned by the assertion semantics to the views of the concrete program threads is established later in Definition 6.7.

Definition 6.4 Assertion semantics Let GConf be the set of global configurations of the promising machine, and View the set of view functions. The assertion semantics

$$[\![\cdot]\!]\colon \mathsf{Assn} \to \mathcal{P}(\mathsf{GConf} \times \mathsf{View} \times \mathsf{WrPerm})$$

is given by the structural recursion on assertions

$$\begin{aligned}
[\![\mathsf{O}(x, v, t)]\!] &\coloneqq \{(gconf, V, r) \mid \exists i.\, \langle x :_i v, t\rangle \in gconf.M \wedge t \leqslant V(x)\}\\
[\![\mathsf{W}^{\pi}(x, \mathcal{S})]\!] &\coloneqq \{(gconf, V, r) \mid r = \{(x, (\pi, \mathcal{S}))\} \wedge \mathrm{snd}(\max(\mathcal{S})) \leqslant V(x)\}\\
[\![P * Q]\!] &\coloneqq \{(gconf, V, r_1 \oplus r_2) \mid (gconf, V, r_1) \in [\![P]\!] \wedge (gconf, V, r_2) \in [\![Q]\!]\}
\end{aligned}$$

and all other logical connectives and quantifiers interpreted in the standard way.

To summarize, Definition 6.4 gives the following semantics to observations and write permissions:

- $\mathsf{O}(x, v, t)$ says that the memory contains a message at location x with value v and timestamp t, and the current thread knows about it (i.e., the thread view of x is at least at the timestamp of the message).

- The write assertion $\mathsf{W}^{\pi}(x, X)$ asserts ownership of a (partial, with fraction π) write resource at location x, and requires that the largest timestamp recorded in X does not exceed the view of the current thread.

6.2.2. Erasure and View Shift

As discussed earlier, write permissions are modeled using abstract resources. Such a definition is enough to establish the (Combine-Writes) property, but it does not provide us with any semantic connection between write permissions and observations.

Observations are modeled directly by the promising machine configuration, so, if we want to have a connection between write permissions and observations, we need a way to relate the abstract write resources with the underlying machine's concrete state. The concept of erasure defined in Definition 6.5 does just that.

Definition 6.5 (Erasure) Let $r_F \colon \mathsf{Tid} \to \mathsf{WrPerm}$ and $r = \bigoplus_{i \in \mathsf{Tid}} r_F(i)$. The erasure of r_F is

$$\begin{aligned}\lfloor r_F \rfloor \coloneqq \{ gconf \mid\ & (\forall x.\ \{(m.\mathtt{val}, m.\mathtt{time}) \mid m \in (gconf.M)_x \setminus (gconf.P)_x\} = \operatorname{snd}(r(x))) \wedge \\ & (\forall m \in gconf.M.\ \operatorname{fst}(r_F(m.\mathtt{tid})(m.\mathtt{loc})) > 0)\}\,,\end{aligned}$$

where $(\mathcal{M})_x \coloneqq \{m \in \mathcal{M} \mid m.\mathtt{loc} = x\}$.

The erasure takes a mapping r_F, telling us which thread owns which write resources and gives us the set of promise machine configurations consistent with those resources. Erasure is asking that the configuration satisfies two things:

1. The set of non-promise messages corresponds exactly to the set of location and value pairs mentioned in the abstract resources.
2. Only threads which are assigned some non-zero fraction of the write permission to a certain location can produce messages associated with that location.

Using erasure we can formalize the notion of view shifts, first mentioned in Section 6.1.2.

Definition 6.6 (View shift) The *view shift function* $vs \colon \mathcal{P}(\mathsf{GConf} \times \mathsf{View} \times \mathsf{WrPerm}) \to \mathcal{P}(\mathsf{GConf} \times \mathsf{View} \times \mathsf{WrPerm})$ is given by

$$\begin{aligned}vs(\mathcal{Q}) \coloneqq \{(gconf, V, r) \mid\ & \forall i \in \mathsf{Tid}, f \in \mathsf{WrPerm}, r_F \colon \mathsf{Tid} \to \mathsf{WrPerm}\,, \\ & gconf \in \lfloor r_F[i \coloneqq r \oplus f] \rfloor \wedge \mathrm{wf}_{\mathrm{prom}}(i, gconf, V) \to \\ & \exists r'.\, (gconf, V, r') \in \mathcal{Q} \wedge gconf \in \lfloor r_F[i \coloneqq r' \oplus f] \rfloor\}\,,\end{aligned}$$

where

$$\mathrm{wf}_{\mathrm{prom}}(i, gconf, V) \equiv \forall m \in gconf.P.\ m.\mathtt{tid} = i \to V(m.\mathtt{loc}) < m.\mathtt{time}\,.$$

The *view shift relation* between assertions is defined as

$$P \Rrightarrow Q \quad \text{if and only if} \quad [\![P]\!] \subseteq vs([\![Q]\!])\,.$$

View shift allows us to switch between abstract resources corresponding to the same concrete state in a frame-preserving way. Definition 6.6 is saying that if we have some resource r', composable with *frame* given by r_F, we can replace it with r as long as

1. if r is composable with some frame r_F, then the original resource r' has to also be composable with r_F (think of r_F as the resources owned by the other threads);
2. the global concrete state is unaffected by changing our resource from r' to r (formalized using erasure); and
3. all the promises of the thread owning the resource r' are in the future (i.e., the timestamp of the promises is greater than the timestamp given by the view V).

In the theorem below, we show that (Reads-from-Write), which we used as a motivating example for interpreting write permission with abstract resources, can indeed be expressed in terms of view shifts.

Theorem 6.1

$$\mathsf{W}^1(x,\mathcal{S}) \land \mathsf{O}(x,a,t) \Rrightarrow \mathsf{W}^1(x,\mathcal{S}) \land \mathsf{O}(x,a,t) \land (a,t) \in \mathcal{S}$$

PROOF Take an arbitrary $(gconf, V, r) \in [\![\mathsf{W}^1(x,\mathcal{S}) \land \mathsf{O}(x,a,t)]\!]$. From Definition 6.4 we know that

$$\begin{aligned} r &= \{(x,(1,\mathcal{S}))\}\,, \\ \mathrm{snd}(\max(\mathcal{S})) &\leqslant V(x)\,, \\ \langle x :_i a, t\rangle &\in gconf.M, \text{ for some } i \in \mathsf{Tid}, \text{ and} \\ t &\leqslant V(x)\,. \end{aligned}$$

We have to prove that

$$(gconf, V, r) \in vs([\![\mathsf{W}^1(x,\mathcal{S}) \land \mathsf{O}(x,a,t) \land (a,t) \in \mathcal{S}]\!])\,.$$

To that end, take an arbitrary j, f, and $r_F\colon \mathsf{Tid} \to \mathsf{WrPerm}$ such that

$$\begin{aligned} &r_F(j) = r \oplus f \text{ and,} \\ &gconf \in \lfloor r_F \rfloor\,. \end{aligned}$$

Note that $j = i$. That follows from the fact that $r = \{(x,(1,\mathcal{S}))\}$, which means any resource composable with r cannot have any write permissions associated with x, and then Definition 6.5 tells us that it had to have been the thread i which wrote the message m.

Now we need to find r' such that

$$\begin{aligned} (gconf, V, r') &\in [\![\mathsf{W}^1(x,\mathcal{S}) \land \mathsf{O}(x,a,t) \land (a,t) \in \mathcal{S}]\!] \text{ and,} \\ gconf &\in \left\lfloor r_F[i \coloneqq r' \oplus f]\right\rfloor\,. \end{aligned}$$

Clearly, setting $r' = r$ would work perfectly, if only we could prove that $(a,t) \in \mathcal{S}$.

From $gconf \in \lfloor r_F \rfloor$, we already know that $m \in (gconf.M)_x$. The erasure will also give us $(a,t) \in \mathcal{S}$ if we establish $m \notin (gconf.P)_x$. We already noticed that $m.\mathtt{tid} = i$, so $m \in (gconf.P)_x$ would be a contradiction with $\mathrm{wf}_{\mathrm{prom}}(i, gconf, V)$. □

The rest of the view-shift relations from Section 6.1.2 are proven in a similar way.

6.2.3. Semantics of Triples

The semantics of triples is given in terms of the safety predicate defined in Definition 6.7. Intuitively, $(gconf, V, r) \in \mathrm{safe}_n(\sigma, \mathcal{Q})$ means that a thread with thread state σ, view V, owning resources r, running on a machine with global configuration $gconf$ can safely run for n execution steps. Moreover, if the thread terminates during those n steps, the state it ends up in will be among the states in $\mathcal{Q}$.

Definition 6.7 (Safety) For a thread state σ, a set $\mathcal{Q} \subseteq \mathsf{GConf} \times \mathsf{View} \times \mathsf{WrPerm}$, and a natural number $n \in \mathbb{N}$, the *safety* predicate $\mathrm{safe}_n(\sigma, \mathcal{Q})$ is defined by recursion on n as follows.

$$\begin{aligned}
\mathrm{safe}_0(\sigma, \mathcal{Q}) &\coloneqq \mathsf{GConf} \times \mathsf{View} \times \mathsf{WrPerm} \\
\mathrm{safe}_{n+1}(\sigma, \mathcal{Q}) &\coloneqq \{(gconf, V, r) \mid (\sigma.s = \textbf{skip} \to (M, V, r) \in vs(\mathcal{Q})) \land \\
&\qquad (\forall r_F, \sigma', gconf', V', i. \\
&\qquad\quad gconf \in \lfloor r_F[i \mapsto r] \rfloor \land \langle\langle \sigma, V\rangle, gconf\rangle \Longrightarrow_i \langle\langle \sigma', V'\rangle, gconf'\rangle \\
&\qquad \to \exists r'.\, gconf' \in \lfloor r_F[i \mapsto r'] \rfloor \land (gconf', V', r') \in \mathrm{safe}_n(\sigma', \mathcal{Q}))\}
\end{aligned}$$

Note how we use the safety predicate to link the thread's view with the view used as a part of the assertion semantics: we require that when the thread is taking the step its view is the view coming from the assertion semantics.

Finally, we give the semantics of triples by saying that a triple $\{P\}\, s\, \{Q\}$ holds when the thread starting from a state which satisfies P can execute safely for arbitrarily many steps, and if it terminates, the concluding state satisfies Q.

Definition 6.8 (Triple semantics) A triple $\{P\}\, s\, \{Q\}$ holds if and only if for every thread state $\sigma = (-, s)$ and every $n \in \mathbb{N}$,

$$[\![P]\!] \subseteq \mathrm{safe}_n((\mu, s), [\![Q]\!])\,.$$

6.3. Soundness

In this section, we present a short overview of the soundness proof of Weasel. Our focus is not on the technical details of the proof, but on the main challenge in the proof: reasoning about promises.

To establish soundness of Weasel's proof rules, we have to prove that the safety predicate holds for an arbitrary number of steps, *including promise steps.* The trouble with reasoning about promise steps is that they can nondeterministically appear at any point of the execution. Therefore, we have to account for them in the soundness proof of every rule of our logic. To make this task manageable, we encapsulate reasoning about the promise steps in a theorem, thus enabling the proofs of soundness for proof rules to consider only the non-promise steps.

To do so, we utilize the certification runs for promises. Recall that whenever a thread makes a step, it has to be able to fulfill its promises without help from other threads (Section 5.3). Since there will be no interference by other threads, performing promise steps during certification is of no use (because promises can only be used by other threads). Therefore, we can assume that the certification runs are always promise-free.

Now that we have noted that certifications are promise-free, the key idea behind encapsulating the reasoning about promises is as follows. If we know that all executions of our program are safe for arbitrarily many non-promising steps, we can use this to conclude that they are safe for promising steps too. Here, we use the fact that certification runs are possible runs of the program, and the fact that certifications are promise-free.

Let us now see how to formalize the idea presented above. First, we need a way to state that executions are safe for non-promising steps. This is expressed by the non-promising safety predicate defined in Definition 6.9.

Definition 6.9 (Non-promising safety) For a thread state σ, a set $\mathcal{Q} \subseteq \mathsf{GConf} \times \mathsf{View} \times \mathsf{WrPerm}$, and a natural number $n \in \mathbb{N}$, the *non-promising safety* predicate $\mathrm{npsafe}_n(\sigma, \mathcal{Q})$ is defined by recursion on n as follows.

$$\begin{aligned}
\mathrm{npsafe}_0(\sigma, \mathcal{Q}) &\coloneqq \mathsf{GConf} \times \mathsf{View} \times \mathsf{WrPerm} \\
\mathrm{npsafe}_{n+1}(\sigma, \mathcal{Q}) &\coloneqq \{(gconf, V, r) \mid (\sigma.s = \mathbf{skip} \rightarrow (M, V, r) \in vs(\mathcal{Q})) \wedge \\
&\qquad (\forall r_F, \sigma', gconf', V', i. \\
&\qquad\quad gconf \in \lfloor r_F[i \mapsto r] \rfloor \wedge \langle\langle \sigma, V\rangle, gconf\rangle \xrightarrow{\mathrm{NP}}_i \langle\langle \sigma', V'\rangle, gconf'\rangle \wedge \\
&\qquad\quad \mathrm{wf}_{\mathrm{prom}}(i, gconf, V) \wedge \mathrm{wf}_{\mathrm{prom}}(i, gconf', V') \\
&\qquad \rightarrow \exists r'.\, gconf' \in \lfloor r_F[i \mapsto r'] \rfloor \wedge (gconf', V', r') \in \mathrm{npsafe}_n(\sigma', \mathcal{Q}))\},
\end{aligned}$$

where

$$\mathrm{wf}_{\mathrm{prom}}(i, gconf, V) \equiv \forall m \in gconf.P.\ m.\mathtt{tid} = i \rightarrow V(m.\mathtt{loc}) < m.\mathtt{time}\,.$$

Non-promising safety looks a lot like safety with two main differences. First, and most importantly, instead of machine transitions ($\Longrightarrow_i$) used by safety, non-promising safety uses non-promising thread reductions ($\xrightarrow{\mathrm{NP}}_i$). Second, there is an additional constraint on the promises being well-formed; we need that extra bit because $\xrightarrow{\mathrm{NP}}_i$ reductions do not certify promises, so we have to make sure we are not in a situation where the active thread can read from its own promises.

Notice how even though non-promising safety does not allow new promises to be generated, it still considers starting configurations which contain preexisting promises. This is very important for Theorem 6.2, since safety considers all possible starting configurations, including the ones with existing promises.

We can now state the theorem which allows us to completely short-circuit reasoning about promises. Theorem 6.2 is telling us that we need to consider only non-promising safety when establishing that a triple $\{P\}\, s\, \{Q\}$ holds; because if we establish that a program is non-promising safe for arbitrary many steps, then we can immediately conclude that it is also safe for arbitrarily many steps.

Theorem 6.2 (Non-promising safety implies safety) *Let σ be a thread state, $gconf \in \mathsf{GConf}$ a global configuration, $V \in \mathsf{View}$ a view, $r \in \mathsf{WrPerm}$ a write resource, and $\mathcal{Q} \subseteq \mathsf{GConf} \times \mathsf{View} \times \mathsf{WrPerm}$. Then*

$$\big(\forall n.\, (gconf, V, r) \in \mathrm{npsafe}_n(\sigma, \mathcal{Q})\big) \implies \big(\forall n.\, (gconf, V, r) \in \mathrm{safe}_n(\sigma, \mathcal{Q})\big)\,.$$

On a very high level, the proof of Theorem 6.2 involves decomposing the promise machine transitions, $\Longrightarrow_i$, used by the safety predicate, into the corresponding sequences

of thread transitions, $\longrightarrow_i$. Those thread transitions might involve some promise steps, but, since machine transitions have to be certifiable, there will always be a certifying run involving only non-promising thread steps.

Given the certifications (which contain no promising steps), we can then utilize the assumption of non-promising safety for arbitrary many steps to establish the safety of the machine transitions.

6.3.1. The Correctness Theorem

Since our language has only top-level parallel composition, we need a way to distribute initial resources to the various threads, and to collect all the resources once all the threads have finished. The correctness theorem gives us precisely that:

Theorem 6.3 (Correctness) If A is a finite set of locations and

1. $\circledast_{x \in A} \mathsf{W}^1(x, \{(0,0)\}) \Rrightarrow \circledast_{i \in \mathsf{Tid}} P_i$
2. $\{P_i\}\ s_i\ \{Q_i\}$ for all i
3. $\langle \lambda i.\, \langle (\mu_i, s_i), \bot \rangle, \langle M^0, \emptyset \rangle \rangle \Longrightarrow^* \langle \mathcal{TS}, \mathit{gconf} \rangle$ and $\mathcal{TS}(i).\sigma = \textsf{skip}$ for all i
4. $\vdash \circledast_{i \in \mathsf{Tid}} Q_i \Rrightarrow Q$
5. $\mathit{FRV}(Q_i) \cap \mathit{FRV}(Q_j) = \emptyset$ for all distinct $i, j \in \mathsf{Tid}$

then there exists $r\colon \mathsf{Tid} \to \mathsf{WrPerm}$ such that

$$(\mathit{gconf}, \mathbf{V}, \bigoplus_{i \in \mathsf{Tid}} r) \in [\![Q]\!] \qquad \text{and} \qquad \mathit{gconf} \in \lfloor r \rfloor\,,$$

where

$$\mathbf{V}(x) = \max\left(\{\mathcal{TS}(i).V(x) \mid i \in \mathsf{Tid}\}\right)\,,$$

and $\mathit{FRV}(P)$ denotes the set of free register variables in P.

Although the statement of the theorem looks very technical, it is simple in its essence. The theorem states if we assume nothing about the program beyond what the initial state tells us (all locations are initialized to 0), and verify all the threads separately, we can then collect the postconditions of all the threads in a *-conjunction, and know that the final state of the machine satisfies the *-conjunction of all the threads' postconditions.

The final view, $\mathbf{V}$, is not a particular view of any of the threads, but the per-location-maximal view taken over all of the thread views at the end of the execution. Intuitively, this is a sensible notion of what "the final view of the machine" would be, as it accounts for all the messages in the memory (there cannot be any outstanding promises at the end of an execution). Formally, setting $\mathbf{V}$ to be the per-location-maximal view across all the threads ensures that the observations and the write permissions coming from different threads will all be satisfied on the same view. (Note how Definition 6.4 makes sure that observations and write permissions cannot be invalidated if the view gets increased.)

The proof of the theorem is also rather straightforward. The safety definition ensures that the resources owned by the threads remain composable as the machine proceeds to take steps, and it ensures that the configuration, thread's final view, and the resources owned by each thread satisfy the thread's postcondition. Therefore, combining all those

resources, and taking the per-location-maximal view across the threads, gives us a structure satisfying the *-conjunction of all the individual thread's postconditions.

6.4. An Overview of Full SLR

After looking at the weasel fragment of SLR (Svendsen et al., 2018), let us briefly discuss the features of the full SLR logic, and the additional challenges faced in the soundness proof of SLR. The soundness proof of SLR (including the full technical details of the proof of Theorem 6.2) was primarily developed by Kasper Svendsen, and as such is largely beyond the scope of this thesis. However, for the sake of completeness of the discussion this brief overview is included.

Besides reasoning about relaxed accesses, SLR supports reasoning about release-acquire synchronization and ownership transfer between threads. To enable that kind of reasoning, SLR is equipped with the $\mathsf{Rel}(x, \phi)$ and $\mathsf{Acq}(x, \phi)$ assertions which function analogously to the corresponding FSL assertions explained in Section 3.1.3. As a result of including resource transfer in the logic, semantic structures discussed in Section 6.2 become significantly more intricate:

- Abstract resources for the release and acquire permissions need to hold syntactic assertions, similar to what was described in Section 3.2.1.
- Resources are no longer owned by threads only, but by messages too. Messages written by release writes hold onto resources which acquire writes are supposed to pick up.
- The notion of erasure becomes significantly more intricate, as it has to account for the possibility of resource transfer when stating which messages could have been written by which threads. This is particularly challenging for the messages belonging to the promised part of the memory.
- The definition of non-promising safety requires additional conditions to ensure that all the existing promises properly respect the given resources.

Still, the soundness proof revolves around the main idea presented in Section 6.3: use the non-promising safety predicate to encapsulate the reasoning about promises in a single theorem.

6.4.1. Comparison with FSL

When comparing SLR to FSL, we can immediately see that SLR is a significantly less expressive logic when it comes down to reasoning about synchronization patterns:

- SLR has no support for memory fences; all the synchronization has to happen via direct release write $\rightarrow$ acquire read communication.
- SLR has no support for atomic updates; only memory accesses available are reads and writes.

On the other hand, SLR is sound for a weaker model than FSL, and can reason about coherence properties of relaxed accesses.

Per-Execution vs. Multi-Execution Models

Looking at the way the soundness proofs of FSL and SLR work, we see that moving away from per-execution models into the territory of multi-execution models comes with a heavy burden of reasoning about multiple possible executions at the same time. The soundness proof we saw in Section 6.3 shoulders that burden by encapsulating the reasoning about promises and their certifications (which is the way the promising semantics exposes the alternate executions) into a single theorem. It is not clear how flexible that kind of approach is and how far can it be pushed. In the case of SLR, the semantics and the soundness proof had the promise encapsulation trick in mind from the beginning. A question that should be explored in the future is *how limiting is the encapsulation trick to the design and expressiveness of the logic?*

Execution Graphs vs. Operational Semantics

It is also interesting to note that the SLR soundness proof feels much more brittle than the FSL soundness proof. By that we mean envisioning the alterations of the FSL proof if we want to support some additional features comes easier than a similar exercise applied to SLR, even though SLR is arguably a simpler logic. A possible culprit here is the difference between the styles of underlying execution semantics those two logics are considering. We are not talking about per-execution vs. multi-execution difference, but about the difference between declarative and operational semantics.

FSL had the execution graph structure available, and used it to express the resource transfer in a rather straightforward manner. SLR has the operational semantics of the promise machine with its pool of messages and no explicit tracking of which load reads from which store, leading to a somewhat cumbersome technique of attaching resources to messages and accounting of who can get those resources.

Moreover, execution graphs are very good at avoiding "duplicating" explanations for a single observable program behavior (see Section 2.4.1 for an illustration of this property), while operational semantics often produce multiple runs which give rise to the same observable behavior (promising semantics is especially prolific in this regard, with the abundance of possible ways promises can happen). In a way, this "wastefulness" on behalf of operational semantics is giving the soundness proof more work, as it has to account for all those potential executions even though they lead to the same observable behavior.

This brief analysis suggests that—at least from the perspective of program logic design—execution-graph-based semantics might be a better fit to express weak memory models than operational semantics. A way to test this claim would be to establish soundness of a SLR-like logic for an execution-graph-based multi-execution semantics, and compare the proof structure and the difficulties faced between the soundness proof of SLR and the soundness proof of that new logic. This is a topic of ongoing research efforts, with the target model being the WEAKESTMO model of Chakraborty and Vafeiadis (2019).

Part III.

Related Work and Discussion

7. Related Work

This chapter discusses work related to verification of concurrent programs in the weak memory context. We divide the discussion in two parts: an overview of program logics and other deductive verification approaches to weak memory; and a less detailed look at other topics related to program verification under weak memory.

7.1. Program Logics

7.1.1. Logics for Language-level Memory Models

Relaxed Separation Logic

This work builds on top of relaxed separation logic (RSL) (Vafeiadis and Narayan, 2013), a logic whose primary focus is reasoning about the release-acquire fragment of the C11 memory model. There are two versions of RSL: a weak one that is sound with respect to the original C/C++11 memory model, which features out-of-thin-air reads, and a stronger one that is sound with respect to the model presented in Chapter 2.

The weak version of RSL cannot reason about relaxed writes at all. It does not constrain the value returned by a relaxed read, as not much else can be done in the presence of out-of-thin-air behaviors.

The stronger version relies on the stronger guarantees provided by the model that prevents load buffering. It allows users to attach an invariant to each location constraining which values can be written by relaxed writes, and providing very basic guarantees on the values seen by relaxed reads.

FSL significantly expands RSL by (1) allowing relaxed accesses to be used for resource transfer in coordination with memory fences; (2) making the rules for compare-and-swap (CAS) instructions more powerful by exploiting the release-sequence synchronization mechanism; and (3) introducing the capability to reason about the ghost state. Additionally, FSL's soundness proof required novel proof techniques, compared to the ones used to establish the soundness of RSL.

RSL makes no attempt to reason about coherence properties of relaxed accesses (nor does FSL, for that matter). Weasel fills that gap by considering coherence, and in fact doing so under a much weaker model than the one considered by RSL and FSL.

Other Logics and Deductive Frameworks

Another logic targeting the release-acquire fragment of C11 is GPS (Turon et al., 2014). Compared to RSL, GPS is significantly more expressive, reasoning about ownership

transfer using protocols and escrows. Recently, He et al. (2020) extended GPS with support for ownership transfer via relaxed accesses, paralleling the development of FSL on top of RSL.

Kaiser et al. (2017) used the Iris framework (Jung et al., 2015, 2016; Krebbers et al., 2017) to develop iRSL and iGPS, higher-order versions of RSL and GPS. By virtue of being implemented in Iris, iRSL and iGPS inherit Iris' advanced features, such as higher-order impredicative quantification. In order to use Iris, it was necessary to recast the release-acquire fragment of the C11 memory model in an operational semantics in the style of Lahav et al. (2016).

Dang et al. (2020) merge FSL features into iGPS (fences, modalities, modality-agnostic ghost state), and enrich it with an additional kind of ghost state, aware of the modalities. The modality-aware ghost state is used to reason about memory deallocation in the effort of verification of several Rust libraries, including a more advanced version of the ARC algorithm.

Apart from various separation logics discussed so far, there are two Owicki-Gries-like logics: OGRA by Lahav and Vafeiadis (2015) for the release-acquire part of the C11 model, and the logic of Dalvandi et al. (2020) which includes support for relaxed accesses, but not for memory fences. These approaches are by construction non-compositional, as they require the user to consider stability conditions derived from inspecting the entire program.

Beyond logics, the deductive framework of Alglave and Cousot (2017) can reason about models expressible using the `cat` language (Alglave et al., 2014, 2016). The framework is extremely flexible regarding the models it can consider, as long as the desired model is a per-execution model, since `cat` cannot express multi-execution models such as the promising model described in Chapter 5. The framework is also non-compositional; requiring verification of complete programs.

All the logics and frameworks mentioned above assume some kind of a per-execution memory model dealing with the out-of-thin-air problem by disallowing load buffering behaviors altogether. For multi-execution models—apart from Weasel/SLR—there are only invariant-based logics (Kang et al., 2017; Jeffrey and Riely, 2019) which are strong enough to establish the absence of the most obvious out-of-thin-air behaviors[1] in the underlying memory models, but very little beyond that. They are by design unable to reason about examples such as (RNG) from Section 6.1.4, where having a bound on the set of values written to a location is not enough, let alone reasoning about functional correctness of a program.

Automation of FSL

FSL has been automated by Summers and Müller (2020) using Viper (Müller et al., 2016). The automation encodes all of FSL's features with only minimal restrictions, most of which are not of practical concern.

[1]Soundness of invariant-based logics demonstrates that the locations in programs without arithmetic will never hold values not mentioned by the program.

7.1.2. Logics for Hardware-level Memory Models

Multiple logics have been developed for reasoning under the x86-TSO memory model, such as iCAP-TSO (Sieczkowski et al., 2015), the rely-guarantee proof system of Ridge (2010), and the program logic of Wehrman and Berdine (2011).

The lace logic of Bornat et al. (2015) targets hardware-style memory models which do allow load-buffering (e.g., Power). As it is dealing with hardware-level per-execution models, lace can prove the absence of load buffering in LB+FAKEDEP example from Fig. 1.2.

7.2. Other Verification Topics

Robustness Theorems

Robustness theorems allow us to reduce reasoning about programs under a weak memory model M_w to reasoning under a stronger memory model M_s. This is done by establishing sufficient conditions for which the observable behaviors of the program are indistinguishable under M_w and M_s.

Most of the robustness results reduce weak memory models to sequential consistency. The memory models considered are often hardware-level models (Owens, 2010; Bouajjani et al., 2011; Derevenetc and Meyer, 2014), but recently results exploring robustness for fragments of the C11 memory model have appeared (Lahav and Margalit, 2019; Margalit and Lahav, 2021).

For the promising semantics, Kang et al. (2017, §5.4) have established three theorems that reduce the reasoning to sequential consistency, to release-acquire consistency, and to the promise-free fragment of the promising semantics respectively, depending on how strict conditions one is willing to impose on programs.

Model Checking

Model checking techniques explore all possible observable behaviors of a program by enumerating all executions (and striving to avoid repeated checking of executions which result in previously checked behaviors).

Most model checking work has been done for per-execution memory models, be it hardware- or language-level models which completely forbid load-buffering (Demsky and Lam, 2015; Abdulla et al., 2018; Kokologiannakis et al., 2018, 2019), or hardware-level models which allow LB, but forbid LB+FAKEDEP (see Fig. 1.2) (Alglave et al., 2013; Abdulla et al., 2016; Kokologiannakis and Vafeiadis, 2020).

Hardly any model-checking work has been done for models which allow both LB and LB+FAKEDEP. CDSChecker (Norris and Demsky, 2013) works under a bounded version of the original C11 model allowing out-of-thin-air behaviors, which leads it down the path of exploring many unfeasible executions. Paviotti et al. (2020) propose a novel model for avoiding out-of-thin-air behaviors and develop a model checking technique for it.

Specification of Library Correctness

Under sequential consistency, the most well-established criterion for correctness of concurrent libraries is linearizability. Unfortunately, linearizability is no longer compositional under weak memory, and is thus not an adequate correctness criterion.

Several proposals have been made to address this issue, either by modifying the notion of linearizability in order to recover compositionality (Doherty et al., 2018), or by developing novel correctness criteria for weak memory (Batty et al., 2013; Raad et al., 2019). All of these approaches concern per-execution models which forbid load-buffering in order to avoid out-of-thin-air behaviors.

As of now, none of the approaches have established themselves as the definite correctness criterion for concurrent libraries under weak memory.

8. Lessons Learned and Future Directions

8.1. Looking Back

In this thesis, we have explored two program logics dealing with two very different weak memory models: FSL for reasoning about programs under a strengthening of the C11 memory model, and Weasel (a fragment of SLR), operating under the promising semantics.

From the beginning, the design principle behind the logics was to prefer clean and transparent inference rules over potentially more expressive, but more convoluted, rules. This approach was chosen for two main reasons:

1. We wanted to isolate core reasoning principles valid under weak memory, and to look for general proof approaches when it comes to reasoning about soundness under weak memory models. In a way, we wanted our logics to function as exploratory tools and sanity-checks of the underlying memory models, and if successful, to serve as witnesses to the quality of the models.

2. As the formulations of weak memory models are very complex, being able to understand behaviors of even simple programs from the standpoint of a weak memory model can be a challenging task. Logics with simple inference rules can provide a much needed abstraction layer over the underlying models, and tell a more intuitive story which can then be very useful for programmers (even if they do not go through the effort of formal verification).

With those two points in mind, let us evaluate the success of the two presented projects.

FSL – A Resounding Success

Instead of the execution graphs, FSL's inference rules tell a simple story of resources being transferred between threads via reads and writes, guarded by fences where appropriate. The intricate synchronization patterns of the C11 model are summarized by the interplay of the $\triangle$ and $\triangledown$ modalities and fences. Despite their simplicity, the rules of FSL proved to be expressive enough to reason about more than just "toy" examples, as evidenced by the verification of ARC.

Additional confidence in FSL's way of talking about C11 synchronization patterns is given the follow-up logics (He et al., 2020; Dang et al., 2020), capable of reasoning about the interplay of memory fences and relaxed accesses, picking up on FSL's modalities and rules governing them.

It is also worth noting that only the fragment of the C11 model considered by FSL remained unchanged since the model appeared as a part of the C and C++ standards (ISO/IEC, 2011a,b). Under the influence of research (e.g. Lahav et al., 2017; Vafeiadis et al., 2015) uncovering inconsistencies and unintended consequences of the original definitions, the subsequent standards' revisions (ISO/IEC, 2014, 2018, 2017, 2020) include several modifications to the original model, most notably:

- restricting the release-sequences to the definition FSL relies on (see the remark on page 16); and
- significantly restructuring the guarantees provided by the sequentially consistent atomics.

The current standards even forbid accessing a location using both atomic and non-atomic accesses, which is a restriction FSL imposes on the programs whose validity it can prove.

Even though the rest of the model has been in flux, it is safe to say that the fragment covered by FSL will not see any changes in the future (as long as the standardizing committee keeps the per-execution style of defining the model). Among other things, having logical foundations and principles behind the model explored and expressed in a clear manner (such as having an expressive-enough program logic) serves as strong evidence that the model is sensible and is providing us with enough guarantees to enable structured reasoning about behavior of our programs.

In summary, we can confidently say that FSL succeeds in demonstrating the reliability and usefulness of the underlying model. Additionally, it can, at least to some extent, serve as a decent abstraction layer over the details of the model.

Weasel/SLR – A Significant Step Forward

Weasel (and especially SLR as a whole) definitely demonstrates that multi-execution models (and the promising semantics, in particular) provide enough structure to support expressive deductive reasoning. However, when compared with FSL, we see that several key features are lacking, such as the support for memory fences and atomic updates.

On the front of abstracting and encapsulating the model, SLR does well when it comes to the release-acquire ownership transfer (following in the footsteps of RSL and FSL), but the Weasel fragment and its capabilities for reasoning about coherence do not fully encapsulate the underlying model.

Weasel enables the user to reason about coherence properties, and is the only logic so far to do that for a language-level model, but does so by exposing the timestamp mechanism of the underlying machine, and providing a tidy way of keeping track of them. Arguably, this is probably the best that can be done, as coherence is a very low-level property, and a very non-local property. It is defined in terms of threads agreeing on the order of stores to a single location, and explicitly recording the information about that order seems necessary to be able to reason about it.

The most useful lessons obtained by the development of SLR/Weasel come from the difficulties faced in constructing soundness proofs under multi-execution memory models (see Section 6.4.1 for a more detailed discussion). Although it is not completely clear which difficulties should be ascribed to the multi-execution models as a whole, and which

stem from the particular model used by Weasel/SLR, the difficulties we experienced fit into a larger pattern.

Looking at the landscape of formal verification under weak memory (see Chapter 7), we can see that in each area a lot of work has been done under per-execution models, but when it comes to multi-execution models[1], the landscape suddenly turns rather desolate. The stark contrast between availability of work about per-execution and about multi-execution models indicates that there is some kind of a fundamental, and yet unresolved, difficulty when it comes to reasoning about multi-execution models. In light of this, Weasel/SLR should be seen as a push, from the area of program logics, towards uncovering the difficulties preventing us from approaching multi-execution models in a more effective way.

8.2. Looking Forward

When considering future research directions, we should be aware of the difference in the levels of understanding when it comes to per-execution and multi-execution weak memory models, and the research goals and ambitions should be adjusted accordingly.

Per-Execution Memory Models

From the work presented in this thesis, and the general state of the area discussed in Chapter 7, it is clear that the per-execution models are rather well understood. This is particularly true for memory models which forbid load-buffering behaviors. The maturity of theoretical understanding of the per-execution models disallowing load-buffering indicates that these kinds of models should be the focus when it comes to research aiming to have not only theoretical, but potential practical impact too. Granted, the practicality of disallowing load-buffering is still a matter of debate, but those for whom having verified concurrent software is a priority might want to consider taking a potential performance hit in exchange for a promise of better tool support. For this reason, building verified compilers, expressive and automated logics, practical model checkers, and testing tools over the models which disallow load-buffering in mind should be an important research avenue in the near future.

Concerns about potential practical drawbacks of models forbidding load buffering (see Section 2.5 for a more detailed discussion) should be explored further, e.g., by measurements such as the ones by Ou and Demsky (2018). Coming up with additional ways to evaluate and measure the cost incurred by restricting load buffering would help establish more confidence in practicality of per-execution language-level models, even when it comes to the hardware architectures which allow some forms of load buffering.

Some theoretical foundations are still missing, though. The two most prominent ones are the lack of a well-understood criterion for library correctness under weak memory, and the relatively shaky understanding of the sequentially-consistent atomic access mode.

[1] Or any kind of models permitting both LB and LB+FAKEDEP, while forbidding LB+DEP.

The situation with library correctness criteria was already mentioned at the end of Section 7.2. More work needs to be done on understanding what kind of correctness criteria fit weak memory, and to see which approaches lend themselves better to building tool support for them.

When it comes to sequentially-consistent (sc) atomic access mode, the root of the problem is in the lack of clear intuitive understanding behind the guarantees provided by sc accesses. Most theoretical work so far shies away from dealing with sc accesses, and when it does approach the issue (e.g., Lahav et al., 2017), the discussion remains on the level of relations in the execution graphs, and looking at particular examples; very little to no intuition in terms other than the execution graphs is provided.

The difficulty arises from the main motivation behind sc accesses being that a program which uses only sc accesses in its racy parts should behave as specified by the sequential consistency semantics. However, once other access modes (**rlx**, **rel**, **acq**) are thrown into the mix, it becomes exceedingly unclear what kind of guarantees should we expect.

An interesting research direction would be to look for ways to express the effect of sc accesses in terms of resource manipulation, providing us with another kind of story to tell about sc access mode. This could maybe be done via a program logic, similarly to how FSL provides us with a higher-level story about the synchronization mechanisms regarding the other access modes.

Multi-Execution Memory Models

Multi-execution models arose as a way of solving the out-of-thin air problem on the language-level, without completely forbidding load buffering behaviors. The main approaches to the design of these models are either based on event structures (Jeffrey and Riely, 2019; Pichon-Pharabod and Sewell, 2016; Chakraborty and Vafeiadis, 2019) or operational (Kang et al., 2017), with novel approaches still appearing (Paviotti et al., 2020). The diversity of the proposed models shows that there is currently no consensus on which approach is most suitable for defining language-level models which allow load buffering but forbid out-of-thin-air behaviors.

In this area, the research should be focused on gaining better understanding of the various styles of models, their advantages and drawbacks. Developing simple logics of similar style to Weasel/SLR can point out where certain models shine, and where dealing with them gets cumbersome.

Back in Section 6.4.1, we discussed how the different structure of soundness proofs of FSL and Weasel/SLR might indicate that the event-structure-based models might be a better fit for development of this style of program logics. Here, we also want to bring to attention the line of work on developing iRSL, iGPS (Kaiser et al., 2017), and iRC11 (Dang et al., 2020), showing how operational models can be useful to exploit the power of the Iris framework.

The model assumed by iRSL, iGPS, and iRC11 is an operational version of the model presented in Chapter 2, very similar to the promise machine of Chapter 5, but without the promise steps. It would be interesting to see if the promise encapsulation trick employed by Weasel/SLR could somehow be extended to those logics.

Bibliography

P. A. Abdulla, M. F. Atig, B. Jonsson, and C. Leonardsson. Stateless model checking for POWER. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 134–156. Springer, 2016. doi: 10.1007/978-3-319-41540-6_8. URL https://doi.org/10.1007/978-3-319-41540-6_8.

P. A. Abdulla, M. F. Atig, B. Jonsson, and T. P. Ngo. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM Program. Lang.*, 2(OOPSLA):135:1–135:29, 2018. doi: 10.1145/3276505. URL https://doi.org/10.1145/3276505.

J. Alglave and P. Cousot. Ogre and pythia: an invariance proof method for weak consistency models. In G. Castagna and A. D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 3–18. ACM, 2017. URL http://dl.acm.org/citation.cfm?id=3009883.

J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In N. Sharygina and H. Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2013. doi: 10.1007/978-3-642-39799-8_9. URL https://doi.org/10.1007/978-3-642-39799-8_9.

J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014. doi: 10.1145/2627752. URL https://doi.org/10.1145/2627752.

J. Alglave, P. Cousot, and L. Maranget. Syntax and semantics of the weak consistency model specification language cat. *CoRR*, abs/1608.07531, 2016. URL http://arxiv.org/abs/1608.07531.

M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In T. Ball and M. Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 55–66. ACM, 2011. doi: 10.1145/1926385.1926394. URL https://doi.org/10.1145/1926385.1926394.

M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 235–248. ACM, 2013. doi: 10.1145/2429069.2429099. URL https://doi.org/10.1145/2429069.2429099.

M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The problem of programming language concurrency semantics. In J. Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 283–307. Springer, 2015. doi: 10.1007/978-3-662-46669-8_12. URL https://doi.org/10.1007/978-3-662-46669-8_12.

L. Birkedal, K. Støvring, and J. Thamsborg. The category-theoretic solution of recursive metric-space equations. *Theor. Comput. Sci.*, 411(47):4102–4122, 2010. doi: 10.1016/j.tcs.2010.07.010. URL https://doi.org/10.1016/j.tcs.2010.07.010.

R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 259–270. ACM, 2005. doi: 10.1145/1040305.1040327. URL https://doi.org/10.1145/1040305.1040327.

R. Bornat, J. Alglave, and M. J. Parkinson. New lace and arsenic: adventures in weak memory with a program logic. *CoRR*, abs/1512.01416, 2015. URL http://arxiv.org/abs/1512.01416.

A. Bouajjani, R. Meyer, and E. Möhlmann. Deciding robustness against total store ordering. In L. Aceto, M. Henzinger, and J. Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, volume 6756 of *Lecture Notes in Computer Science*, pages 428–440. Springer, 2011. doi: 10.1007/978-3-642-22012-8_34. URL https://doi.org/10.1007/978-3-642-22012-8_34.

J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003. doi: 10.1007/3-540-44898-5_4. URL https://doi.org/10.1007/3-540-44898-5_4.

C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 366–378. IEEE Computer Society, 2007. doi: 10.1109/LICS.2007.30. URL https://doi.org/10.1109/LICS.2007.30.

S. Chakraborty and V. Vafeiadis. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.*, 3(POPL):70:1–70:28, 2019. doi: 10.1145/3290383. URL https://doi.org/10.1145/3290383.

S. Dalvandi, S. Doherty, B. Dongol, and H. Wehrheim. Owicki-gries reasoning for C11 RAR. In R. Hirschfeld and T. Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 11:1–11:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi: 10.4230/LIPIcs.ECOOP.2020.11. URL https://doi.org/10.4230/LIPIcs.ECOOP.2020.11.

H. Dang, J. Jourdan, J. Kaiser, and D. Dreyer. Rustbelt meets relaxed memory. *Proc. ACM Program. Lang.*, 4(POPL):34:1–34:29, 2020. doi: 10.1145/3371102. URL https://doi.org/10.1145/3371102.

B. Demsky and P. Lam. Satcheck: Sat-directed stateless model checking for SC and TSO. In J. Aldrich and P. Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 20–36. ACM, 2015. doi: 10.1145/2814270.2814297. URL https://doi.org/10.1145/2814270.2814297.

E. Derevenetc and R. Meyer. Robustness against power is pspace-complete. In J. Esparza, P. Fraigniaud, T. Husfeldt, and E. Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 158–170. Springer, 2014. doi: 10.1007/978-3-662-43951-7_14. URL https://doi.org/10.1007/978-3-662-43951-7_14.

T. Dinsdale-Young, L. Birkedal, P. Gardner, M. J. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 287–300. ACM, 2013. doi: 10.1145/2429069.2429104. URL https://doi.org/10.1145/2429069.2429104.

S. Doherty, B. Dongol, H. Wehrheim, and J. Derrick. Making linearizability compositional for partially ordered executions. In C. A. Furia and K. Winter, editors, *Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings*, volume 11023 of *Lecture Notes in Computer Science*, pages 110–129. Springer, 2018. doi: 10.1007/978-3-319-98938-9_7. URL https://doi.org/10.1007/978-3-319-98938-9_7.

M. Doko and V. Vafeiadis. A program logic for C11 memory fences. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January*

17-19, 2016. Proceedings, volume 9583 of *Lecture Notes in Computer Science*, pages 413–430. Springer, 2016. doi: 10.1007/978-3-662-49122-5_20. URL https://doi.org/10.1007/978-3-662-49122-5_20.

M. Doko and V. Vafeiadis. Tackling real-life relaxed concurrency with FSL++. In H. Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 448–475. Springer, 2017. doi: 10.1007/978-3-662-54434-1_17. URL https://doi.org/10.1007/978-3-662-54434-1_17.

M. He, S. Qin, and Z. Xu. A program logic for reasoning about C11 programs with release-sequences. *IEEE Access*, 8:173874–173903, 2020. doi: 10.1109/ACCESS.2020.3024681. URL https://doi.org/10.1109/ACCESS.2020.3024681.

INRIA. The Coq proof assistant. http://coq.inria.fr/.

ISO/IEC. Programming language C++, standard revision 14882:2011, 2011a.

ISO/IEC. Programming language C, standard revision 9899:2011, 2011b.

ISO/IEC. Programming language C++, standard revision 14882:2014, 2014.

ISO/IEC. Programming language C++, standard revision 14882:2017, 2017.

ISO/IEC. Programming language C, standard revision 9899:2018, 2018.

ISO/IEC. Programming language C++, standard revision 14882:2020, 2020.

A. Jeffrey and J. Riely. On thin air reads: Towards an event structures model of relaxed memory. *Log. Methods Comput. Sci.*, 15(1), 2019. doi: 10.23638/LMCS-15(1:33)2019. URL https://doi.org/10.23638/LMCS-15(1:33)2019.

J. B. Jensen and L. Birkedal. Fictional separation logic. In H. Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 377–396. Springer, 2012. doi: 10.1007/978-3-642-28869-2_19. URL https://doi.org/10.1007/978-3-642-28869-2_19.

R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM, 2015. doi: 10.1145/2676726.2676980. URL https://doi.org/10.1145/2676726.2676980.

R. Jung, R. Krebbers, L. Birkedal, and D. Dreyer. Higher-order ghost state. In J. Garrigue, G. Keller, and E. Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 256–269. ACM, 2016. doi: 10.1145/2951913.2951943. URL https://doi.org/10.1145/2951913.2951943.

J. Kaiser, H. Dang, D. Dreyer, O. Lahav, and V. Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in iris. In P. Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 17:1–17:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi: 10.4230/LIPIcs.ECOOP.2017.17. URL https://doi.org/10.4230/LIPIcs.ECOOP.2017.17.

J. Kang, C. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. In G. Castagna and A. D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 175–189. ACM, 2017. URL http://dl.acm.org/citation.cfm?id=3009850.

M. Kokologiannakis and V. Vafeiadis. HMC: model checking for hardware memory models. In J. R. Larus, L. Ceze, and K. Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 1157–1171. ACM, 2020. doi: 10.1145/3373376.3378480. URL https://doi.org/10.1145/3373376.3378480.

M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.*, 2(POPL):17:1–17:32, 2018. doi: 10.1145/3158105. URL https://doi.org/10.1145/3158105.

M. Kokologiannakis, A. Raad, and V. Vafeiadis. Model checking for weakly consistent libraries. In K. S. McKinley and K. Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 96–110. ACM, 2019. doi: 10.1145/3314221.3314609. URL https://doi.org/10.1145/3314221.3314609.

R. Krebbers, R. Jung, A. Bizjak, J. Jourdan, D. Dreyer, and L. Birkedal. The essence of higher-order concurrent separation logic. In H. Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 696–723. Springer, 2017. doi: 10.1007/978-3-662-54434-1_26. URL https://doi.org/10.1007/978-3-662-54434-1_26.

O. Lahav and R. Margalit. Robustness against release/acquire semantics. In K. S. McKinley and K. Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA,*

June 22-26, 2019, pages 126–141. ACM, 2019. doi: 10.1145/3314221.3314604. URL https://doi.org/10.1145/3314221.3314604.

O. Lahav and V. Vafeiadis. Owicki-gries reasoning for weak memory models. In M. M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 2015. doi: 10.1007/978-3-662-47666-6_25. URL https://doi.org/10.1007/978-3-662-47666-6_25.

O. Lahav, N. Giannarakis, and V. Vafeiadis. Taming release-acquire consistency. In R. Bodík and R. Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 649–662. ACM, 2016. doi: 10.1145/2837614.2837643. URL https://doi.org/10.1145/2837614.2837643.

O. Lahav, V. Vafeiadis, J. Kang, C. Hur, and D. Dreyer. Repairing sequential consistency in C/C++11. In A. Cohen and M. T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 618–632. ACM, 2017. doi: 10.1145/3062341.3062352. URL https://doi.org/10.1145/3062341.3062352.

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979. doi: 10.1109/TC.1979.1675439. URL https://doi.org/10.1109/TC.1979.1675439.

R. Ley-Wild and A. Nanevski. Subjective auxiliary state for coarse-grained concurrency. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 561–574. ACM, 2013. doi: 10.1145/2429069.2429134. URL https://doi.org/10.1145/2429069.2429134.

S. Mador-Haim, R. Alur, and M. M. K. Martin. Generating litmus tests for contrasting memory consistency models. In T. Touili, B. Cook, and P. B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 273–287. Springer, 2010. doi: 10.1007/978-3-642-14295-6_26. URL https://doi.org/10.1007/978-3-642-14295-6_26.

S. Mador-Haim, R. Alur, and M. M. K. Martin. Litmus tests for comparing memory consistency models: how long do they need to be? In L. Stok, N. D. Dutt, and S. Hassoun, editors, *Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10, 2011*, pages 504–509. ACM, 2011. doi: 10.1145/2024724.2024842. URL https://doi.org/10.1145/2024724.2024842.

J. Manson, W. Pugh, and S. V. Adve. The java memory model. In J. Palsberg and M. Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005, pages 378–391. ACM, 2005. doi: 10.1145/1040305.1040336. URL https://doi.org/10.1145/1040305.1040336.

L. Maranget, S. Sarkar, and P. Sewell. A tutorial introduction to the ARM and POWER relaxed memory models. http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf, 2012.

R. Margalit and O. Lahav. Verifying observational robustness against a c11-style memory model. *Proc. ACM Program. Lang.*, 5(POPL):1–33, 2021. doi: 10.1145/3434285. URL https://doi.org/10.1145/3434285.

P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016. doi: 10.1007/978-3-662-49122-5_2. URL https://doi.org/10.1007/978-3-662-49122-5_2.

B. Norris and B. Demsky. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *OOPSLA 2013*, pages 131–150, New York, NY, USA, 2013. ACM.

P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375 (1-3):271–307, 2007. doi: 10.1016/j.tcs.2006.12.035. URL https://doi.org/10.1016/j.tcs.2006.12.035.

P. Ou and B. Demsky. Towards understanding the costs of avoiding out-of-thin-air results. *Proc. ACM Program. Lang.*, 2(OOPSLA):136:1–136:29, 2018. doi: 10.1145/3276506. URL https://doi.org/10.1145/3276506.

S. Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In T. D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 478–503. Springer, 2010. doi: 10.1007/978-3-642-14107-2_23. URL https://doi.org/10.1007/978-3-642-14107-2_23.

M. Paviotti, S. Cooksey, A. Paradis, D. Wright, S. Owens, and M. Batty. Modular relaxed dependencies in weak memory concurrency. In P. Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *Lecture Notes in Computer Science*, pages 599–625. Springer, 2020. doi: 10.1007/978-3-030-44914-8_22. URL https://doi.org/10.1007/978-3-030-44914-8_22.

J. Pichon-Pharabod and P. Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *POPL 2016*, pages 622–633.

ACM, 2016. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837616. URL http://doi.acm.org/10.1145/2837614.2837616.

A. Raad, M. Doko, L. Rožić, O. Lahav, and V. Vafeiadis. On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models. *Proc. ACM Program. Lang.*, 3(POPL):68:1–68:31, 2019. doi: 10.1145/3290381. URL https://doi.org/10.1145/3290381.

T. Ridge. A rely-guarantee proof system for x86-tso. In G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings*, volume 6217 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 2010. doi: 10.1007/978-3-642-15057-9_4. URL https://doi.org/10.1007/978-3-642-15057-9_4.

Rust Team. Atomic reference counter (ARC) documentation. https://doc.rust-lang.org/std/sync/struct.Arc.html, 2015a.

Rust Team. The Rust programming language. https://www.rust-lang.org/, 2015b.

F. Sieczkowski, K. Svendsen, L. Birkedal, and J. Pichon-Pharabod. A separation logic for fictional sequential consistency. In J. Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 736–761. Springer, 2015. doi: 10.1007/978-3-662-46669-8_30. URL https://doi.org/10.1007/978-3-662-46669-8_30.

A. J. Summers and P. Müller. Automating deductive verification for weak-memory programs (extended version). *Int. J. Softw. Tools Technol. Transf.*, 22(6):709–728, 2020. doi: 10.1007/s10009-020-00559-y. URL https://doi.org/10.1007/s10009-020-00559-y.

K. Svendsen, J. Pichon-Pharabod, M. Doko, O. Lahav, and V. Vafeiadis. A separation logic for a promising semantics. In A. Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 357–384. Springer, 2018. doi: 10.1007/978-3-319-89884-1_13. URL https://doi.org/10.1007/978-3-319-89884-1_13.

A. Turon, V. Vafeiadis, and D. Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In A. P. Black and T. D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 691–707. ACM, 2014. doi: 10.1145/2660193.2660243. URL https://doi.org/10.1145/2660193.2660243.

V. Vafeiadis and C. Narayan. Relaxed separation logic: a program logic for C11 concurrency. In A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 867–884. ACM, 2013. doi: 10.1145/2509136.2509532. URL https://doi.org/10.1145/2509136.2509532.

V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Z. Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 209–220. ACM, 2015. doi: 10.1145/2676726.2676995. URL https://doi.org/10.1145/2676726.2676995.

I. Wehrman and J. Berdine. A proposal for weak-memory local reasoning. In *LOLA*, 2011. URL https://www.microsoft.com/en-us/research/publication/a-proposal-for-weak-memory-local-reasoning/.

Appendix

A. Coq Implementation of FSL

FSL has been formally verified using the Coq proof assistant. The details can be seen on the project's web page, http://plv.mpi-sws.org/fsl/.

The C11 Model

The C11 model in the Coq development follows the presetation of the model outlined in (Vafeiadis and Narayan, 2013, §3), appropriately extended with memory fences. The presentation in Chapter 2 follows the more modern approach of Lahav et al. (2017).

Semantically, considering the features FSL relies upon, there is no difference between the model conisdered in the Coq development and the one presented in Chapter 2. Translation between the two presentations of the model can be found in (Lahav et al., 2017, §3.4, Proposition 1).

FSL Assertion Semantics

The assertion semantics in the Coq development follows the semantics given in (Doko and Vafeiadis, 2016, §5). Instead of having three separate resource heaps to represent "usable" resources and the resources protected by the modalities (as in Section 3.2), there is a single resource heap, and each resource on the heap is marked by a set of labels ($\circ$, $\scriptstyle\triangle$, or $\scriptstyle\triangledown$), depending on which modalities apply to it.

The two presentations are almost equivalent. The only difference is that in the original presentation, acquire permissions were modality-agnostic, i.e., we had

$$\mathsf{Acq}(x, \mathcal{Q}) \iff \triangle\mathsf{Acq}(x, \mathcal{Q}) \iff \triangle\mathsf{Acq}(x, \mathcal{Q})\,.$$

This equivalence does not cause any soundness issues, since acquire permission $\mathsf{Acq}(x, \mathcal{Q})$ cannot ever be used without a corresponding initialization assertion $\mathsf{Init}(x)$, which are affected by modalities.

The version of assertion semantics from Section 3.2 is generally cleaner and easier to work with. In particular, the "resource tracking" technique used in the proofs of Theorems 3.3 to 3.5 can be expressed much easier when the semantics is given using three separate resource heaps.

Definitions and Theorem Names in the Coq Development

Table A.1 lists the corresponding names of important theorems and definitions mentioned in Chapters 3 and 4.

Theorem/Definition	Coq development	
	Name	File
Definition 3.4	`Asim`	`fslassn.v`
Definition 3.10	`assn_sem`	`fslassnsem.v`
Proposition 3.1	`sim_assn_sem`	`fslassnlemmas.v`
Proposition 3.5	`MFPassn_normalizable`	`normalizability.v`
Definition 3.14	`hmap_valid` `hmap_valids`	`fslmodel.v`
Definition 3.16	`hist_closed`	`fslmodel.v`
Definition 3.18	`safe`	`fslmodel.v`
Definition 3.19	`triple`	`fslmodel.v`
Theorem 3.1	`independent_two`	`ihc.v`
Theorem 3.2	`independent_heap_compat`	`ihc.v`
Theorem 3.3	`valid_implies_mem_safe`	`memsafe.v`
Theorem 3.4	`valid_implies_initialized_reads`	`initread.v`
Theorem 3.5	`valid_implies_race_free`	`drf.v`
Theorem 3.6	`adequacy`	`fsl.v`
Theorem 3.7	`rule_store_na1` `rule_store_na2`	`fsl.v`
Theorem 3.8	`rule_fence_acq`	`fsl.v`
Theorem 3.9	`rule_fence_rel`	`fsl.v`
Theorem 3.10	`rule_load_na`	`fsl.v`
Lemma 3.3	`wellformed`	`fslhmapa.v`
Theorem 3.11	`rule_load_acq` `rule_load_rlx`	`load.v`

Theorem 3.12	`rule_cas_ar`
	`rule_cas_rel`
	`rule_cas_acq`
	`rule_cas_rlx`
	`rule_cas_shortcut`
		`rmw.v`

Figure 3.5	`fetch_and_modify` `rule_fetch_and_modify`	`fsl_addons.v`
Lemma 4.1	`TokenMonoid`	`arc_ghosts.v`
Definition 4.1	`QQ`	`arc.v`
Definition 4.2	`ARC`	`arc.v`

Theorem 4.1	`ARC_new_spec`
	`ARC_load_spec`
	`ARC_clone_spec`
	`ARC_drop_spec`
		`arc.v`

Table A.1.: List of names in the Coq development. The names and files are referring to the FSL's Coq development available at `http://plv.mpi-sws.org/fsl/ARC/fsl-2.0.2.tbz2`.

Curriculum Vitae

Research Interests

Programming languages and verification, formal methods in computer science, weak memory concurrency.

Education and employment

2021 – Assistant Professor, Heriot-Watt University, Edinburgh, United Kingdom

2013 – 2021 Doctoral student, Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

2007 – 2013 Teaching assistant, Department of Mathematics, University of Zagreb, Croatia

Responsible for the following courses: Introduction to Programming, Data Structures and Algorithms, Computer Networks, Databases, Computability Theory, Set Theory. Enrolled in a doctoral program in mathematics.

2001 – 2006 Dipl. Ing. (MS equivalent) in Mathematics (profile: Computer Science), Department of Mathematics, University of Zagreb, Croatia.