Deductive Verification of Real-World C++ Weak-Memory Programs

Pascal Wiesmann
Supervised by Dr. Alexander J. Summers

June 2, 2019

Abstract

In this work we provide a high-level and intuitive logic for the verification of weak memory C++ programs. The logic is based on FSL++[4] and improves it in two crucial aspects. First, it significantly simplifies the task of writing the specification. Second, it is more suitable for automating verification. We also provide a tool that verifies weak-memory C++ programs using our new logic. The program and the specifications are encoded into Viper in such a way that they can then be automatically verified.

Contents

1 Introduction **4**

2 Background **6**
2.1 Weak Memory 6
2.2 Data races 7
2.3 Atomics and Synchronization 7
2.4 Relaxed Separation Logic 8
2.5 Fenced Separation Logic (FSL) 9
2.6 FSL++ 10
2.7 Entity Fractional Counting 10

3 Token-based Reasoning for Weak Memory Programs **11**
3.1 Tokens 12
3.2 Splitting and Merging Tokens 13
3.3 The source of the tokens 13
3.4 Generating tokens from the source and merging tokens back in . 13
3.5 Rules to get the write permission 14
3.6 The relation of tokens and the EFC monoid 14
3.7 Modalities 15
3.8 The *up*-modality 16
3.9 Rules for Acquire Fences 17
3.10 Rules for Release Fences 17
3.11 The life of a non-atomic variable and its tokens 18
3.12 Specification Syntax for Token-based Proofs 18
3.12.1 Source code annotations 18
3.12.2 Using Proof Rules from the RSL logics on IDF-Style Assertions 19

4 Automation of Token-based Proofs **20**
4.1 Merging and splitting resources 22
4.2 Method calls 22
4.3 Release-acquire RMW operations 22
4.4 Generalized RMW operations 23

5 Proofs for example programs **25**
5.1 Spinlock . . . 26
5.1.1 `try_lock_shared` . . . 27
5.1.2 `unlock_shared` . . . 28
5.1.3 `try_lock` . . . 29
5.1.4 `unlock` . . . 29
5.1.5 `unlock_and_lock_shared` . . . 30
5.2 Barrier . . . 30
5.2.1 Proof . . . 32
5.3 ARC . . . 33
5.3.1 Constructor . . . 34
5.3.2 `read` . . . 34
5.3.3 `clone` . . . 35
5.3.4 `drop` . . . 35
5.4 ARC version 2 . . . 36
5.4.1 Constructor . . . 36
5.4.2 `read` . . . 37
5.4.3 `clone` . . . 37
5.4.4 `drop` . . . 38
5.5 ARC comparison to FSL++ . . . 38
5.5.1 FSL++ proof of drop function . . . 38

6 Syntax for C++ input programs **40**

7 C++ to Viper encoding **42**
7.1 Ghost locations and modalities . . . 42
7.2 Encoding Tokens . . . 43
7.2.1 Encoding the Token Counting . . . 43
7.2.2 Encoding the Permission Sum Associated with the Tokens 43
7.2.3 Encoding the Relation between the Token Count and the Permission . . . 43
7.2.4 Inhaling Tokens . . . 45
7.2.5 Exhaling Tokens . . . 45
7.3 Encoding a C++ program . . . 46
7.4 Field declarations . . . 48
7.5 Non-Atomics . . . 48
7.6 Atomics . . . 49
7.7 Local variables . . . 49
7.8 Methods . . . 49
7.9 Assertions . . . 52
7.10 Inhales and Exhales . . . 52
7.11 Fences . . . 52
7.12 Read-Modify-Write Operations . . . 53
7.13 Location Invariants . . . 53
7.13.1 Inhaling the source as part of the location invariant . . . 53
7.13.2 Exhaling the source as part of the location invariant . . . 53

8 Verification Tool for Weak Memory Programs **58**
8.1 Parser 58
8.2 Encoder 58
8.3 Benchmarks 59

9 Conclusion and Future Work **60**
9.1 Conclusion 60
9.2 Future Work 60

Acknowledgments **62**

A More details on Viper encoding **65**
A.1 Fences 65
A.2 `exhaleSource` definition 66
A.3 Basic Viper definitions 68
A.4 Parallel Heaps 69
A.5 Barrier without precondition 70
A.5.1 Specification 70
A.5.2 Proof outline 71
A.5.3 Explanations 71
A.6 Full details for Atomic Reference Counter 73
A.6.1 C++ Code 73
A.6.2 Generated Viper Code 74

Chapter 1

Introduction

The performance that can be achieved by using low-level concurrent C++ is often desired in library-programs. For the sake of performance, the memory model that was introduced in C++11 provides very weak guarantees regarding concurrency: this is the reason why it is called *weak memory.* Weak memory programs are very hard to write, as well as to test, for at least two reasons. First, the weak memory semantics allow even more executions of a program than all the possible sequential interleavings of the threads (which might already be many). Second, some of these executions might be extremely rare and it would be very unlikely to observe them using random test-cases. However, such low-level libraries are not only hard to write and to test, it is also absolutely crucial that they are reliable, because they are used in many complex software projects. Therefore, not even extremely rare failures are acceptable. Subtle bugs in the libraries may affect all of these projects in ways which are hard and frustrating to debug. In practice, these libraries are tested extensively, which is very time-consuming and still, one cannot be 100% sure that the program always is correct. Formal proofs, on the other hand, are also expensive, but they provide something very desirable which is usually impossible to achieve with testing: a guarantee that the program is correct in *all* scenarios.

Several logics for reasoning about weak memory programs have been proposed. To our knowledge, the main ones are RSL [12], FSL [3] and FSL++ [4] (both extensions of RSL) and GPS [11]. In this thesis, we focus on RSL and its extensions (collectively referred to as *RSL logics*). RSL and FSL are easier to automate than FSL++, but they are also much less expressive. In order to encode real world programs, a logic with an expressiveness similar to that of FSL++ is required. The problem with FSL++ is that the high flexibility of the encoding makes it hard to automate proofs. Attempts to define a logic that is possibly slightly less flexible than FSL++, but makes automation easier, have been made [7] but not yet compiled to a finished proposal.

A prototype tool for the automated verification of weak memory programs has been published [10]. It encodes large fractions of RSL and FSL into Viper [6], a language for which reliable and established verifiers exist. However, RSL

and FSL only allow reasoning about individual atomic accesses in isolation. This is often not sufficient for the verification of real-world programs [4]. The reason is that in a particular library implementation, identical operations can have different meanings for verification, depending on what happened earlier in the program. In FSL++, ghost state makes it possible to artificially remember what happened earlier in the program, even when this is not possible to determine by looking at the program state.

This thesis addresses many of the shortcomings of existing logics and tools. Our main contributions are: (1) a high-level specification language for weak memory C++, (2) corresponding proof rules, (3) proofs for example programs that show the effectiveness of our system, (4) a tool that encodes our specification language and a subset of C++ into Viper, a framework for deductive verification.

Chapter 2

Background

2.1 Weak Memory

In his book *The C++ programming language*, Bjarne Stroustrup, the creator of C++, writes "Most programmers do not need to understand a memory model at all and can think of reorderings as amusing curiosities" and goes on to say "as sensible and productive programmers, we stay away from the lowest levels of software whenever we can. Leave those for the experts and enjoy the higher levels that those experts provide for you." [9]

If the typical professional software engineer should not have to worry about memory models, the code which experts provide for managing these models has to be absolutely bug-free. This does not mean that programmers do not have to think about parallel programming in general. It just means that most programmers should use library implementations of locks and other synchronization mechanisms (written by experts), which already take care of the relevant weak memory issues.

Weak memory models allow executions of multi-threaded programs that are not sequentially consistent. This means that the execution cannot be explained by a sequential interleaving of all operations. Consider the example in Figure 2.1. We assume the variables are all initialized before the functions are executed and ignore potential data races. Assuming sequential consistency, the resulting values (x, y) can be $(0, 1)$, $(0, 1)$ or $(1, 1)$. An execution resulting in $(0, 0)$ is not possible with a sequentially consistent memory model, but weaker memory models might allow it.

Achieving sequential consistency would lead to a considerable overhead in modern hardware. One reason is the hierarchical architecture of the caches that are used to speed up memory accesses. An operation takes effect on the local cache of a core long before it is visible for other cores. Sequentially consistent memory models require to synchronize the caches with the main memory more often than weaker memory models, which is expensive in terms of performance.

```
// thread 2:
int c = 0;
extern int b;
int f1()
{
  c = 1; // A
  int x = b; // B
  return x;
}
```

```
// thread 2:
int b = 0;
extern int c;
int f2()
{
  b = 1; // C
  int y = c; // D
  return y;
}
```

Figure 2.1: Example adapted from [9]

2.2 Data races

In C++, memory accesses can be atomic or non-atomic. Non-atomic accesses lead to undefined behavior in the semantics of C++ if two accesses are executed at the same time and at least one of them is a write. This is called a data race. One way to prevent undefined behavior while using non-atomic accesses is to protect them using synchronization libraries. These libraries are implemented using atomic operations.

2.3 Atomics and Synchronization

The C++ semantics guarantee that atomic operations by multiple threads are always ordered, even if multiple threads try to execute atomic operations simultaneously. In other words, atomic accesses are allowed to race. Atomic accesses are much slower than non-atomic accesses, therefore using atomic operations on large shared data would be inefficient. Atomic operations are not only safe themselves, they can also be used to synchronize other memory accesses. Efficient library implementations often protect large amounts of shared data using atomic operation on a few small, dedicated memory locations (e.g. simple boolean flags or integer counters). Atomic operations have a synchronization parameter that can be set to e.g. `memory_order_release` (release), `memory_order_acquire` (acquire) or `memory_order_relaxed`. This parameter is also referred to as the *memory order* of an atomic operation. If all threads perceive a certain *release* operation on a memory location before an *acquire* operation on the same location, we get certain guarantees with respect to the order in which memory operations are perceived: All threads will perceive the effects of operations preceding the release operation before the effects of subsequent operations of the acquire operation. This means that non-atomic operations before (and in the same thread as) the release operation will not race with non-atomic operations after (and in the same thread as) the acquire operation. The same kind of synchronization cannot be achieved using relaxed operations.

2.4 Relaxed Separation Logic

Relaxed separation logic (RSL) is a logic that can be used to verify properties of weak memory programs. These programs have to be written in a simplified weak-memory programming language. This programming language is only used for verification purposes and never for programs that are actually executed. One of the simplifications in this programming language is that memory locations can either only be used with atomic operations or only with non-atomic operations. Theses types of locations are called atomic locations and non-atomic locations, respectively. An ownership model is used to prove the absence of data races. $loc \overset{1}{\mapsto} _$ denotes the full ownership/permission (we use the two term interchangeably) of the non-atomic memory location loc with an arbitrary value at this location. $loc \overset{1}{\mapsto} v$ stands for the full ownership and the knowledge that the value is v. With the full permission, a thread is allowed to write to the location. Threads can also have partial ownership of the location, e.g. $loc \overset{1/2}{\mapsto} v$. With partial permission a thread can read from a location but not write to it. When a location is allocated, the permission $loc \overset{1}{\mapsto} _$ is created for the allocating thread. This permission can then be split into parts which can be distributed to other threads and added up again, but there is no way more than the full permission can exist because permission cannot just be created out of thin air. This system guarantees that no write operation can be executed at the same time as another read or write operation, because this would in total require more than the full permission.

Read/write operations: In RSL, threads can send permission to each other using operations on atomic locations. As shown in Figure 2.2, write operations are used to send permission and read operations to receive permission. A so-called location invariant on the atomic location defines exactly which permission has to be sent depending on the value that is written. Reading the value results in receiving the permission. The location invariant is a function from values to assertions. Permission can be contained in the assertions, but the assertions can also contain other information (i.e. that the value at the location always is in a certain range). Assume the location invariant Q on the left side of Figure 2.2 is defined as $Q(v) = x \overset{1}{\mapsto} _$. This invariant does not even depend on the value of the location. In order to write any value, a thread must give up the full permission to the location x. When a thread reads any value, it gets the full permission to x. This is a simplified explanation. In reality, the full RSL Logic is more complex. For example there are mechanisms that make sure that if the same value twice is read twice, the assertion $Q(v)$ is still only received once.

We saw how RSL models sending and receiving of resources. But we also mentioned that weak memory models do not guarantee sequential consistency. This means that a thread can observe the operations of another thread in a different order than the thread that executes them. Clearly, we need some ordering guarantees for sending around and using permission safely. This is exactly what release-acquire synchronization gives us. If thread A sends resources to thread B via an atomic write and corresponding read, A's usages of the resource

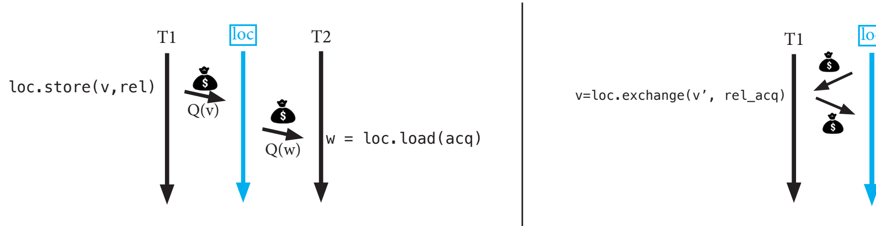

Figure 2.2: Atomic write and read operation on the left and RMW-operation on the right.

are guaranteed to be ordered before B's usages of the resource.

Read-modify-write (RMW) operations: Another kind of atomic operations are RMW operations. With RMW operations, threads both read and write to the location with a single atomic operation. In RSL, this means that resources can be sent and received with a single operation. In RSL, a location has to specify whether its value can be read via normal reads or only via RMW operations. Write operations are possible on both types of locations. The location invariant conceptually defines the permission that is at the location depending on the value. If the value that is read is the same as the one that is written, the thread does not gain or loose any permission. But if, say, the location needs more permission with the new value than it had before, this permission has to come from the thread and the thread will end up with less permission than it had before the operation. An example of such a scenario would be a RMW operation used to release a mutex.

2.5 Fenced Separation Logic (FSL)

We have seen how release-writes and acquire-reads can be used for synchronization. Relaxed writes and reads on the other hand do not synchronize in the same way, but FSL still allows to send resources via a relaxed write, but only if it is guaranteed that all the operations that need the resource are completed. This guarantee can be provided by a release fence. Figure 2.3 shows how permission can be transferred via a relaxed write and a relaxed read. The acquire fence makes sure that all operations after the fence are only visible after the fence. In combination, the two fences make sure that operations before the release fence are completed before operations after the acquire fence start. In this way, the permission can be transferred safely, without being used by both threads simultaneously. In order to reason about the effect of fences, FSL introduces modalities. Resources that were prepared (using a fence) for sending with a relaxed write are labeled with $\triangle$. Resources that were received through a relaxed write and are not usable before an acquire fence is hit are labeled with $\triangledown$.

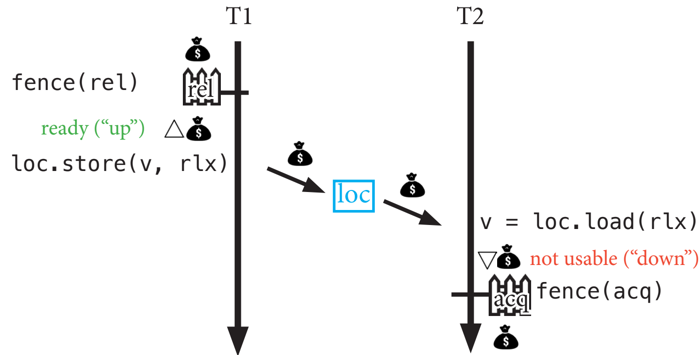

Figure 2.3: Fences and relaxed read/write

2.6 FSL++

In FSL++ it is possible to add memory locations, which are not present in the original program. These so-called *ghost*-locations are only used for verification purposes. The ownership of such locations can be represented by arbitrary partial commutative monoids (PCM). PCMs are associative, commutative algebras and have a neutral element. Since they are partial, the results of some operations can be undefined. This rich facility for defining ghost state makes FSL++ very expressive. PCMs can be tailored to match the exact program that is being verified. Defining a PCM becomes part of writing the specification. This can be a very hard task. The proof of the correctness of the Rust Atomic Reference Counter given in [4] shows how a custom PCM is tailored to exactly contain the information needed for this specific proof. Because of this modular aspect of FSL++, it is hard to automate and automation of FSL++ has only been achieved for simple monoids [10].

2.7 Entity Fractional Counting

Entity Fractional Counting (EFC), a monoid for FSL++, was presented in [7]. This monoid was shown to be effective for the verification of real world weak memory examples. For some programs, a custom-made monoid might still be required, but the EFC monoid seems to capture the intuition of why a weak-memory program is correct very well. Is is unclear how reasoning with the EFC monoid could be automated. One downside of the EFC monoid, is that it uses fractions at places where the only information that seems to matter is whether the fraction is zero, nonzero or exactly one.

Chapter 3

Token-based Reasoning for Weak Memory Programs

As we saw in the previous chapter, permissions play an important role for the verification of parallel programs, and therefore also for the verification of weak memory programs. The RSL logics use fractional permissions for non-atomic locations, and arbitrary PCMs for ghost locations. Fractional permissions have the advantage that when using them, it is straightforward to model the fact that the read permission can be distributed to an arbitrarily amount of threads. However, there are also significant disadvantages. First, it is hard to model *counting* using fractional permissions. This is because fractional permissions are between 0 and 1 and the counting should be unlimited. As soon as we define the fraction that corresponds to some count, the counting is limited from above. Counting can be very useful for the verification of real-world programs, e.g. to count the number of threads that have read permission. A second disadvantage of fractional permissions is that, when using them in proofs, we have to worry a lot about the exact fractions, even though for many real-world examples, what matters is only the difference between no access, read access and write access. This adds unnecessary complexity to the proofs and usually does not reflect the intuition of why a program is correct. For the real-world examples that we are tackling in this work, it seems that they pass read/write access to non-atomic locations around, and track how *many* threads have access, rather than how *much*.

The idea of having a permission structure that allows the counting of the number of threads to which permission was given, is also what motivated the EFC monoid. The EFC monoid relates a number of entities (one can think of them as permission “chunks”) to the permission these entities add up to. The EFC monoid is the basis for the permission structure that is introduced in this work. Our goal is to define a structure similar to the EFC monoid, that is independent of fractions and facilitates automation. In proofs, a ghost location that is governed by the EFC monoid is closely related to a non-atomic memory

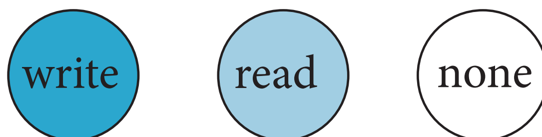

Figure 3.1: The three types of tokens.

location in the program. In our work, we want to use this fact to make the syntax easier.

3.1 Tokens

We now introduce a new permission structure that provides counting and a distinction between no permission, read permission and write permission.

In our logic, permission is held in the form of tokens. They give the owner permission to a certain non-atomic location in memory. It is not possible to have permission without owning any tokens. If multiple tokens for the same non-atomic location are owned by one thread, they can be combined. In case they are combined, only the number of tokens and the permission that they give *overall* to the owner are remembered. The notation for expressing the ownership of tokens for the non-atomic location *loc* is:

$$\mathsf{Tok}(loc, n, \tau) \text{ where } n \in \mathbb{N}_{>0} \text{ and } \tau \in \{none, read, write\}$$

The positive integer n is the number of tokens that are held while τ is the actual permission that the tokens give to the owner. The three kinds of tokens are also shown on figure 3.1. τ tells us what the thread can do with the location for which it has the tokens. With *none* it cannot do anything, with *read* it can read from the location and with *write* it can read and write. Read or write access can only be held in the form of tokens.

A write-token can be exchanged for a read-token:

$$\mathsf{Tok}(l, n, write) \quad \models \quad \mathsf{Tok}(l, n, read) \quad (\text{TOK-WRITE-IS-READ})$$

But a read-token cannot be exchanged for a none-token.

$$\mathsf{Tok}(l, n, read) \quad \not\models \quad \mathsf{Tok}(l, n, none)$$

This is because read-tokens cannot be sent using relaxed operations, whereas none-tokens can.

3.2 Splitting and Merging Tokens

For $n > 0$ and $m > 0$:

$$\text{(NONE-TOK-NONE-TOK)}\quad \mathsf{Tok}(l, n, none) * \mathsf{Tok}(l, m, none) \equiv \mathsf{Tok}(l, n+m, none)$$

$$\text{(READ-TOK-READ-TOK)}\quad \mathsf{Tok}(l, n, read) * \mathsf{Tok}(l, m, read) \equiv \mathsf{Tok}(l, n+m, read)$$

$$\text{(WRITE-TOK-NONE-TOK)}\quad \mathsf{Tok}(l, n, write) * \mathsf{Tok}(l, m, none) \equiv \mathsf{Tok}(l, n+m, write)$$

$$\text{(READ-TOK-NONE-TOK)}\quad \mathsf{Tok}(l, n, read) * \mathsf{Tok}(l, m, none) \equiv \mathsf{Tok}(l, n+m, read)$$

$$\text{(WRITE-TOK-READ-TOK)}\quad \mathsf{Tok}(loc, n, write) * \mathsf{Tok}(loc, m, some) \models False$$

$$\text{(WRITE-SPLIT)}\quad \mathsf{Tok}(l, n+m, write) \models \mathsf{Tok}(l, n, read) * \mathsf{Tok}(l, m, read)$$

Note that the $*$ is usually cancellative, meaning that $A_1 * A_2 \equiv A_1 * A_3 \implies A_2 \equiv A_3$. In our logic, $*$ is *not* cancellative, because otherwise the second and the third rule would imply $\mathsf{Tok}(l, n, read) \equiv \mathsf{Tok}(l, n, none)$.

3.3 The source of the tokens

Another resource in our logic is the so-called *source*. There is exactly one token source per non-atomic location, and all tokens for the non-atomic location come from this unique source. The source can be used to generate tokens. The notation is as follows:

$$\mathsf{Src}(loc, n, \tau) \text{ where } n \in \mathbb{N}_{\geq 0}, \text{ and } \tau \in \{none, read, write\}$$

n keeps track of the number of tokens that were given out and τ keeps track of the kind of tokens that the source can generate. The source can never run out of tokens. At the source, we are allowed to replace the *write*-state with *read* and *read* with *none*. This is not the same as for tokens, because one cannot exchange a read-token for a none-token. We never send the source with relaxed operations, because we only allow the source at the location invariant, which means that we do not send the source around at all.

$$\mathsf{Src}(l, n, write) \models \mathsf{Src}(l, n, read) \quad \text{(SRC-WRITE-IS-READ)}$$
$$\mathsf{Src}(l, n, read) \models \mathsf{Src}(l, n, none) \quad \text{(SRC-READ-IS-NONE)}$$

3.4 Generating tokens from the source and merging tokens back in

The rules in this section show how the source of the tokens interacts with tokens.

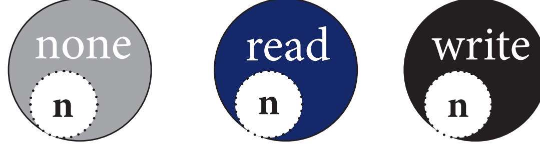

Figure 3.2: The three possible states of the token-source with n missing tokens.

$$\textsc{(none-src-none-tok)} \quad \mathsf{Src}(l, n+m, none) * \mathsf{Tok}(l, m, none) \equiv \mathsf{Src}(l, n, none)$$

$$\textsc{(read-src-read-tok)} \quad \mathsf{Src}(l, n+m, read) * \mathsf{Tok}(l, m, read) \equiv \mathsf{Src}(l, n, read)$$

$$\textsc{(none-src-write-tok)} \quad \mathsf{Src}(l, n+m, none) * \mathsf{Tok}(l, m, write) \equiv \mathsf{Src}(l, n, write)$$

$$\textsc{(any-src-read-tok)} \quad \mathsf{Src}(l, n+m, any) * \mathsf{Tok}(l, m, read) \equiv \mathsf{Src}(l, n, read)$$

$$\textsc{(read-src-none-tok)} \quad \mathsf{Src}(l, n+m, read) * \mathsf{Tok}(l, m, none) \equiv \mathsf{Src}(l, n, read)$$

$$\textsc{(write-src-split)} \quad \mathsf{Src}(l, n, write) \models \mathsf{Src}(l, n+m, read) * \mathsf{Tok}(l, m, read)$$

$$\textsc{(write-src-read-tok)} \quad \mathsf{Src}(l, n, write) * \mathsf{Tok}(l, m, read) \models \mathit{False}$$

$$\textsc{(too-many-tokens)} \quad \mathsf{Src}(l, n, \tau_1) * \mathsf{Tok}(l, m, \tau_2) \models \mathit{False} \text{ if } m > n$$

3.5 Rules to get the write permission

Until now, we saw several rules for splitting the write permission into read permissions. There is one way to get back the write permission after it was split into read permissions. If the source is not missing any tokens, we know that the write permission is at the source, even if the keyword says something different. This fact can be used with the following rules:

$$\mathsf{Src}(l, 0, none) \models \mathsf{Src}(l, 0, write) \quad \textsc{(all-tokens-a)}$$
$$\mathsf{Src}(l, 0, read) \models \mathsf{Src}(l, 0, write) \quad \textsc{(all-tokens-b)}$$

3.6 The relation of tokens and the EFC monoid

All possible tokens- or source-assertion that we saw are closely related to an assertion in FSL++ using the EFC monoid for the ghost locations. $\boxed{\alpha : (n, q)^+}$

represents the ownership of the permission $(n, q)^+$ to the ghost location α, where $(n, q)+$ represents n entities that sum up to q. Having "-" in the exponent (as in some of the expressions below) means that the expression refers to entities that are missing for the full permission (which is $(0, 0)^-$). For more details, see [7]:

$$\begin{aligned}
\mathsf{Tok}(loc, n, write) &\approx loc \overset{1}{\mapsto} _ * \boxed{\alpha : (n, 1)^+} \\
\mathsf{Tok}(loc, n, read) &\approx \exists q.\quad q \in (0, 1] \wedge loc \overset{q}{\mapsto} _ * \boxed{\alpha : (n, q)^+} \\
\mathsf{Tok}(loc, n, none) &\approx \boxed{\alpha : (n, 0)^+} \\
\mathsf{Src}(loc, n, write) &\approx loc \overset{1}{\mapsto} _ * \boxed{\alpha : (n, 0)^-} \\
\mathsf{Src}(loc, n, read) &\approx \exists q.\quad q \in (0, 1] \wedge loc \overset{q}{\mapsto} _ * \boxed{\alpha : (n, 1 - q)^-} \\
\mathsf{Src}(loc, n, none) &\approx \exists q.\quad q \in [0, 1] \wedge loc \overset{q}{\mapsto} _ * \boxed{\alpha : (n, 1 - q)^-}
\end{aligned}$$

α is a ghost location that is governed by the EFC permission structure. If we plug the corresponding FSL++ assertions in for all of our introduced rules, the rules can be proven correct using the FSL++ rules. The keywords that we are using for our abstract reasoning do not always have a direct translation to a fraction. For example, the read permission can correspond to any non-zero fractional permission. The last of the embeddings is maybe the most surprising one. Having a source that can only give none tokens corresponds to owning an arbitrary fractional permission amount to the location. One way to understand why this makes sense is that even if we could have some permission, we cannot be *sure* to have some, so it would be unsound to add a rule which allows to give away nonzero fractional permission (i.e. a read token).

3.7 Modalities

Modalities were briefly introduced in the previous Chapter. The up-modality ($\triangle$) is used for resources that can be sent using a relaxed write. The down-modality ($\triangledown$) is used for resources that were received with a relaxed read and, therefore, cannot be used without an acquire fence. With the concept of modalities added, the notation is now extended: in $\mathsf{Tok}(loc, n, \tau)$, τ can be instantiated with many more options. For any $n > 0$, τ can be in

$$\{none, read, write, \triangledown read, \triangledown write, \triangle read, \triangle write\}$$

If $n > 1$, τ can additionally be in

$$\{(\triangledown read, read),\ (\triangledown read, read, \Sigma write),\ (read, \triangle read),\ (read, \triangle read, \Sigma write)\ \}$$

and if $n > 2$, τ can be all of the above and additionally in

$$\{(\triangledown read, read, \triangle read),\ (\triangledown read, read, \triangle read, \Sigma write)\}$$

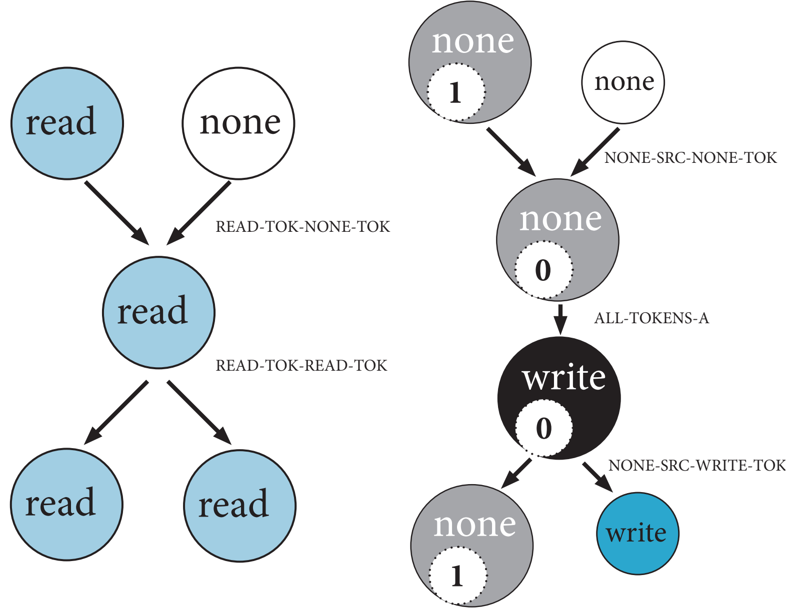

Figure 3.3: Two example derivations and the rules that were used.

For example $\mathsf{Tok}(loc, n, (\triangledown read, read))$ means that we own at least one read token that we received with a relaxed read (a $\triangledown read$-token), and at least one normal read-token. This is the reason why n must be at least 2. Except that there is at least one of each kind, the distribution of the n tokens over the modalities is flexible, e.g. the following equality holds for any $i, j > 0$ such that $i + j = n$:

$$\mathsf{Tok}(l, i, \triangledown read) * \mathsf{Tok}(l, j, read) \quad \equiv \quad \mathsf{Tok}(l, i + j, (\triangledown read, read))$$

$\Sigma write$ means that the tokens carry the full (write) permission between them. There are several ways how this information can get useful, as we will see below.

3.8 The *up*-modality

We add rules that do not exist in FSL++:

$$\begin{aligned} \mathsf{Tok}(l, n, \triangle read) &\models \mathsf{Tok}(l, n, read) \\ \mathsf{Tok}(l, n, \triangle write) &\models \mathsf{Tok}(l, n, write) \end{aligned}$$

We essentially allow the removal of the $\triangle$-modality. Note that we do not allow to add it again and we do not allow something similar for the $\triangledown$-modality. This is sound, because whenever we remove the modality from a resource using our new rule, we could also just not have put it under the modality in the first place.

In FSL++, we are never forced to put anything under the modality whenever we hit a release fence. For each proof that is done using our new rule, a similar proof that uses the FSL++ style could be constructed. This proof can be constructed as follows. First, just use the whole permission state that the thread owns at the moment when it hits a release fence as annotation of the release fence (i.e. everything gets the $\triangle$-modality). Second, continue the verification and whenever a resource is needed without the up-modality that is under the up-modality, remove this resource from the annotation of the fence. These two steps have to be repeated for each release fence. Formally it would make sense to use another symbol instead of $\triangle$ and call it *maybe-up*, but it seems that this would add unnecessary complexity.

3.9 Rules for Acquire Fences

We use $\texttt{fence}_{\texttt{acq}}$ as shorthand for `atomic_thread_fence(memory_order_acquire)`. We assume that we are always reasoning about the same location, therefore we omit it. The idea behind these rules is that whenever we hit an acquire-fence, we can remove the down-modality everywhere. In some cases, this can mean that we get write access.

For $n \geq 1$:

$$\{\mathsf{Tok}(n, \triangledown read)\} \quad \texttt{fence}_{\texttt{acq}} \quad \{\mathsf{Tok}(n, read)\}$$
$$\{\mathsf{Tok}(n, \triangledown write)\} \quad \texttt{fence}_{\texttt{acq}} \quad \{\mathsf{Tok}(n, write)\}$$

For $n \geq 2$:

$$\{\mathsf{Tok}(n, (\triangledown read, read))\} \quad \texttt{fence}_{\texttt{acq}} \quad \{\mathsf{Tok}(n, read)\}$$
$$\{\mathsf{Tok}(n, (\triangledown read, read, \Sigma write))\} \quad \texttt{fence}_{\texttt{acq}} \quad \{\mathsf{Tok}(n, write)\}$$

For $n \geq 3$:

$$\{\mathsf{Tok}(n, (\triangledown read, read, \triangle read))\} \quad \texttt{fence}_{\texttt{acq}} \quad \{\mathsf{Tok}(n, (read, \triangle read))\}$$
$$\{\mathsf{Tok}(n, (\triangledown read, read, \triangle read, \Sigma write))\} \quad \texttt{fence}_{\texttt{acq}} \quad \{\mathsf{Tok}(n, (read, \triangle read, \Sigma write))\}$$

3.10 Rules for Release Fences

We use $\texttt{fence}_{\texttt{rel}}$ as shorthand for `atomic_thread_fence(memory_order_release)` and again omit the location. These rules add the up-modality to all the permission that does not have a modality.

$$
\begin{array}{rcl}
\{\mathsf{Tok}(n, read)\} & \texttt{fence}_{\texttt{rel}} & \{\mathsf{Tok}(n, \triangle read)\} \\
\{\mathsf{Tok}(n, write)\} & \texttt{fence}_{\texttt{rel}} & \{\mathsf{Tok}(n, \triangle write)\} \\
\{\mathsf{Tok}(n, (read, \triangle read))\} & \texttt{fence}_{\texttt{rel}} & \{\mathsf{Tok}(n, \triangle read)\} \\
\{\mathsf{Tok}(n, (read, \triangle read, \Sigma write))\} & \texttt{fence}_{\texttt{rel}} & \{\mathsf{Tok}(n, \triangle write)\} \\
\{\mathsf{Tok}(n, (\triangledown read, read))\} & \texttt{fence}_{\texttt{rel}} & \{\mathsf{Tok}(n, (\triangledown read, \triangle read))\} \\
\{\mathsf{Tok}(n, (\triangledown read, read, \Sigma write))\} & \texttt{fence}_{\texttt{rel}} & \{\mathsf{Tok}(n, (\triangledown read, \triangle read, \Sigma write))\} \\
\{\mathsf{Tok}(n, (\triangledown read, read, \triangle read))\} & \texttt{fence}_{\texttt{rel}} & \{\mathsf{Tok}(n, (\triangledown read, \triangle read))\} \\
\{\mathsf{Tok}(n, (\triangledown read, read, \triangle read, \Sigma write))\} & \texttt{fence}_{\texttt{rel}} & \{\mathsf{Tok}(n, (\triangledown read, \triangle read, \Sigma write))
\end{array}
$$

3.11 The life of a non-atomic variable and its tokens

acc(loc) is created when a non-atomic location is freshly allocated and initialized. A source is created with the first write operation to the atomic location which has this specific source in its location invariant. If this first write operation is relaxed, the thread either has to own $\triangle$**acc**(loc) or keep a write-token. If the first write is a release-write or if the thread owns $\triangle$**acc**(loc), the thread can also keep only read access or no access at all. As soon as the non-atomic location is "managed" by an atomic location, the thread does not hold any permission in form of **acc**(loc) anymore, and can own tokens instead.

3.12 Specification Syntax for Token-based Proofs

The tokens and the source explained above can be used in the specification of a program. They replace existing approaches for permissions to non-atomic locations. This is not enough for the verification of weak memory programs, but the rest stays the same as in the existing work on automation of weak memory verification [10].

3.12.1 Source code annotations

A central element in specifications is assertions. Our syntax for assertions:

$$
\begin{array}{rcl}
A & ::= & e \mid \mathsf{Tok}(l, n, \tau) \mid \mathsf{Src}(x, n, \tau) \\
 & \mid & A_1 * A_2 \mid e \Rightarrow A \mid (e \; ? \; A_1 : A_2) \mid \mathsf{Uninit}(l) \mid \mathsf{Acq}(l, \mathcal{Q}) \mid \mathbf{acc}(l) \\
 & \mid & \mathsf{Rel}(l, \mathcal{Q}) \mid \mathsf{Init}(l) \mid \triangle A \mid \triangledown A \mid \mathsf{RMWAcq}(l, \mathcal{Q})
\end{array}
$$

where e is an (otherwise pure) expression that is allowed to contain $x.val$ if the context in which it appears contains read permission to x. We often use $\mathsf{Rmw}(l, Q)$ where

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

This is a useful short form because it contains exactly what is needed in order to perform a RMW operation. Preconditions and postconditions are annotated with assertions as defined above, and location invariants as functions from values to assertions. The Src() is only allowed to appear in the location invariant and never at the thread-level and Tok() can only appear at the thread-level.

3.12.2 Using Proof Rules from the RSL logics on IDF-Style Assertions

The RSL logics are based on separation logic, whereas our logic uses a Viper-like Implicit Dynamic Frames [8] syntax. Our proof rules are all derived from FSL++. Why is this sound? Because for our real-world examples, the IDF-formulas on which we apply the rules usually correspond to a formula in separation logic, on which the separation logic version of the rules hold.

Chapter 4

Automation of Token-based Proofs

The token-based specification and the proof rules that we provided are arguably at a much higher abstraction level than FSL++. However, we did not yet see how our logic is more suitable for automation than existing approaches. In this chapter we present a proof style that determines which rules should be applied at which step of the verification. In a proof there are often point where multiple valid rules could be applied and it is not clear which one ultimately leads to the assertion we actually want to prove. In the following we show guidelines that determine which outcomes of applying rules are preferred over others.

1. **The threads get as much permission as possible.** With "as much as possible" we mean that we prefer *write* or $\Sigma write$ over any form and combination of *read*s and and any *read* over *none*. E.g. if a thread owns $\mathsf{Tok}(loc, 2, read)$ and calls a function that needs $\mathsf{Tok}(loc, 1, read)$, it would itself keep $\mathsf{Tok}(loc, 1, read)$ and not $\mathsf{Tok}(loc, 1, none)$ (both would be possible according to the rules). Read and write tokens are not strictly more powerful than none tokens, because e.g. a relaxed RMW-operation that fails if the thread has a read token could potentially succeed if the thread would have a none token. So why is giving as much permission as possible to the thread the right choice? In practice, it appears that it never happens that none-tokens are distributed just to allow for relaxed operations. If we happen to only get a none-token even though we wanted read access, then we can at least get rid of it cheaply. However, this does not mean that we should favor none-tokens simply because it is easy to get rid of them later on.

   Why should we always prefer *write* over *read*? There is nothing the owner can do with read access that they cannot do with write access. The only advantage of not having the write permission is that someone else can be sure that we cannot modify the data. However, this person should keep a

read permission anyway, because otherwise they cannot be sure that the data does not change. Now if they keep read access for themselves, there cannot even be the option for us to get the write access. This shows that if we have the opportunity to get the write permission, we should take it.

Another non-trivial application of this principle would be to prefer *write* over $\triangle read$. This scenario can happen if a thread has $\triangle read$ and can get a read token from a location invariant, such that both would add up to write, but the thread can only have one token. Then it will remove the up-modality from the token it already has, and end up with a write-token. This is just an application of the principle that we favor *write* over anything else. The reason why we think this makes sense is that if the idea of the proof is that no-one ever has the write permission, the location invariant can just always keep the read permission. Otherwise, chances are that the situations where a thread *can* get the write permission, are the situations where we *want* it to get the write permission.

2. **Prefer the real heap over the *down*-heap.** If a thread already has read access and performs a RMW operation without the acquire memory order, it could in theory get another read token from the location invariant, which would then end up under the down-modality. This does not seem to be useful and for this reason our automated proofs would not do that. There is one exception if what the thread already has and the read token it can get from the location invariant sum up to *write*. In this case we will add the read-token and the information that the sum is *write* to the permission state of the thread.

3. **Prefer the *up*-heap over real heap.** This rule works similarly to the previous one. If the thread already has $\triangle read$ it will avoid additionally getting *read* or $\triangledown read$ except in case the tokens add up to *write*. But not only that, it should also actively try to get rid of *read* if it already has $\triangle read$. E.g. if the thread performs a RMW-operation with the release memory order, it will use this to get rid of the read token (except in case the tokens add up to *write*).

These principles lead us to conclude the following order of preference for the permission at the thread-level:

1. $\triangle write$
2. $(read, \triangle read, \Sigma write)$
3. $write$
4. $(\triangledown read, read, \triangle read, \Sigma write)$
5. $(\triangledown read, read, \Sigma write)$
6. $\triangle read$

7. $(read, \triangle read)$
8. $read$
9. $(\triangledown read, read, \triangle read)$
10. $(\triangledown read, read)$
11. $\triangledown read$
12. $none$

4.1 Merging and splitting resources

When adding permissions together, the rules in Chapter 3 never leave a choice for what the resulting permission is. However, after merging, we always try to apply the rule that gives us the write permission at the source if no tokens are missing (ALL-TOKENS-A and ALL-TOKENS-B). This is a crucial part of our automation approach because it helps in giving as much permission as possible to the threads. In the programs that we verify, we always split resources when we want to remove something specific from a state. What is left after the removal of the specified resources is not given, and sometimes multiple rules could be applied. The "right" rule to apply (according to our system) is the one that leaves as much permission as possible in the part that is not removed. With "as much as possible" we refer to the order of preference introduced above.

4.2 Method calls

When a method is called, the current resources have to be divided into what the caller keeps and what the callee gets. Since what the callee gets is already fixed (it is written in its precondition), the only question is what the caller can keep. The answer is straightforward: the caller gets as much permission as possible (referring to the order of preference shown above) while the callee gets enough to match his precondition. After the method call, we simply add the postcondition of the method to the current state.

4.3 Release-acquire RMW operations

Figure 4.1 shows how a read-modify-write operation is verified in case the memory order is `rel_acq`. We can simply think of it as adding $Q(v)$ to the current state of the thread (in the way described above). Then, the merged state is split into a part that will go to the location invariant ($Q(v_{new})$) and a leftover that the thread can keep. The split is done in a way that maximizes the thread can keep.

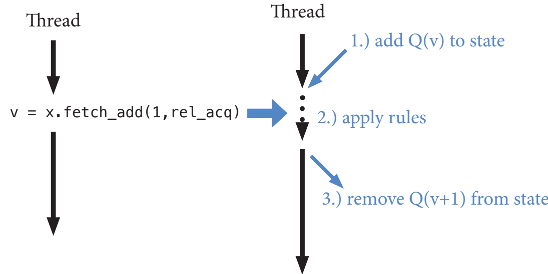

Figure 4.1: Automation of RMW-operations with `rel_acq` memory order (`fetch_add` example).

4.4 Generalized RMW operations

If the memory order is `rel`, `acq` or `rlx`, the situation is a bit more complicated and the simple explanation from above does not quite fit. A more general way to think about it is the following (see Figure 4.2): Before the operation, the thread has a permission state P. The previous value at the location invariant is some value v. $Q(v)$ defines the resources that are at the atomic location before the RWM operation. Let v' be the new value at the atomic location. Then $Q(v')$ defines the resources that need to be at the invariant after the RMW-operation. This can fail if there is no way to get this amount of resources. The resources for $Q(v')$ can come either from $Q(v)$ or from P. We can take them from $Q(v)$ regardless of the memory order (green, dotted arrow). If the memory order is `acq` or `rlx`, what comes from P (orange, diagonal arrow) has to be under the up-modality, except for none-tokens. What ends up at the location is fixed, but what the thread gets (P') depends on how we apply the rules, i.e. how we "fill" the diagonal arrows in Figure 4.2. This is again done in the way that maximizes the permission in P' (while leaving enough for $Q(v')$). What the thread previously had it can keep regardless of the modality (green, dotted arrow). If it wants to get rid of some permission, it can only do so if it had the up-modality or the memory order is `rel`. Permission it gets from $Q(v)$ ends up on the down-heap if the memory order is `rel` or `rlx`. Not satisfying $Q(v')$ is the only way the RMW can fail, but it can happen in nontrivial ways. E.g. assume $P = \mathsf{Tok}(a, 1, read)$, $Q(v) = \mathsf{Src}(a, 2, none)$ and $Q(v') = \mathsf{Src}(a, 1, none)$ and the memory order is `rlx`. The source does not want any permission from the thread, but it needs an additional token. This token cannot come from the thread, because given the relaxed memory order, only none-tokens and tokens that have the up-modality can be sent in this direction. This RMW-operation fails and it would not fail if the thread had a none-token instead of a read-token.

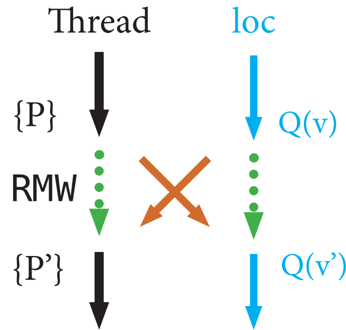

Figure 4.2: Automation of general RMW-operations.

In this explanation, unlike in the simplified one in Section 4.3, P and $Q(v)$ are never explicitly combined. However, we can still learn something by imagining what would happen if we combined P and $Q(v)$. First, we can learn that a certain value v cannot even be read, because $P * Q(v) \models \mathit{False}$. Second, we check whether P contains all the tokens for some non-atomic location that are missing in $Q(v)$. If it does, and $Q(v')$ happens to not keep any permission to the non-atomic location, we know that P' contains the write permission.

Chapter 5

Proofs for example programs

We apply our proof system manually on three examples: a spinlock and a barrier from the Folly library and the atomic reference counter from the Rust programming language. Proofs for these examples using FSL++ and the EFC monoid have been given in [7], but we now prove them on the level of abstraction presented in Chapter 3, in a way that can be automated using the ideas presented in Chapter 4. The goal of verification is set by the pre- and postconditions of the functions. For our examples, the only annotations that are needed additionally consist of location invariants. After the specification we show proof outlines where we explicitly show the resources that a thread owns after each line of code. The resources that the thread owns before the first instruction is the precondition. After the last instruction, the current state of the thread should equal the postcondition. The interesting parts of the proof outlines are the read-modify-write (RMW) operations. For those, we provide detailed explanations. For the RMW operations we essentially prove two properties. First, that the RMW operation is successful, meaning that the location invariant can be re-established for the new value at the atomic location. Second, that after the RMW operation, the thread actually owns the resources which we claim it owns in our proof outline. We always leave out $\mathsf{Rmw}(l, Q)$ $(= \mathsf{Init}(l, Q) * \mathsf{Rel}(l, Q) * \mathsf{RMWAcq}(l, Q))$ from the postconditions in order to keep them short. In theory this could lead to the situation where a caller of the functions loses $\mathsf{Rmw}(l, Q)$. However, because $\mathsf{Rmw}(l, Q)$ is duplicable, the caller of the functions can in practice always create a backup duplicate before calling a function. Also, we do not mention $\mathsf{Rmw}(l, Q)$ at all in the proof outlines, because for functions in which this was in the precondition it is easy to show that it is always present whenever needed.

The actual C++ code that we verify is never exactly the same as in the original, but the core functionality and (crucially) the modes of synchronization are the same. The atomic reference counter was not written in C++ at all, so we just verify our own C++ implementation of it. In the original code that we

got from the Folly library, as well as in our adapted versions, the functions are all methods of a class and the (atomic and non-atomic) memory locations are fields of the class.

In the following sections we do not show the whole C++ classes. For each class we first show the field declarations and then discuss the methods individually. The code that we show is a mix between C++ and annotations, which are also written in C++ syntax (but also using language features that are not part of the C++ language). The features that we use for annotations are declared in our own C++ header file. The detailed syntax for the C++ and annotations that we support can be found in Chapter 6. The location invariants are added as annotations directly to the C++, but we show them in a more readable format here (see below).

5.1 Spinlock

In this section we prove that the reader-writer-spinlock in the Folly library is correct. The lock has a state that is stored in the atomic integer `bits_`. The least significant bit of `bits_` is 1 if there currently is a writer and 0 otherwise. In the original implementation, the second least significant bit is used if a thread has a read lock and wants a write lock. In our adapted version however, the second least significant bit is not used. It seems that adding support for this additional functionality would not make the proof harder (only longer). The number encoded by the rest of the bits represents the number of readers. The number of readers is the value divided by 4 and rounded down. When threads try to get read access, they increment the value at the atomic location by 4. A thread also increments the value of `bits_` by 4 if there is already a writer. When inspecting the value that it read from the atomic location, it will later discover that it did not actually get read access and decrement the counter. This means that `bits_/4` (using integer division) does not necessarily correspond to the number of threads that actually have read access. It also includes threads that tried to get read access and did not yet decrement the value.

The actual spinning is not part of the code that we verify here. We verify the functions that are repeatedly called if a thread tries to get a lock and that e.g. a client who gets the write lock (potentially after trying to get it many times) can be sure no one else has a reader or a writer lock.

Location invariant

The proof we are showing uses the following location invariant for the atomic location `bits_`:

$$
\begin{aligned}
Q(v) \quad := \quad & \text{let } n = \lfloor \tfrac{v}{4} \rfloor, w = lsb(v) \text{ in:} \\
& v \geq 0 \land \\
& (w = 1 \ ?\ \mathsf{Src}(res, n+1, none) * \mathsf{Src}(rds, n, write) \\
& :\ \mathsf{Src}(res, n, read) * \mathsf{Src}(rds, n, read))
\end{aligned}
$$

res stands for resource and it represents the data that the lock protects. The ghost location *rds* stands for readers and is used to keep track of the reader locks that were distributed. This is used to make sure that if someone has a read lock, no-one can have a write lock, not even the thread itself. In other words the information we capture is that threads cannot themselves transform a write lock into a read lock: if they want to do so, they have to call `unlock_and_lock_shared`.

Field definitions

```
int res;
int rds = GhostRef();
atomic<int> bits_ = atomic<int>(Q);
```

5.1.1 try_lock_shared

Specification, Code and Proof outline

```
bool try_lock_shared() {
    ////////////////////////////////////////
    Requires(Rmw(bits_,Q));
    Ensures(boolRes() ? (Tokens(res, 1, read) && Tokens(rds, 1, read))
                : true);
    ////////////////////////////////////////

    {Rmw(bits_, Q)}

    bool ret;
    int value;
    value = bits_.fetch_add(4, memory_order_acquire); // RMW1
    {v0  mod 2 = 1  ?  Tok(res, 1, none) * Tok(rds, 1, none)
    : Tok(res, 1, read) * Tok(rds, 1, read)}
    if (value % 2 == 1) {
      {Tok(res, 1, none) * Tok(rds, 1, none)}
      bits_.fetch_add(-4, memory_order_release); // RMW2
      ret = false;
    }
    else {
      ret = true;
    }
    return ret;
    {ret  ?  Tok(res, 1, none) * Tok(rds, 1, none) : true}
  }
};
```

RMW1

Case $lsb(v) = 0$: The location invariant contains *read* permission to *res* and to *rds*. Increasing the counter by 4 makes the counter for missing tokens go up by one for both *rds* and *res*, which means that the invariant gives away a token of each. Both are read-tokens, because taking tokens from a *read*-source produces *read*-tokens.

Case $lsb(v) = 1$: The location invariant has *none* permission to *res* and *write* to *rds*. As before, it gives away one token for *rds* and one for *res* to the

thread. The token for res is of the type $none$, because the source is in the $none$ state. The token to rds is also none, because the location invariant wants to keep $write$ after the update-operation.

RMW2

Case $v < 0$: such a v cannot be read, because the location invariant states $v \geq 0$.

Case $0 < v < 4$: $\mathsf{Src}(rds, 0, \tau) * \mathsf{Tok}(rds, 1, none) \models False$ (TOO-MANY-TOKENS). Intuitively, the location invariant did not give away any tokens for rds and the thread holds one, which is a contradiction. Therefore, this case cannot occur.

Case $v \geq 4$: The location invariant needs to get back one token of each res and rds. This is exactly what the thread had, therefore it does not keep any tokens. Also, $v - 4$ is still positive, which is needed in order to establish the location invariant after writing the new value.

5.1.2 unlock_shared

Specification, Code and Proof outline

```
void unlock_shared() {
    /////////////////////////////////
    Requires(Rmw(bits_,Q) && Tokens(res, 1, read) && Tokens(rds, 1,
        read));
    /////////////////////////////////
    {Rmw(bits_, Q) * Tok(res, 1, read) * Tok(rds, 1, read)}

    bits_.fetch_add(-4, memory_order_release); // RMW
    {true}
}
```

RMW

Case $lsb(v) = 1$: The location invariant holds $\mathsf{Src}(rds, n, write)$ (i.e. all of rds), and the thread $\mathsf{Tok}(rds, 1, read)$. Such a v cannot be read because $\mathsf{Src}(rds, n, write) * \mathsf{Tok}(rds, 1, read) \models False$ (WRITE-SRC-READ-TOK). Intuitively, this case cannot happen because together they hold more than the full permission, which is a contradiction.

Case $v < 4 \wedge lsb(v) = 0$: The location invariant holds $\mathsf{Src}(res, 0, read)$ and the thread $\mathsf{Tok}(res, 1, read)$. In other words the thread has a token even though the location invariant did not give out any. This is a contradiction and therefore this case cannot occur.

Case $v \geq 4 \wedge lsb(v) = 0$: The counter of missing tokens for both locations goes down by one. The location invariant can be established because it can take the two tokens that it needs from the thread.

5.1.3 try_lock

Specification, Code and Proof outline

```
bool try_lock() {
    //////////////////////////////////////////////////////
    Requires(Rmw(bits_,Q));
    Ensures(boolRes() ? Tokens(res, 1, write) : true);
    //////////////////////////////////////////////////////
    {Rmw(bits_, Q)}

    bool ret;
    int expect;
    expect = 0;
    ret = bits_.compare_exchange_strong(
      expect, 1, memory_order_acq_rel); // RMW
    {ret ⟹ Tok(res, 1, write)}
    return ret;
}
```

RMW

Case $v = 0$: The resources held by the invariant change from $\mathsf{Src}(res, 0, read)$ (which implies $\mathsf{Src}(res, 0, write)$) to $\mathsf{Src}(res, 1, none)$. The thread therefore gets $\mathsf{Tok}(res, 1, write)$. The thread does not get any token to rds, because the corresponding count did not change at the invariant.

Case $v > 0$: The value and the permission at the location invariant are left unchanged.

5.1.4 unlock

Specification, Code and Proof outline

```
void unlock(bool getRead) {
    //////////////////////////////////
    Requires(Rmw(bits_,Q) && (getRead ? Tokens(res, 2, write) && Tokens
        (rds, 1, none)
                    : Tokens(res, 1, write)));
    Ensures(getRead ? Tokens(res, 1, read) && Tokens(rds, 1, read) :
        true);
    //////////////////////////////////
    {Rmw(bits_, Q) * (getRead ? Tok(res, 2, write) * Tok(rds, 1, none) : Tok(res, 1, write))}


    bits_.fetch_add(-1, memory_order_release); // RMW
    {getRead ? Tok(res, 1, read) * Tok(rds, 1, read) : true}
}
```

RMW

Case $lsb(v) = 1 \wedge getRead = false$: The thread gives a $write$-token for res back to the invariant. The invariant then holds $\mathsf{Src}(res, n, write)$ which can be replaced with $\mathsf{Src}(res, n, read)$. No rds-tokens are moved around, but the abstract keyword changes from $write$ to $some$, which is allowed.

Case $lsb(v) = 1 \land getRead = true$: The thread gives a *read*-token for *res* back to the invariant, which then holds $\mathsf{Src}(res, n, some)$ and keeps a *read*-token. Since the thread has a *none*-token for *rds*, and the invariant stars with *write*. Since the invariant only wants to keep *read*, the *none*-token can be exchanged for a *read*-token. The acquire memory order is not needed because *rds* is a ghost-location.

Case $lsb(v) = 0$: The thread holds a *write* token for *res* and the location invariant has *read* permission. Applying the READ-SRC-WRITE-TOK rule shows that this case cannot occur.

5.1.5 unlock_and_lock_shared

Specification, Code and Proof outline

```
void unlock_and_lock_shared() {
    ////////////////////////////////////////////////////
    Requires(Rmw(bits_, Q) && Tokens(res, 1, write));
    Ensures(Tokens(res, 1, read) && Tokens(rds, 1, read));
    ////////////////////////////////////////////////////
    {Rmw(bits,Q) * Tok(res, 1, write)}

    bits_.fetch_add(4, memory_order_acquire); // RMW
    {Rmw(bits,Q) * Tok(res, 2, write) * Tok(rds, 1, none)}
    unlock(true);
    {Rmw(bits,Q) * Tok(res, 1, read) * Tok(rds, 1, read)}
}
```

RMW

Case $lsb(v) = 0$: The thread holds a *write* token for *res* and the location invariant has *read* permission as well. This adds up to more than the full permission which is a contradiction.

Case $lsb(v) = 1$: Increasing the counter by 4 makes the invariant give out one token of both *res* and *rds*. Since the invariant only holds *none* for *res* and wants to keep *write* for *rds* both are *none*-tokens.

5.2 Barrier

In this section we will verify a barrier implementation from the Folly library. In the original, an instance of this barrier is created for a certain number of threads. In our version we use a constant number, but our proof works for any constant that would be allowed as parameter of the original constructor. The `wait`-function is the central element of the barrier. It returns a future-object that is completed after all threads called `wait()`. The `wait`-function consists of two steps. The last thread that executes the first RMW-operation performs a write-operation to a data structure of the barrier. Then the last thread that executes the second RMW-operation also performs a write operation on the same data structure. Figure 5.1 illustrates the principle. This is not an explanation why the two steps are needed, it is just an observation that in the original

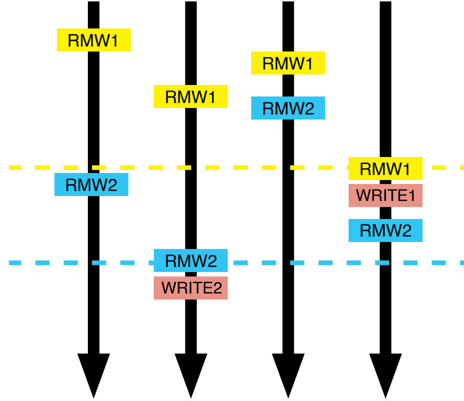

Figure 5.1: Example execution of four threads calling `wait()`.

implementation there are two write operations that would race without proper synchronization. We also verified a different specification, which mainly shows that our logic can be used for complex proofs (see Appendix A.5).

Location invariant

$$
\begin{aligned}
Q(r,v) \quad := \quad & 0 \le r \wedge r \le v \wedge \\
& \big(\mathsf{Src}(g, n-v, write) * \\
& \text{if } v < n \text{ then } \mathsf{Src}(data, 2*r, write) \\
& \text{elseif } (\text{r,v}) = (\text{0,n}) \text{ then } \mathsf{Src}(data, 1, none) \\
& \text{else } \mathsf{Src}(data, 2*r, none)\big)
\end{aligned}
$$

Field definitions

```
const int kReader = 8;
const int size_ = 5;
int g = GhostRef();
int data;
atomic<int> valueAndReaderCount = atomic<int>(Q);
```

Specification, Code and Proof outline

```
void wait() {
    ///////////////////////////////////////////////////////////
    Requires(Rmw(valueAndReaderCount,Q) && Tokens(g,1,none));
    ///////////////////////////////////////////////////////////
    {Rmw(valueAndReaderCount, Q) * Tok(g, 1, none)}

    int prev;
    prev = valueAndReaderCount.fetch_add(kReader + 1,
        memory_order_acquire); // RMW1
    {(prev mod kReader) = size_ - 1 ==> Tok(data, 2, write)
    ∧ (prev mod kReader) < size_ - 1 ==> Tok(data, 2, none)}
    if ((prev%kReader) + 1 == size_) {
        ////////////////////////////
        Assert(Tokens(data,1,write));
        ////////////////////////////
    }
    prev = valueAndReaderCount.fetch_add(-kReader, memory_order_acq_rel
        ); // RMW2
    {prev == (size_ + kReader) ==> Tok(data, 1, write)}
    if (prev == (size_ + kReader)) {
        ////////////////////////////
        Assert(Tokens(data,1,write));
        ////////////////////////////
    }
}
```

5.2.1 Proof

RMW1

Case we read $v \geq n$: This case cannot happen because the location invariant holds $\mathsf{Src}(g, 0, \mathit{write})$ and the thread holds $\mathsf{Tok}(g, 1, \mathit{none})$, which is one token too much.

Case we read $v = n - 1$: The thread sets v to n and increases r by one. We obviously get two tokens for *data*. The keyword of the permission that the invariant holds changes from *write* to *none*. The *write* permission therefore goes to the thread (along with the two tokens). The location invariant takes back the token for g.

Case we read $v < n - 1$: By increasing r by one the thread gets two tokens for *data* and gives up one for *ticket*. The keyword at the location invariant stays *write*, therefore the thread gets *none*.

RMW2

Case we read $r = 0 \wedge v < n$: The source is not missing any token, but the thread has two, which is a contradiction.

Case we read $r = 0 \wedge v = n$: The source is missing only one token, but the thread has two, which is a contradiction.

Case $r = 1 \wedge v = n$: The source and the invariant together have all tokens and therefore the full permission. The invariant keeps only *none*, and gives out

one token. The thread therefore gets one token with the *write* permission.

Case $r > 1$: The thread has to give up two tokens, and with that also all the real permission it holds.

5.3 ARC

We verify a version of the atomic reference counter (ARC) from Rust that we reimplemented in C++. The ARC allows to store some data in the memory, and then give read access to many threads. The value that is stored stays the same as long as some thread has a reference. The count of threads that have a reference is stored in an atomic location. If a thread does not need its reference anymore, it calls the drop function. The drop function then decrements the atomic counter and if the thread was the last one to hold a reference, it will deallocate the memory.

In our verification, holding one reference is represented as $\mathsf{Tok}(g, 1, none) * \mathsf{Tok}(data, 1, read) * \mathsf{Rmw}(count)$, which is (unsurprisingly) exactly the precondition of `read()`. When a new reference is created, the read permission never comes from the location invariant. It always comes from the thread that owns a reference and clones it. Permission only gets to the location invariant at all if a thread calls drop. As soon as the counter reaches 0, the location invariant will give away all the permission it received by previous drop calls to the last thread that has a reference. This last thread will then have two tokens, the one it already had and one that contains the rest of what is missing for the full permission. g is a ghost location that is needed only for verification. In the next section we will see a slightly different proof, where this ghost location is not needed.

Location invariant

$Q(v) := \mathsf{Src}(g, v, write) \wedge (v = 0\ ?\ \mathsf{Src}(data, 2, none) : \mathsf{Src}(data, v, none))$

Field definitions

```
int data;
int g = GhostRef();
atomic<int> count = atomic<int>(Q);
```

5.3.1 Constructor

Specification, Code and Proof Outline

```
Arc(int v)
{
    ////////////////////////////////////////////////////////////
    Requires(RMWAcq(count, Q) && Rel(count, Q) && Uninit(data) && acc(g
        ));
    Ensures(Tokens(data, 1, read) && Tokens(g, 1, read));
    ////////////////////////////////////////////////////////////
    {RMWAcq(count, Q) * Rel(count, Q) * Uninit(data) * acc(g)}
    data = v;
    {RMWAcq(count, Q) * Rel(count, Q) * acc(g) * acc(data) * data = v}
    count.store(1, memory_order_relaxed);
    {Init(count) * RMWAcq(count, Q) * Rel(count, Q) * Tok(g, 1, none) * Tok(data, 1, write) * data = v}

    {Tok(g, 1, none) * Tok(data, 1, write) * data = v}
    {Tok(g, 1, none) * Tok(data, 1, read)) * data = v}
  }
}
```

Write operation

Before the write (`store`) operation the thread has the full access to $data$ and to g. When the 1 is written to the location, the invariant wants $\mathsf{Src}(data, 1, none)$ and $\mathsf{Src}(data, 1, write)$, so the thread can keep $\mathsf{Tok}(data, 1, write)$ and $\mathsf{Tok}(g, 1, none)$. Since no real permission gets transferred to the invariant, the write operation can be relaxed. The full permission to g goes from the thread to the location invariant, but since g is a ghost location, it can be sent with a relaxed write.

5.3.2 `read`

Specification, Code and Proof Outline

```
int read()
{
    ////////////////////////////////////////////
    Requires(Tokens(g, 1, none) && Tokens(data, 1, read) && Rmw(count,Q
        ));
    Ensures(Tokens(g, 1, none) && Tokens(data, 1, read));
    ////////////////////////////////////////////

    {Tok(g, 1, none) * Tok(data, 1, read)}
    return data;
    {Tok(g, 1, none) * Tok(data, 1, read)}
}
```

5.3.3 `clone`

Specification, Code and Proof Outline

```
void clone()
{
    /////////////////////////////////////////////
    Requires(Tokens(data, 1, read) && Tokens(g, 1, none))&& Rmw(count,Q
        ));
    Ensures(Tokens(data, 2, read) && Tokens(g, 2, none)));
    /////////////////////////////////////////////
    {Tok(data, 1, read) * Tok(g, 1, none)}

    count.fetch_add(1, memory_order_relaxed); // RMW
    {Tok(data, 2, read) * Tok(g, 2, none)}
}
```

RMW

Case $v = 0$: Cannot happen because the location invariant is not missing any tokens for g and the thread has one.

Case $v > 0$: Increasing the counter by 1 makes the invariant give away one token of both g and $data$. Since the invariant keeps the write permission for g, the thread gets a none-token for g. Since the thread already has read permission to $data$, it only needs a none-token for $data$ which can be transferred with the relaxed memory order.

5.3.4 `drop`

Specification, Code and Proof Outline

```
void drop()
{
    ////////////////////////////////////////////////
    Requires(Tokens{data, 1, read} && Tokens{g, 1, none} && Rmw{count,Q
        });
    ////////////////////////////////////////////////
    {Tok(data, 1, read) * Tok(g, 1, none)}
    int y;
    y = count.fetch_add(-1, memory_order_release); // RMW
    {y = 1  ⟹  Tok(data, 2, (∇read, read, Σwrite))}
    if (y == 1) {
        {Tok(data, 2, (∇read, read, Σwrite))}
        atomic_thread_fence(memory_order_acquire);
        {Tok(data, 2, write)}
        delete this;
    }
}
```

RMW

Case $v = 0$: Cannot happen because the location invariant is not missing any tokens for g and the thread has one.

Case $v = 1$: The invariant and the thread together hold $\mathsf{Src}(data, 0, write)$. Since the invariant keeps $\mathsf{Src}(data, 2, none)$, the thread can have all the permission and two tokens. But because the memory order is neither *acq* nor *rel_acq*, the permission that the thread gets from the invariant needs is under the down-modality. This permission is exactly what is missing for the thread in order to have *write* access. The location invariant needs one more token for g. The write permission it already had before, so a none-token is enough.

Case $v > 1$: After updating the value, the invariant needs one additional tokens for g and one for *data*. The thread is left with no tokens, and therefore also without permission. Since the memory order is *rel*, the transfer of permission is successful.

5.4 ARC version 2

This proof uses a different location invariant than the previous one. Unlike the other one, it does not need a ghost location. What we achieved with the ghost location we now encode with the amount of tokens. Holding a reference is now represented by $\mathsf{Tok}(data, 3, read) * \mathsf{Rmw}(count)$. The number three could be replaced with anything larger than three. The only thing that matters is that the number is strictly larger than two. This is needed to prove that we can never read zero from the counter if we hold a reference, because the source would be missing two tokens, whereas the thread has more than two.

Location invariant

$Q(v) := v = 0 \; ? \; \mathsf{Src}(data, 2, none) : \mathsf{Src}(data, 3 * v, none)$

5.4.1 Constructor

Specification, Code and Proof Outline

```
Arc(int v)
{
    ////////////////////////////////////////////////////////////
    Requires(RMWAcq(count, Q) && Rel(count, Q) && Uninit(data));
    Ensures(Tokens(data, 2, read) && Rmw(count,Q));
    ////////////////////////////////////////////////////////////
    {RMWAcq(count, Q) * Rel(count, Q) * Uninit(data)}
    data = v;
    {RMWAcq(count, Q) * Rel(count, Q) * Owns(data) * data = v}
    count.store(1, memory_order_relaxed); // write operation
    {Init(count) * RMWAcq(count, Q) * Rel(count, Q) * Tok(data, 3, write) * data = v}
    {U(count, Q) * Tok(data, 3, write) * data = v}
    {U(count, Q) * Tok(data, 3, read)) * data = v}
  }
}
```

Write operation

When the 0 is written to the location, the invariant only wants $\mathsf{Src}(data, 3, none)$, so the thread can keep $\mathsf{Src}(data, 3, write)$. Since no real permission gets transferred to the invariant, the write operation can be relaxed.

5.4.2 `read`

Specification, Code and Proof Outline

```
int read()
{
    ///////////////////////////////////////////////
    Requires(Tokens(data, 3, read) && Rmw(count,Q));
    Ensures(Tokens(data, 3, read));
    ///////////////////////////////////////////////

    {Tok(data, 3, read) * Rmw(count, Q)}
    return data;
    {Tok(data, 3, read)}
}
```

5.4.3 `clone`

Specification, Code and Proof Outline

```
void clone()
{
    ///////////////////////////////////////////////
    Requires(Tokens(data, 3, some) && Rmw(count,Q));
    Ensures(Tokens(data, 3, some) * Tokens(data, 3, some));
    ///////////////////////////////////////////////
    {Tok(data, 3, read)}

    count.fetch_add(1, memory_order_relaxed); // RMW
    {Tok(data, 6, read)}
    {Tok(data, 3, read) * Tok(data, 3, read)}
}
```

RMW

Case $v = 0$: Cannot happen because the location invariant gave out only two tokens and the thread owns three.

Case $v > 0$: Increasing the counter by 1 makes the invariant give away three tokens. Since the invariant holds $\mathsf{Src}(data, 3v, none)$ it can give out only *none*-tokens.

5.4.4 drop

```
void drop()
{
    //////////////////////////////////////////////////
    Requires(\Tokens{data, 3, read} && \Rmw{count,Q});
    //////////////////////////////////////////////////
    {Tok(data, 3, read) ∗ Rmw(count, Q)}
    int y;
    y = count.fetch_add(-1, memory_order_release); // RMW
    {y = 1  ⟹  Tok(data, 2, (∇read, read, Σwrite))}
    if (y == 1) {
        {Tok(data, 2, (∇read, read, Σwrite))}
        atomic_thread_fence(memory_order_acquire);
        {Tok(data, 2, write)}
        delete this;
    }
}
```

RMW

Case $v = 0$: Cannot happen because the location invariant gave out only two tokens and the thread owns three.

Case $v = 1$: The invariant and the thread together hold $\mathsf{Src}(data, 0, write)$. The thread can have two tokens, and since the invariant keeps $\mathsf{Src}(data, 2, none)$, the thread can have all the permission. But because the read mode is neither *acq* nor *rel_acq*, the permission that the thread gets from the invariant needs to be fenced. This permission is exactly what is missing for the thread in order to have *write* access.

Case $v > 1$: After updating the value, the invariant needs three additional tokens. The thread is left with no tokens, and therefore also without permission. Since the memory order is *rel*, the transfer of permission is successful.

5.5 ARC comparison to FSL++

5.5.1 FSL++ proof of drop function

$$Q(c) \quad := \quad c = 0 \; ? \; \boxed{\lambda : 0^-} * \boxed{\delta : 0^-} : \exists f \in \mathbb{Q} \cap [0,1] \; . \; data \overset{f}{\mapsto} _ * \boxed{\lambda : (c - 1 + f)^-} * \boxed{\delta : (1 - f)^-}$$

The ghost locations are governed by the SFC monoid defined in [4].

```
void drop()
{
    {Rmw(count, Q) ∗ ∃q ∈ ℚ ∩ (0,1].data ↦^f q ∗ [λ : (1 − q)^+] ∗ [δ : (q)^+]}
    int y;
    y = count.fetch_add(-1, memory_order_release);
    {y = 1  ⟹  data ↦^q v ∗ ▽data ↦^{1−q} _}
    if (y == 1) {
        {data ↦^q v ∗ ▽data ↦^{1−q} v}
        atomic_thread_fence(true, memory_order_acquire);
        {a.data ↦^1 v}
        delete this;
    }
}
```

Transition invariant for the FSL++ proof

Transition invariants were introduced in [7] and the one we present here is taken from [7]. A transition invariant defines how an existentially quantified variable in a location invariant should be instantiated when the new value is written to the atomic location.

$$t(c, f, c', P) = \begin{cases} f & \text{if } c' = c + 1 \\ f + q & \text{if } c' = c - 1 > 0 \text{ and } P \models \boxed{\delta : q^+} \text{ for } q > 0 \\ 0 & \text{if } c' = c - 1 = 0 \\ \textit{undefined} & \text{otherwise} \end{cases}$$

Where c is the old value at the location and f the old value of f. c' is the value that is being written to the location and P is the current permission of the thread.

Observations

The location invariant for the FSL++ proof seems much harder to understand than the one for the proof we gave above. Another difference is that FSL++ does not define how to conduct the proof. For example with this invariant there are infinite options on how to instantiate the existential. In order to automate the FSL++ proof one would at least need the transition invariant for the existential, which further adds to the size and complexity of the specification that is needed in order to verify the program. In our system, the specification is smaller, simpler and the proof can be fully automated.

Chapter 6

Syntax for C++ input programs

We implemented a tool that verifies C++ programs. This chapter shows the subset of C++ that our tool supports.

```
         Program  ::=  #include "weak-memory-frontend.h"
                       Class
           Class  ::=  class c { public: Constructor [ Decl ]*}
     Constructor  ::=  c ([ Type p ]*) {
                         Precondition
                         Postcondition
                         [ Stmt ]*
                       }
            Decl  ::=  Method | TopLevelVarDecl
 TopLevelVarDecl  ::=  int v = 0; | atomic<int> v  = atomic<int>(LocInv); |
                    |  int v = GhostRef() | const int v = n;
            Type  ::=  int | bool
          Method  ::=  (Type | void) f ([ Type p ]*) {
                         Precondition
                         Postcondition
                         [ Stmt ]*
                         [return v;]?
                       }
            Stmt  ::=  if ( BExpr ){[ Stmt ]*} else {[ Stmt ]*}
                    |  v = Expr; | f([ e ]*); | v = f([ e ]*); | LocalVarDecl
                    |  v = Expr; | v1 = v2.load(Sync);
                    |  v.store(e, Sync); | Rmw;
                    |  atomic_thread_fence(memory_order_acquire);
                    |  atomic_thread_fence(memory_order_release);
                    |  delete this; | AssertionStmt
```

$$
\begin{array}{rcl}
\textit{LocalVarDecl} & ::= & \textit{Type } v \texttt{ ;} \\
\textit{Sync} & ::= & \texttt{memory_order_relaxed} \mid \texttt{memory_order_acquire} \\
 & \mid & \texttt{memory_order_release} \mid \texttt{memory_order_rel_acq} \\
\textit{Rmw} & ::= & v = l\texttt{.compare_exchange_strong}(n_1, n_2, \textit{sync}) \\
 & \mid & v = l\texttt{.fetch_add}(n, \textit{sync}) \\
\textit{Expr} & ::= & v \mid \textit{BExpr} \mid \textit{IExpr} \\
\textit{BExpr} & ::= & \texttt{true} \mid \texttt{false} \mid (b_1 \texttt{ \&\& } b_2) \mid (b_1 \texttt{ || } b_2) \mid (e_1 == e_2) \\
 & \mid & (e_1 \texttt{ != } e_2) \\
\textit{IExpr} & ::= & -n \mid n_1 \texttt{*} n_2 \mid n_1 + n_2 \mid n_1 - n_2 \mid n_1 \texttt{ \% } n_2 \mid n_1 \texttt{ / } n_2
\end{array}
$$

Where $\{v, v_1, v_2, c, f, p, t\}$ are identifiers, $\{e, e_1, e_2\} \subseteq$ *Expr*, $\{n, n_1, n_2\} \subseteq$ *IExpr* and $\{b, b_1, b_2\} \subseteq$ *BExpr*. In this work we often use `rlx` instead of `memory_order_relaxed` and also `rel`, `acq` and `rel_acq` instead of the corresponding long versions. Furthermore, we sometimes write `fence` instead of `atomic_thread_fence`.

The syntax for annotations in C++:

$$
\begin{array}{rcl}
\textit{LocInv} & ::= & \texttt{Invariant } Q \texttt{ = [this] (int } v\texttt{) -> bool } \{ \\
 & & \quad \texttt{return } \textit{Assertion}\texttt{;} \\
 & & \} \\
\textit{Precondition} & ::= & \texttt{Requires}(A)\texttt{;} \\
\textit{Postcondition} & ::= & \texttt{Ensures}(A)\texttt{;} \\
\textit{AssertionStmt} & ::= & \texttt{Assert}(A)\texttt{;} \\
\textit{Assertion} & ::= & A_1 \texttt{ \&\& } A_2 \mid \textit{BExpr} \texttt{ ? } A_1 \texttt{:} A_2 \mid BExpr \\
 & \mid & \textit{OnlyAtLocInv} \mid \textit{OnlyAtThread} \\
\textit{OnlyAtLocInv} & ::= & \mathsf{Src}(loc, n, Perm) \\
\textit{OnlyAtThread} & ::= & \mathsf{Tok}(loc, n, Perm) \mid \mathsf{Rmw}(loc) \mid \mathsf{Uninit}(loc) \\
 & \mid & \mathsf{Rel}(loc) \\
\textit{Perm} & ::= & \texttt{write} \mid \texttt{read} \mid \texttt{none}
\end{array}
$$

where $\{A, A_1, A_2\} \subseteq$ *Assertion* and Q and v are Identifiers. We additionally allow `boolRes()` and `intRes()` in the pre- and postconditions, so that we can reason about the return value even before the method body starts.

Chapter 7

C++ to Viper encoding

This chapter shows how C++ code with specification can be encoded into Viper. First, we briefly explain the encoding of ghost locations and modalities. Second, look at how tokens are encoded. Third, we will tackle real C++ code.

7.1 Ghost locations and modalities

The encoding of ghost locations and modalities is the same as in [10]. For this reason we will only give a brief overview here. C++ locations are encoded as **`Ref`**s in Viper. The function `is_ghost(r)` returns **`true`** if and only if `r` is a Viper reference that represents a ghost location. Viper references that do *not* originate from a ghost location are distributed over three heaps, called down-heap, real heap and down-heap. One real (non-ghost) memory location in the original program is represented by three **`Ref`**s in Viper, each belonging to a different one of the three heaps. This is because to a real memory location, we can have three different types of permission depending on whether the permission has 1. the down-modality, 2. no modality or 3. the up-modality. Permission with the down-modality is on the down-heap, permission without modality is on the real-heap and permission with the up-modality is on the up-heap. To which heap a reference `r` points can be checked using the `heap(r)`-function, which returns `0` if it points to the real heap, and `-1` or `1` if it points to the down- or the up-heap, respectively. From a reference `r` on the real heap, the corresponding references on the up- and the down-heap can be retrieved using the `up(r)` and `down(r)`, respectively.

During the verification of RMW-operations, the verifier is in an intermediate state, where so-called temp-heaps are additionally used. Using the `temp(r)`-function, a temporary heap location can be retrieved for any non-ghost Viper-**`Ref`**. The full Viper domain that is used for encoding the different heaps can be found in Appendix A.4.

7.2 Encoding Tokens

In the following, we first describe how we encode the ownership of a certain number of tokens and then the encoding of the permission the tokens give to the owner.

7.2.1 Encoding the Token Counting

This section shows how we encode the counting of tokens. The number of tokens is the sum of all tokens, regardless of the modalities. One non-atomic location is modeled as several `Refs` in Viper, one for each modality and an additional `temp`-heap. The token count, on the other hand, should be combined for all of these `Refs`. We provide a function `tokCountRef(r:Ref)` which returns a ghost reference used to model the counting. Tokens are then modeled as access to the `posTok`-predicate on this , and missing tokens (at the source) as access to the `negTok`-predicate. These kind of predicates can only be held together with the source. We model owning the source as having the full access to the `ownsSource` field. Partial access is never used on this field and also the actual value at the field is irrelevant. An individual assertion that expresses ownership of n positive tokens to the location x would then be encoded as `acc(posTok(tokCountRef(x)),n/1)`. The full Viper definitions for the token counting are given on figure 7.1.

7.2.2 Encoding the Permission Sum Associated with the Tokens

When encoding tokens in Viper, we use the macro `tokens(loc,n,q)`, defined on figure 7.2. Unlike in the specification, the `q` used here is a concrete fraction, and not an abstract keyword. For the keywords *none* and *write*, the transition from the keywords to fractions is trivial. *none* translates directly to `0/1`. Viper even supports the keyword `none` that can be used instead of `0/1`. *write* also exists in Viper, and corresponds to the fraction `1/1`. The *read*-keyword is more complicated, as it does not correspond to a concrete fraction, but can be instantiated with any fraction in $(0, 1]$. When the keyword *read* appears, we often define a variable with the associated permission amount, and assume that the variable is nonzero.

7.2.3 Encoding the Relation between the Token Count and the Permission

If x is a non-atomic location in the original C++ program, we should always be able to assert `isValidLoc(x)` (see figure 7.2) before and after the encoding of a C++ statement. The helper macro `permSumOverHeaps` sums up the permission to a location over all heaps (see also definition of parallel heaps). `isValidLoc` makes sure that someone who does not hold the source and does not have any token, gets no real permission. Someone who does not hold the

```
// tokenCounting.vpr

import "parallelHeaps.vpr"
field ownsSource: Int

define tokCount(r) perm(posTok(tokCountRef(loc)))
define missingTokCount(r) perm(negTok(tokCountRef(loc)))

predicate posTok(r:Ref)
predicate negTok(r:Ref)

domain TokCountRef {
    function tokCountRef(r:Ref): Ref
    function tokCountRef_inv(r:Ref): Ref
    axiom inv {
        (forall r: Ref :: { tokCountRef(r) } (heap(r)==0 || is_ghost(
            r)) ==> tokCountRef_inv(tokCountRef(r)) == r)
    }
    axiom always_ghost {
        (forall r: Ref :: { tokCountRef(r) } is_ghost(tokCountRef(r))
            )
    }
    axiom always_on_separate_heap {
        (forall r: Ref :: { tokCountRef(r) } heap(tokCountRef(r)) ==
            2)
    }
    axiom up_has_same_counter {
           (forall r: Ref :: { up_inv(r) } (!is_ghost(r) && heap(r)
               == 1 )==> tokCountRef(up_inv(r))==tokCountRef(r))
    }
    axiom down_has_same_counter {
        (forall r: Ref :: { down_inv(r) } (!is_ghost(r) && heap(r) ==
             -1)==> tokCountRef(down_inv(r))==tokCountRef(r))
    }
    axiom temp_has_same_counter_1 {
        (forall r: Ref :: { temp_inv(r) } (!is_ghost(r) && heap(r) ==
             -2)==> tokCountRef(temp_inv(r))==tokCountRef(r))
    }
    axiom temp_has_same_counter_2 {
        (forall r: Ref :: { temp_inv(r) } (!is_ghost(r) && heap(r) ==
             -3)==> tokCountRef(temp_inv(r))==tokCountRef(r))
    }
    axiom temp_has_same_counter_3 {
        (forall r: Ref :: { temp_inv(r) } (!is_ghost(r) && heap(r) ==
             -4)==> tokCountRef(temp_inv(r))==tokCountRef(r))
    }
}
// end of file
```

Figure 7.1: Viper encoding of token counting.

```
// tokens.vpr

define tokens(loc,count,sum) acc(posTok(tokCountRef(loc)), count/1)
    && acc(loc.val, sum)

define source(loc,count,sum) acc(negTok(tokCountRef(loc)), count/1)
    && acc(loc.val, sum) && acc(tokCountRef(loc).ownsSource)

define permSumOverHeaps(loc) is_ghost(loc) ? perm(loc.val) : perm(
    down(loc).val)  + perm(loc.val) + perm(temp(loc).val) + perm(up(
    loc).val) + perm(temp(up(loc)).val) + perm(temp(down(loc)).val)

define isValidLoc(loc)
    permSumOverHeaps(loc) <= 1/1
    && (perm(tokCountRef(loc).ownsSource) == 0/1 ==>
        ((perm(posTok(tokCountRef(loc)))==0/1 ==> permSumOverHeaps(
            loc)==0/1))
        && perm(negTok(tokCountRef(loc)))==0/1)
    && (perm(tokCountRef(loc).ownsSource) == 1/1 ==>
        ((perm(posTok(tokCountRef(loc)))==perm(negTok(tokCountRef(loc
            )))==>permSumOverHeaps(loc)==1/1)
        && perm(posTok(tokCountRef(loc)))<=perm(negTok(tokCountRef(
            loc)))))
// end of file
```

Figure 7.2: Viper macros for token-based permissions.

source cannot hold any negative tokens. If someone owns the source and the same amount of negative and positive tokens, it means that they did not give away any tokens and therefore have the full (i.e. *write*) permission.

7.2.4 Inhaling Tokens

For inhaling tokens we basically inhale the `tokens` macro, which takes fractional permissions as third argument. *none* and *write* are already a exact permission amount, whereas *read* is not. For *read*, we inhale an unspecified, but nonzero amount of permission. See figure 7.3.

7.2.5 Exhaling Tokens

For exhaling tokens we use the macro `exhaleTokens` (see Figure 7.4). If the thread only exhales none-tokens, it does not matter whether the location is a ghost location. We simply exhale `tokens(loc,count,0/1)`, which exhales `count` instances of the `posTok`-predicate. If a thread exhales *read* permission to a ghost location, we first assert it has nonzero permission to `loc.val`. Then we check whether the thread will have any tokens left. If not, we exhale all the permission the thread has, because without any tokens it cannot keep any permission. If yes, we exhale half of the permission. If the location is *not* a ghost location, it is more complicated. The read-permission can come either from the real heap or from the up-heap, so we assert that there is permission on one of them. Then the thread again tries to exhale only part of its permission if possible. It cannot keep any of the permission in question if either it does not

```
// inhaleTokens.vpr

define inhaleTokens(loc,count,sum) {
    if (sum == NONE) {
        inhale tokens(loc,count,none)
    }
    elseif (sum == READ) {
        var q:Perm
        q := havocedPerm()
        inhale q > 0/1
        inhale tokens(loc,count,q)
    }
    elseif (sum == WRITE) {
        inhale tokens(loc,count,write)
    }
    else {
        assert false
    }
}

// end of file
```

Figure 7.3: Macro for inhaling tokens.

keep any tokens or if it keeps only one token, but needs that token for permission it has on the down-heap. If the thread has permission on both the real and the up-heap and can keep some of it, it keeps only the permission on the up-heap, because this is strictly more useful (as the exact permission amounts do not matter anyway as long as they are not 0 or 1).

If the thread wants to exhale the *write* permission for a ghost location, we can simply exhale `tokens(loc,count,1/1)`. If the location is not a ghost location, the write permission can come partly from the real heap and partly from the up-heap. We first assert that their sum is write and then exhale both.

7.3 Encoding a C++ program

As described in chapter 6, our input C++ programs consist of one class and their member fields and methods. Figure 7.5 shows how a concrete C++ program is translated to Viper. The produced Viper file consists of an import statement and the encodings of the class members. For the class members, Figure 7.5 shows only the encoding of fields. Location invariants and methods are explained in separate sections.

The encoding that we present here uses Viper macros extensively. Every generated Viper program includes `includes.vpr` (see appendix A.3), which again imports many files that stay the same for every C++ program that is encoded. This allows us to clearly separate parts of the encoding that depend on the input C++ program and parts that do not. The files that are included do not only contain macros, but also other Viper definitions (e.g. domains).

```
// exhaleTokens.vpr

define exhaleTokens(loc,count,sum) {
    if (sum == NONE) {
        exhale tokens(loc,count,0/1)
    }
    elseif (sum == READ) {
        if (is_ghost(loc)) {
            assert perm(loc.val) > 0/1
            if (perm(posTok(tokCountRef(loc))) == count / 1) {
                exhale tokens(loc, count, perm(loc.val))
            }
            else {
                exhale tokens(loc, count, perm(loc.val) / 2)
            }
        }
        else {
            assert perm(loc.val) + perm(up(loc).val) > 0/1
// exhaling all tokens from up and real
            if (perm(posTok(tokCountRef(loc))) == count / 1 || (perm(
                posTok(tokCountRef(loc))) == (count / 1) + 1/1 &&
                perm(down(loc).val) > 0/1)) {
                exhale acc(posTok(tokCountRef(loc)), count/1) && acc(
                    loc.val, perm(loc.val)) && acc(up(loc).val, perm(
                    up(loc).val))
            }
// leaving at least one token on up or real. if possible, we only
                leave permission on "up"
            else {
                if (perm(loc.val) > 0/1 && perm(up(loc).val) > 0/1) {
                    exhale tokens(loc,count,perm(loc.val))
                }
                elseif (perm(loc.val) > 0/1 ) {
                    exhale tokens(loc,count,perm(loc.val)/2)
                }
                elseif (perm(up(loc).val) > 0/1) {
                    exhale tokens(up(loc),count,perm(up(loc).val)/2)
                }
            }
        }
    }
    elseif (sum == WRITE) {
        if (is_ghost(loc)) {
            exhale tokens(loc,count,1/1)
        }
        else {
            assert perm(loc.val) + perm(up(loc).val) == 1/1
            exhale acc(posTok(tokCountRef(loc)), count/1) && acc(loc.
                val, perm(loc.val)) && acc(up(loc).val, perm(up(loc).
                val))
        }
    }
}

// end of file
```

Figure 7.4: Macro for exhaling tokens.

```
⟦[ include ]* using namespace std; Class⟧ ⇝
import "include/includes.vpr"
[⟦TopLevelDecl⟧]*

⟦(int v;)TopLevelDecl⟧ ⇝
function v():Ref
   ensures !is_ghost(result) && heap(result)==0

⟦(atomic<int> v = atomic<int>();)TopLevelDecl⟧ ⇝
function v():Ref
   ensures !is_ghost(result) && heap(result)==0

⟦(int v = GhostRef();)TopLevelDecl⟧ ⇝
function v():Ref
   ensures is_ghost(result) && heap(result)==0

⟦(const int v = n;)TopLevelDecl⟧ ⇝
define v n
```

Figure 7.5: Viper encoding of a C++ program and fields of a class.

7.4 Field declarations

Fields of the class (except constants) are encoded as **`Ref`**s. For each field we generate a Viper function (with the same name as the original C++ field) that returns the **`Ref`**. The postcondition ensures that `is_ghost(`**`result`**`)` returns true for a ghost location and false otherwise. `heap(res)==0` ensures that the reference is on the real heap. A field can be atomic or non-atomic, but this difference is not visible when looking at the definition of the function that returns the **`Ref`**. The difference between atomic and non-atomic locations will be visible in our encoding of the usage of the locations (see sections on non-atomics and atomics).

We also support definitions of constant integers as fields of the class. This is not crucial to our encoding. It just increases convenience and readability. For each such constant, a Viper macro is defined which simply replaces occurrences of the variable with the integer value.

7.5 Non-Atomics

This section describes how non-atomic fields are used. The value of a non-atomic location `loc` is accessed through `loc.val`. For this, the appropriate permission has to be held. Most of the reasoning in our proofs is about permission to `loc.val` for some non-atomic `loc`. Figure 7.6 shows how read and write operations to a non-atomic field are encoded. If enough permission on the real heap is held (i.e. $> 0/1$ in case we want to read, and $1/1$ in case we want to write) *loc.val*

can be accessed directly. If we do *not* have enough permission on the real heap, we try to move some from the up-heap to the real heap. With our set of proof rules (unlike in the RSL logics!), this is allowed. The permission has to be moved from the up-heap to the real heap, because as long as it is under the modality, it cannot be used for non-atomic reads or writes. For an non-atomic read, if the thread has a read token, the read operation can be performed without any changes to the state. If this is not the case, the read operation is still possible given that the thread has at least one read-token with the up-modality (`perm(up(loc).val)>0/1 && !is_ghost(loc)`). If the thread has exactly one token with the up-modality (`perm(posTok(tokCountRef(loc)))==1/1`), it has to remove the modality of this token and ends up with $\mathsf{Tok}(loc, 1, read)$. Similarly, if the thread has $\mathsf{Tok}(loc, 2, (\triangledown read, \triangle read))$ it also cannot keep a read-token with the up-modality and ends up with $\mathsf{Tok}(loc, 2, (\triangledown read, read))$. If the thread has enough tokens to have permission both on the up-heap and on the real-heap, we move half of the permission from the up-heap to the down-heap. It does not matter that it is a half, it could be any nonzero fraction. If the thread has neither permission on the up-heap nor on the real heap, we make sure the verification fails by asserting **`false`**.

7.6 Atomics

The encoding of atomic writes (`loc.store()`) is shown on listing 7.7, where Q is the location invariant of the location. The definition of the macro `ExhaleInv` is explained in the section on the encoding of location invariants (Section 7.13). An encoding of atomic reads is not shown, because we only allow atomic reads in form of read-modify-writes 7.12, which are treated later.

7.7 Local variables

With local variables we mean variables that are declared in the scope of a method and not as a field of a class. In our model, local variables are not on the heap and they are not shared between threads. Local variables are therefore simply encoded as local Viper variables and no permission is involved.

7.8 Methods

Figure 7.8 shows how methods and method calls are encoded. The macros `exhale_precondition_`f and `exhale_postcondition_`f are not used for the verification of the method itself. They are only used at places where the method f is called.

v = loc; ⇝ `nonAtomicRead(loc,`v`)`
loc = e; ⇝ `nonAtomicWrite(loc,`$[\![e]\!]$`)`

```
// nonAtomics.vpr

define nonAtomicRead(loc, target) {
    if (perm(loc.val) > 0/1) {
        target := loc.val
    }
    elseif (perm(up(loc).val) > 0/1 && !is_ghost(loc)) {
        if ((perm(posTok(tokCountRef(loc))) == 1/1) || (perm(posTok(
            tokCountRef(loc))) == 2/1 && perm(down(loc).val) > 0/1))
            {
// move all from up to real
            inhale acc(loc.val, perm(up(loc).val))
            exhale acc(up(loc).val, perm(up(loc).val))
        }
        else {
// move part from up to real
            inhale acc(loc.val, perm(up(loc).val)/2)
            exhale acc(up(loc).val, perm(up(loc).val)/2)
        }
        target := loc.val
    }
    else {
        assert false
    }
}

define nonAtomicWrite(loc, v) {
    if (perm(loc.val) == 1/1) {
        loc.val := v
    }
    elseif (perm(up(loc).val) + perm(loc.val) == 1/1 && !is_ghost(loc
        )) {
// move all from up to real
        inhale acc(loc.val, perm(up(loc).val))
        exhale acc(up(loc).val, perm(up(loc).val))
        loc.val := v
    }
    else {
        assert false
    }
}
```

Figure 7.6: Viper encoding of non-atomic memory accesses.

$[\![$`v.store(`e`,`$sync$`);`$]\!]$ ⇝ `store(`v, $[\![e]\!]$, $[\![sync]\!]$`)`

```
// in file includes/BasicDefinitions.vpr
define store(v, newVal, sync) {
    assert SomeRel(v)
    ExhaleInv(newVal,sync)
    inhale Init(v)
}
```

Figure 7.7: Viper encoding of atomic writes.

```
T f(params) {
  [ Stmt ]*
  return ret;
}
⇝
method f(this:Ref,⟦params⟧)
  returns (res:⟦T⟧)
{
  ⟦A⟧_inhale
  [⟦Stmt⟧]*
  res := ret
  ⟦B⟧_exhale
}
define exhale_precondition_f(this, ⟦params⟧) {
  ⟦A⟧_exhale
}
define inhale_postcondition_f(this, ⟦params⟧, res) {
  ⟦B⟧_inhale
}

void f(params) { ... }
⇝ ... (nearly identical as case with return value)

⟦v = f(params)⟧
⇝
exhale_precondition(this,⟦params⟧)
v = havoced⟦T⟧()
inhale_precondition(this,⟦params⟧,v)

⟦f(params)⟧
⇝
exhale_precondition(this,⟦params⟧)
inhale_precondition(this,⟦params⟧)
```

Figure 7.8: Encoding of methods and method calls.

7.9 Assertions

The encoding of assert statements:

$$
\begin{array}{rcl}
[\![\texttt{Assert(}A\texttt{);}]\!] & \leadsto & \texttt{assert(}[\![A]\!]\texttt{)} \\
[\![A_1 \texttt{ \&\& } A_2]\!] & \leadsto & [\![A_1]\!] \texttt{ \&\& } [\![A_2]\!] \\
[\![\mathit{BExpr} \texttt{ ? } A_1 \texttt{ : } A_2]\!] & \leadsto & [\![\mathit{BExpr}]\!] \texttt{ ? } [\![A_1]\!] \texttt{ : } [\![A_2]\!] \\
[\![\texttt{Tokens(}\mathit{loc}, n, \mathit{ThreadPerm})]\!] & \leadsto & \texttt{tokens(}\mathit{loc}\texttt{(this)}, n, [\![\mathit{perm}]\!]\texttt{)} \\
[\![\texttt{Rmw}(\mathit{loc})]\!] & \leadsto & \texttt{rmw(}\mathit{loc}\texttt{(this))} \\
[\![\texttt{Uninit}(\mathit{loc})]\!] & \leadsto & \texttt{uninit(}\mathit{loc}\texttt{(this))} \\
[\![\texttt{none}]\!] & \leadsto & \texttt{none} \\
[\![\texttt{read}]\!] & \leadsto & \texttt{wildcard} \\
[\![\texttt{write}]\!] & \leadsto & \texttt{write}
\end{array}
$$

7.10 Inhales and Exhales

For inhales:

$$
\begin{array}{rcl}
[\![\mathit{BExpr} \texttt{ ? } A_1 \texttt{: } A_2]\!] & \leadsto & \texttt{if(}[\![\mathit{BExpr}]\!]\texttt{)\{}[\![A_1]\!]\texttt{\}else\{}[\![A_2]\!]\texttt{\}} \\
[\![\mathit{BExpr}]\!] & \leadsto & \texttt{inhale } [\![\mathit{BExpr}]\!] \\
[\![A_1 \texttt{ \&\& } A_2]\!] & \leadsto & [\![A_1]\!] \\
 & & [\![A_2]\!] \\
[\![\texttt{Rmw}(\mathit{loc})]\!] & \leadsto & \texttt{inhale rmw(}\mathit{loc}\texttt{(this))} \\
[\![\texttt{Uninit}(\mathit{loc})]\!] & \leadsto & \texttt{inhale uninit(}\mathit{loc}\texttt{(this))} \\
[\![\texttt{Tokens(loc,n,}\mathit{perm})]\!] & \leadsto & \texttt{inhaleTokens(loc,n,}[\![\mathit{perm}]\!]\texttt{)} \\
[\![\texttt{write}]\!] & \leadsto & \texttt{WRITE} \\
[\![\texttt{read}]\!] & \leadsto & \texttt{READ} \\
[\![\texttt{none}]\!] & \leadsto & \texttt{NONE}
\end{array}
$$

Note that `inhale` is a built-in Viper keyword and `inhaleTokens` is a macro that we defined.

For exhales the encoding is very similar as for inhales. The only differences are that `inhale` is replaced with `exhale` and `inhaleTokens` is replaced with `exhaleTokens`.

7.11 Fences

Acquire fences are encoded the same way as in [10], but release fences are different. Acquire fences just move all the permission that is held on the down-heap to the real heap. Unlike in the previous work, our release fences do not need any annotation. All the permission that is held on the real heap is moved to the up-heap instead of only what was specified in the annotation. The idea behind the Viper implementation of this transfer is the same as for the acquire fences (see Appendix A.1 for the Viper code).

7.12 Read-Modify-Write Operations

Figure 7.9 shows how we encode a RMW operation with *sync* $\in$ {`rlx`, `rel`, `acq`, `rel_acq`}. Note that the described encoding only works if the source permission is at the location invariant before and after the RMW operation.

7.13 Location Invariants

Figure 7.10 shows the encoding of location invariants. Each invariant that is defined gets a fresh index. When we write $index(Q)$ we mean this index. At the Viper level only the index is used and not the identifier Q. We want to be able to use the location invariant in two ways. We want to be able to inhale it and exhale it. For these usages we generate the macros `InhaleInv`$index(Q)$ and `ExhaleInv`$index(Q)$.

The macro on figure 7.11 are generated depending on how many location invariants there are in the program. It connects the locations with the corresponding invariants. If the value at a location is updated, the macro makes sure that the correct location invariant is inhaled and exhaled (i.e. the one for which the thread holds $\mathsf{Rmw}(loc, Q)$).

7.13.1 Inhaling the source as part of the location invariant

See figure 7.12. The most interesting case is when the keyword is *none*. With *none* we can only be sure to have some permission if we have all the tokens. If we do not have all tokens (i.e. `perm(negTok(.))!=perm(posTok(.))`, we make sure that the inhaled permission is zero. If we have all the tokens, we still don't know for sure how much of the full permission was contributed by the thread and how much by the invariant, so we just don't constrain the permission amount. Inhaling `isValidLoc` later on will add the constraint that they add up to *write*. Note that we inhale the actual permission to the temp heap and not to the real heap.

7.13.2 Exhaling the source as part of the location invariant

Exhaling the location invariant usually happens as part of a RMW operation, i.e. after inhaling it and modifying the value at the location. If we (in the following explanation) refer to what the invariant or the thread had *before*, we mean what it had before the whole RMW operation.

See appendix A.2 for the definition of the Viper macro `exhaleSource`. First we make sure to split off positive tokens if the thread will get more positive tokens than it had before (using `produceMissingTokens`). If this is the case we then exhale all the negative tokens and no positive ones. If the invariant wants less negative tokens than it had before, it has to inhale the difference in form of positive tokens. E.g. let us say it had 3 missing tokens and now it only

$[\![$v$_{old}$ = loc.fetch_add(x, $sync$};$]\!]$
$\rightsquigarrow$ fetch_and_add(loc, x, v_{old}, $[\![sync]\!]$)

$[\![$v = loc.compare_exchange_strong(v_{old}, v_{new}, $sync$};$]\!]$
$\rightsquigarrow$ compare_exchange_strong(loc, v_{old}, v_{new}, v, $[\![sync]\!]$)

```
// RMWs.vpr

define Rmw(l, QIndex) Init(l) && RMWAcq(l,QIndex) && SomeRel(l) && l.
    rel == 0

define compare_exchange_strong(loc, expect, newVal, success, sync) {
    assert Init(loc) && SomeRMWAcq(loc) && SomeRel(loc)
    var vRet:Int
    vRet := havocedInt()
    InhaleInv(loc, vRet)

    var vNew:Int := vRet
    if (vRet == expect) {
        vNew := newVal
        success := true
    }
    else {
        success := false
    }
    ExhaleInv(loc,vNew,sync)
}
define fetch_add_discard(loc, x, sync) {
    assert Init(loc) && SomeRMWAcq(loc) && SomeRel(loc)
    var vRet:Int
    vRet := havocedInt()
    InhaleInv(loc,vRet)
    var vNew:Int := vRet + x
    ExhaleInv(loc,vNew,sync)
}
define fetch_add(loc, x, vRet, sync) {
    assert Init(loc) && SomeRMWAcq(loc) && SomeRel(loc)
    vRet := havocedInt()
    InhaleInv(loc, vRet)
    var vNew:Int := vRet + x
    ExhaleInv(loc,vNew,sync)
}
// end of file
```

Figure 7.9: Viper encoding of RMW-operations.

```
Invariant Q = [this] (int v) -> bool {
  return Assertion;
}
```

⇝

```
define InhaleInvindex(Q)(v,sync) {
  ⟦Assertion⟧_inhaleLocInv
  for all non-atomic locations "loc" that appear in Q
  inhale isValidLoc(loc)
}
define ExhaleInvindex(Q)(v,sync) {
  ⟦Assertion⟧_exhaleLocInv
  for all non-atomic locations "loc" that appear in Q

  if(!is_ghost(loc(this)))
    inhale acc(downOrReal(loc(this),sync).val, perm(temp(loc(this)).val
        ))
    exhale acc(temp(loc(this)).val, perm(temp(loc(this)).val))
  }

}
```

$[\![Assertion]\!]_{inhaleLocInv}$ is similar to the encoding of other inhales. The only addition is:

$[\![$`Source(loc,n,`$perm$`)`$]\!]$ ⇝ `inhaleSource(loc,n,`$[\![perm]\!]$`, sync)`

$[\![Assertion]\!]_{exhaleLocInv}$ is similar, with one difference:

$[\![$`Source(loc,n,`$perm$`)`$]\!]$ ⇝ `exhaleSource(loc,n,`$[\![perm]\!]$`, sync)`

Figure 7.10: Encoding of location invariants.

```
define InhaleInv(loc, v) {
  if (perm(AcqConjunct(loc,0)) > 0/1) {
    InhaleInv0(v)
  }
  ...
  elseif (perm(AcqConjunct(loc,n)) > 0/1) {
    InhaleInvn(v)
  }
  else {
    assert false
  }
}
define ExhaleInv(loc, v, sync) {
  if (loc.rel==0) {
    ExhaleInv0(v, sync)
  }
  ...
  elseif (loc.rel==n) {
    ExhaleInvn(v, sync)
  }
  else {
    assert false
  }
}
```

Figure 7.11: Dispatching of location invariants.

```
// inhaleSource.vpr

define inhaleSource(loc,count,sum) {
    inhale acc(negTok(tokCountRef(loc)), count/1)
    inhale acc(tokCountRef(loc).ownsSource)
    var havocedP:Perm
    havocedP := havocedPerm()
    if (sum == READ) {
        assume havocedP > 0/1
    }
    elseif (sum == WRITE) {
        assume havocedP == 1/1
    }
    elseif (sum == NONE) {
        if (perm(negTok(tokCountRef(loc))) != perm(posTok(tokCountRef
            (loc)))) {
            assume havocedP == 0/1
        }
    }
    else {
        assert false
    }
    inhale acc(temp(loc).val, havocedP)
}
// end of file
```

Figure 7.12: Definition of the Viper macro `inhaleSource`.

wants 1 missing. Exhaling 2 positive tokens along with all negative tokens will give it the 1 missing it wanted. (Note that tokens and missing tokens cancel out each other.)

As a second step we check what kind of permission the invariant needs. If it had it already before, we can just take if from the temp heap (because this is where the invariant was inhaled to). Notice that in the case of *read* we take less than what is there, because potentially the thread could use *read* as well. If the invariant needs more permission than it had before, it has to come from the thread. If the memory order is not *rel*, the resource needs to be fenced, and therefore come from the up-heap.

If the thread does not keep any tokens for a location, it cannot keep any of the permission it had to the location. If the memory order is *rel*, it can give everything from the real and the up-heap to the location invariant. This permission is simply exhaled. Permission from the down heap cannot be handed over to the location invariant, and can also not be kept, so we assert that such permission is not held by the thread. If the memory oder is not *rel*, only permission from the up-heap can be exhaled. Other permission should not be held.

Chapter 8

Verification Tool for Weak Memory Programs

In this work we provide a tool [1] that implements the concepts described in the previous chapter. The tool consists of three parts: A parser, an encoding to Viper and the existing Viper backends.

8.1 Parser

For parsing C++ programs, we use Libclang, which is a C Interface to Clang. "The C Interface to Clang provides a relatively small API that exposes facilities for parsing source code into an abstract syntax tree (AST), loading already-parsed ASTs, traversing the AST, associating physical source locations with elements within the AST, and other facilities that support Clang-based development tools."[2]

8.2 Encoder

The encoding part is programmed in Python, using Python bindings ([1]) for Libclang. This has the advantage that we can reuse functionality from the Nagini [5] Verifier, which also uses Viper as a backend. Nagini provides code that interacts with the Viper backends, which run on the JVM. Our tool encodes C++ programs into Viper, invokes a Viper verification backend and reports back whether the verification was successful or not.

[1] The tool can be downloaded here: https://polybox.ethz.ch/index.php/s/5AL2vXFz7zVvLj6

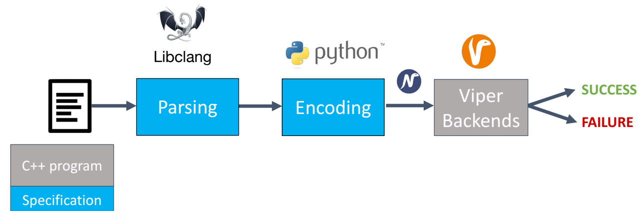

Figure 8.1: An overview of the tool.

Program	Frontend	Verification (Silicon)	Verification (Carbon)
RWSpinlock.cpp	1s	43s	58s
barrier.cpp	0.6s	27s	39s
barrier2.cpp	0.6s	24s	36s
barrier_err1.cpp	0.6s	24s	44s
arc.cpp	1s	29s	40s
arc_err1.cpp	0.7s	19.8s	21s
release-fence-ex.cpp	0.8s	620s	14.9s

Figure 8.2: Benchmarks

8.3 Benchmarks

The benchmarks show that in all cases the runtime of the frontend (parsing and encoding to Viper) is negligible. Silicon and Carbon typically need between 30s and 60s, but in the only example which contains a release fence Silicon is much slower (620s).

Chapter 9

Conclusion and Future Work

9.1 Conclusion

We presented a token-based logic for the verification of weak-memory C++ programs that is simpler to use and more suitable for automation than existing approaches. Our system allows for proofs that are intuitive and capture the essence of why the programs are correct. We also automated the release fences in the sense that they do not need annotations in our logic. We provided a Viper encoding for our logic features. We have implemented a tool that automates the encoding procedure and evaluated our work on real-world examples from concurrency libraries.

9.2 Future Work

For future work we mainly suggest two directions:

1. It is surprising how well our way of reasoning works for the three very different examples on which we evaluated it and the next step is definitely to investigate how our logic can be applied on more examples and to find its limitations.

2. Our Viper encoding currently uses fractional permissions for encoding none, read and write permission but it might be possible to avoid using fractions at all. E.g. one could use `read(r:Ref)`, `up_read(r:Ref)`, `write(r:Ref)`, `sum_is_write(r:Ref)` etc. The advantage would be that it seems like a more direct implementation of our high-level logic. On the negative side one might also loose the convenience of fractional permission based features that Viper provides.

Further ideas for future work are: proving the soundness of our logic, extending the subset of C++ that the tool supports, IDE support and improved error reporting. Also, one could re-implement the contributions of the previous work [10] on using Viper for Weak Memory programs that are not yet part of our new tool.

Acknowledgments

I would like to thank my supervisor Alexander Summers who guided me in a very kind and intelligent way during the whole project. Without him I would have been stuck immediately. I would also like to thank Professor Müller for the opportunity to work on this interesting project. Thank you all the other members of the group who were very willing to helped me with my questions on Viper and Nagini. I would also like to thank Esther Kaplony and my other friends who constantly encouraged me with kind words and also with delicious snacks.

Bibliography

[1] Libclang Python bindings. https://github.com/llvm-mirror/clang/tree/master/bindings/python. Accessed: 2019-04-10.

[2] LLVM website. https://clang.llvm.org/doxygen/group__CINDEX.html#details. Accessed: 2019-04-10.

[3] Marko Doko and Viktor Vafeiadis. A program logic for C11 memory fences. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 413–430, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[4] Marko Doko and Viktor Vafeiadis. Tackling real-life relaxed concurrency with FSL++. In Hongseok Yang, editor, *Programming Languages and Systems*, pages 448–475, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[5] Marco Eilers and Peter Müller. Nagini: A static verifier for python. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 596–603, Cham, 2018. Springer International Publishing.

[6] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, pages 41–62, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[7] Gaurav Parthasarathy. Applying and extending the weak-memory logic FSL++. 2017.

[8] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, May 2012.

[9] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.

[10] Alexander J. Summers and Peter Müller. Automating deductive verification for weak-memory programs. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 190–209, Cham, 2018. Springer International Publishing.

[11] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. *SIGPLAN Not.*, 49(10):691–707, October 2014.

[12] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for C11 concurrency. *SIGPLAN Not.*, 48(10):867–884, October 2013.

Appendix A

More details on Viper encoding

A.1 Fences

```
// fences.vpr
define fence(sync) {
    if (sync == ACQ) {
        var FenceRefSet: Set[Ref]
        FenceRefSet := havocedRefSet()
        inhale (forall r: Ref :: { (r in FenceRefSet) } (r in
            FenceRefSet) ==> heap(r) == 0 && (!is_ghost(r) && perm(
            down(r).val) > none))
        inhale (forall r: Ref :: { down(r) } perm(down(r).val) > none
             && !is_ghost(r) ==> (r in FenceRefSet))
        inhale (forall r: Ref :: { (r in FenceRefSet) } (r in
            FenceRefSet) ==> acc(r.val, perm(down(r).val)))
        inhale (forall r: Ref :: { (r in FenceRefSet) } { down(r) } (
            r in FenceRefSet) ==> r.val == down(r).val)
        exhale (forall r: Ref :: { (r in FenceRefSet) } (r in
            FenceRefSet) ==> acc(down(r).val, perm(down(r).val)))
    }
    if (sync == REL) {
        var FenceRefSet: Set[Ref]
        FenceRefSet := havocedRefSet()
        inhale (forall r: Ref :: { (r in FenceRefSet) } (r in
            FenceRefSet) ==> heap(r) == 0 && (!is_ghost(r) && perm(r.
            val) > none))
        inhale (forall r: Ref :: { heap(r) } heap(r)==0 && perm(r.val
            ) > none && !is_ghost(r) ==> (r in FenceRefSet))
        inhale (forall r: Ref :: { (r in FenceRefSet) } (r in
            FenceRefSet) ==> acc(up(r).val, perm(r.val)))
        inhale (forall r: Ref :: { (r in FenceRefSet) } { heap(r) } (
            r in FenceRefSet) ==> up(r).val == r.val)
        exhale (forall r: Ref :: { (r in FenceRefSet) } (r in
            FenceRefSet) ==> acc(r.val, perm(up(r).val)))
    }
}
// end of file
```

A.2 exhaleSource definition

```
// exhaleSource.vpr

define exhaleSource(loc,count,sum,sync) {
    assert acc(tokCountRef(loc).ownsSource)
    produceMissingTokens(loc,count)
    exhale acc(posTok(tokCountRef(loc)), perm(negTok(tokCountRef(loc)
        ))-count/1)
    exhale acc(negTok(tokCountRef(loc)), perm(negTok(tokCountRef(loc)
        )))
    exhale acc(tokCountRef(loc).ownsSource)

// give to the loc inv what is needed
    if (sum == READ) {
        if (is_ghost(loc)) {
            assert perm(loc.val) > 0/1
            exhale acc(loc.val, perm(loc.val)/2)
        }
        elseif (isRel(sync) && perm(loc.val) > 0/1 && perm(up(loc).
            val) > 0/1) {
            exhale acc(loc.val, perm(loc.val))
        }
        elseif (perm(temp(loc).val) > 0/1) {
            exhale acc(temp(loc).val, perm(temp(loc).val)/2)
        }
        elseif (isRel(sync) && perm(loc.val)>0/1) {
            exhale acc(loc.val, perm(loc.val)/2)
        }
        else {
            assert perm(up(loc).val) > 0/1
            exhale acc(up(loc).val, perm(up(loc).val)/2)
        }
    }
    elseif (sum == WRITE) {
        if (is_ghost(loc)) {
            exhale acc(loc.val, write)
        }
        elseif (isRel(sync)) {
            assert perm(loc.val) + perm(up(loc).val) + perm(temp(loc)
                .val) == write
            exhaleAll(loc)
            exhaleAll(up(loc))
            exhaleAll(temp(loc))
        }
        else {
            assert perm(up(loc).val) + perm(temp(loc).val) == write
            exhaleAll(up(loc))
            exhaleAll(temp(loc))
        }
    }
    else {
        assert sum == NONE
    }

// give away permission that cannot be kept
    if (is_ghost(loc)) {
        if (perm(posTok(tokCountRef(loc))) == 0/1) {
            exhaleAll(loc)
        }
    }
    else {
// case thread keeps no tokens
```

```
        if (perm(posTok(tokCountRef(loc))) == 0/1) {
            if (sync == REL || sync == REL_ACQ) {
// exhale everything that thread has left on the real heap
                exhaleAll(loc)
            }
            else {
// assert it does not have anything on the real heap
// TODO: example where this fails
                assertNone(loc)
            }
            exhaleAll(up(loc))
        }
// case the thread keeps one token
        elseif (perm(posTok(tokCountRef(loc))) == 1/1) {
// if there is something on the down-heap, nothing else can be kept
            if (perm(down(loc).val) > 0/1) {
                if (isRel(sync)) {
                    exhaleAll(loc)
                }
                else {
                    assertNone(loc)
                }
                exhaleAll(up(loc))
                if (isAcq(sync)) { // todo: explain
                    exhaleAll(temp(loc))
                }
                else {
// we keep what is on the temp heap
// because it will end up on the down-heap and might add up to "write
                        " there
                }
            }
// case there is something on the up-heap
            elseif (perm(up(loc).val) > 0/1) {
// case it will add up to write -> sacrifice up-modality
                if (perm(loc.val) + perm(up(loc).val) == 1/1
                    || (perm(loc.val) + perm(up(loc).val) + perm(temp
                        (loc).val) == 1/1) && isAcq(sync))
                {
                    inhale acc(loc.val, perm(up(loc).val))
                    exhaleAll(up(loc))
                }
// keep only what is on up-heap
                else {
                    exhaleAll(up(loc))
                    exhaleAll(temp(loc))
                    if (isRel(sync)) {
                        exhaleAll(loc)
                    }
                    else {
                        assertNone(loc)
                    }
                }
            }
        }
// case the thread can keep two tokens and it might be a problem
// because the thread already has some permission on up-heap
        elseif (perm(posTok(tokCountRef(loc))) == 2/1 && perm(up(loc)
            .val) > 0/1) {
// case something will end up on down-heap
            if (temp(loc).val > 0/1 && !isAcq(sync)
                || perm(down(loc).val) > 0/1)
            {
// case we need to sacrifice up-modality
```

```
                if (perm(loc.val) + perm(up(loc).val) + perm(temp(loc
                    ).val) == 1/1
                   || perm(loc.val) > 0/1 && !isRel(sync)) {
                    inhale acc(loc.val, perm(up(loc).val))
                    exhaleAll(up(loc))
                }
// just exhale what is and what would end up on real heap
                else {
                    if (isRel(sync)) {
                        exhaleAll(loc)
                    }
                    else {
                        assertNone(loc)
                    }
                    exhaleAll(temp(loc))
                }
            }
        }
    }
}
// end of file
```

A.3 Basic Viper definitions

```
// imports.vpr

import "tokens.vpr"
import "inhaleTokens.vpr"
import "exhaleTokens.vpr"
import "RMWs.vpr"
import "basicDefinitions.vpr"
import "parallelHeaps.vpr"
import "tacasDefinitions.vpr"
import "tacasDefinitions.vpr"
import "inhaleSource.vpr"
import "exhaleSource.vpr"
import "tokenCounting.vpr"
import "nonAtomics.vpr"
import "fences.vpr"
// end of file
```

```
// basicDefinitions.vpr

field val: Int

define RLX 0
define REL 1
define ACQ 2
define REL_ACQ 3

define READ 1
define WRITE 2
define NONE 3

define exhaleAll(loc) {
    exhale acc(loc.val, perm(loc.val))
}
define assertNone(loc) {
    assert perm(loc.val) == 0/1
}

define isAcq(sync) (sync == ACQ || sync == REL_ACQ)
```

```
define isRel(sync) (sync == REL || sync == REL_ACQ)

define upOrReal(loc, sync) (sync == REL || sync == REL_ACQ) ? loc :
    up(loc)
define downOrReal(loc, sync) (sync == ACQ || sync == REL_ACQ) ? loc :
     down(loc)

define Uninit(x) acc(x.init) && acc(x.val) && !x.init && acc(
    tokCountRef(x).ownsSource)

define store(v, newVal, sync) {
    assert SomeRel(v)
    ExhaleInv0(newVal,sync)
    inhale acc(v.init, wildcard)
}

define produceMissingTokens(loc, wanted) {
    if (perm(negTok(tokCountRef(loc))) < wanted/1) {
        inhale acc(posTok(tokCountRef(loc)), wanted/1 - perm(negTok(
            tokCountRef(loc))))
        inhale acc(negTok(tokCountRef(loc)), wanted/1 - perm(negTok(
            tokCountRef(loc))))
    }
}

// uninteresting definitions
method havocedPerm() returns (res:Perm)
method havocedInt() returns (res:Int)
method havocedBool() returns (res:Bool)
method havocedRefSet() returns (res:Set[Ref])
// end of file
```

```
// tacasDefinitions.vpr

field init: Bool // value is used for nonatomics, only permissions
    are used for atomics
field rel: Int
field acq: Bool // use true to indicate RMWAcq, false to indicate
    normal Acq
define SomeRel(x) acc(x.rel, wildcard)
define SomeAcq(x) acc(x.acq, wildcard) && x.acq == true
define SomeRMWAcq(x) acc(x.acq, wildcard) && x.acq == false
define SomeAcqOrRMWAcq(x) acc(x.acq, wildcard)
predicate Acq(x: Ref, idx: Int)
define Init(x) acc(x.init, wildcard)

define Rel(x, idx) SomeRel(x) && x.rel == idx
define RMWAcq(x, idx) SomeRMWAcq(x) && acc(Acq(x, idx), wildcard)

// end of file
```

A.4 Parallel Heaps

```
domain parallelHeaps {
    function up(x: Ref): Ref
    function down(x: Ref): Ref
    function up_inv(x: Ref): Ref
    function down_inv(x: Ref): Ref
    function temp(x: Ref): Ref
    function temp_inv(x: Ref): Ref
    function heap(x: Ref): Int
    function is_ghost(x: Ref): Bool
```

```
    axiom inv_up {
        (forall r: Ref :: { up(r) } up_inv(up(r)) == r && (is_ghost(r
            ) ? up(r) == r : heap(up(r)) == heap(r) + 1))
    }
    axiom inv_up_inv {
        (forall r: Ref :: { up_inv(r) } up(up_inv(r)) == r && (
            is_ghost(r) ? up_inv(r) == r : heap(up_inv(r)) == heap(r)
             - 1))
    }
    axiom inv_down {
        (forall r: Ref :: { down(r) } down_inv(down(r)) == r && (
            is_ghost(r) ? down(r) == r : heap(down(r)) == heap(r) -
            1))
    }
    axiom inv_down_inv {
        (forall r: Ref :: { down_inv(r) } down(down_inv(r)) == r && (
            is_ghost(r) ? down_inv(r) == r : heap(down_inv(r)) ==
            heap(r) + 1))
    }
    axiom inv_temp {
        (forall r: Ref :: { temp(r) } temp_inv(temp(r)) == r && (
            is_ghost(r) ? temp(r) == r : heap(temp(r)) == heap(r) -
            3))
    }
    axiom inv_temp_inv {
        (forall r: Ref :: { temp_inv(r) } temp(temp_inv(r)) == r && (
            is_ghost(r) ? temp_inv(r) == r : heap(temp_inv(r)) ==
            heap(r) + 3))
    }
}
// end of file
```

A.5 Barrier without precondition

This version of the barrier does not need a precondition. Like the other, it can only be used for one epoch. This proof uses an old version of the syntax. $\boxed{loc : (n)^-, \tau)}$ is now written as $\mathsf{Src}(loc, n, \tau)$ and $\boxed{loc : (n)^+, \tau)}$ as $\mathsf{Tok}(loc, n, \tau)$.

A.5.1 Specification

b is the amount of bits that are available for the value/reader count. The actual value at the location is $bits$. $bits$ is a bit-level encoding of two counters r and v. $bits = r + 2^b v$. pd stands for "possible decrements".

Location invariant

$Q(bits = (r, v)) := 0 \leq r \wedge r \leq v \wedge$ **if** $v <= n$ **then**

$\Big(\boxed{pd : ((r)^-, none} *$

if $v < n$ then $\boxed{data : ((2 * r)^-, write)}$

else if (r,v) = (0,n) then $\boxed{data : (1^-, none)}$

else $\boxed{data : ((2 * r)^-, none)}\Big)$

else if $v > n$ **then** $* \boxed{data : 2 * (bits)^{-}, none} * \boxed{pd : (bits - (n+1) * 2^b)^{-}, none}$

Pre- and Postconditions

$\{emp\}$ `wait()` $\{emp\}$

A.5.2 Proof outline

```
wait() {
    {emp}
    (r1, v1) := fetch_and_add_acq(VRC, (1, 1)) // RMW1
    {v1 > n ⟹ [pd : (2^b + 1, none)^+] * [data : (2 * (2^b + 1))^+, none)]}
    ∧  v1 = n ∧ r = 0 ⟹ [pd : (1, none)^+] * [data : (2^(b+1)(n + 1) + 1)^+, none)]
    ∧  v1 = n ∧ r > 0 ⟹ [pd : (1, none)^+] * [data : (2^(b+1)(n + 1) + 2)^+, none)]
    v1 = n − 1 ⟹ [pd : (1, none)^+] * [data : (2, write)^+]
    ∧  v1 < n − 1 ⟹ [pd : (1, none)^+] * [data : (2, none)^+]}
    if(v1 == n-1) {
        //update signal data structure
    }
    (r2, v2) := fetch_and_add_rel_acq(VRC, (-1,0)) // RMW2
    {(r2, v2) = (1, n)? [data : (1, write)^+] : emp}
    if((r2, v2) == (1,n)) {
        //deallocate signal data structure
    }
    {emp}
}
```

A.5.3 Explanations

RMW1:

Case we read $v > n$: The "branch" of the location invariant that is used stays the same, and it only depends on $bits$. $bits$ is increased by $2^b + 1$ and it is straightforward to see how many positive tokens the thread gets.

Case we read $v = n \land r = 0$: The location invariant initially contains $(0)^{-}$ tokens for pd and $(1)^{-}$ tokens for $data$. After updating the value, $bits = 2^b(n+1) + 1$, which means the location invariant keeps $bits - 2^b(n+1) = 2^b(n+1) + 1 - 2^b(n+1) = 1$ negative tokens for pd (the same amount of positive tokens goes to the thread). The amount of negative tokens for $data$ that is left at the invariant is $2(bits) = 2(2^b(n+1)+1)$. Since the invariant already had -1 before, the thread gets $2^{b+1}(n+1) + 1$ positive tokens.

Case we read $v = n \land r > 0$: The location invariant initially contains $(r)^{-}$ tokens for pd and $(2r)^{-}$ tokens for $data$. After updating the value, $bits = 2^b(n+1) + r + 1$, which means the location invariant keeps $bits - 2^b(n+1) = 2^b(n+1) + 1 + r - 2^b(n+1) = r + 1$ negative tokens for pd. Since the count of

$bits_{old} = 2^b v + r$	$Q(bits_{old})$	Thread (bef.)	$Q(bits_{new})$	Thread (aft.)
$v > n$	$(bits - (n+1)2^b)^-$	$(0)^+$	$(bits - (n+1)2^b + (2^b+1))^-$	$(2^b+1)^+$
$(v = n) \land (r = 0)$	$(0)^-$	$(0)^+$	$(1)^-$	$(1)^+$
$(v = n) \land (r > 0)$	$(r)^-$	$(0)^+$	$(r+1)^-$	$(1)^+$
$v < n$	$(r)^-$	$(0)^+$	$(r+1)^-$	$(1)^+$

Table A.1: The effect of `RMW1`, which adds $2^b + 1$ to the value at the atomic location, with respect to the tokens of the ghost location *pd*.

$bits_{old} = 2^b v + r$	$Q(bits_{old})$	Thread (bef.)	$Q(bits_{new})$	Thread (aft.)
$v > n$	$(2 \cdot bits_{old})^-$	$(0)^+$	$(2(bits_{old} + (2^b+1)))^-$	$(2^b+1)^+$
$(v = n) \land (r = 0)$	$(1)^-$	$(0)^+$	$(2^{b+1}(n+1) + 2)^-$	$(2^{b+1}(n+1) + 1)^+$
$(v = n) \land (r > 0)$	$(2r)^-$	$(0)^+$	$(2^{b+1}(n+1) + 2(r+1))^-$	$(2^{b+1}(n+1) + 2)^+$
$v = n - 1$	$(2r)^-$	$(0)^+$	$(2(r+1))^-$	$(2)^+$
$v < n - 1$	$(2r)^-$	$(0)^+$	$(2(r+1))^-$	$(2)^+$

Table A.2: The effect of `RMW1` with respect to the tokens of *data*.

negative tokens at the invariant increased by one, the thread gets one positive token. The amount of negative tokens for *data* that is left at the invariant is $2(bits) = 2(2^b(n+1) + r + 1)$. Since the invariant already had $-2r$ before, the thread gets $2^{b+1}(n+1) + 2$ positive tokens.

Case $v = n - 1$: For *pd* the count of negative tokens at the location invariant increases by one, and one positive token goes to the thread. The negative token count of *data* increases by two. The abstract permission sum changes from *write* to *none*. The thread therefore gets the write permission in form of two tokens.

Case $v < n - 1$: For *pd* the count of negative tokens at the location invariant increases by one, and one positive token goes to the thread. The negative token count of *data* increases by two. The read gets two tokens with the sum *none*.

Tables A.1 and A.2 might help to get an overview for RMW1.

RMW2:

Let x be the number *pd*-tokens and y the number of *data*-tokens that the thread owns before the RMW operation. We know that $x > 1$ and $y > 2$ by checking all individual possible outcomes.

Case $v \leq n \land r = 0$: The invariant has $(1)^-$ tokens for *data* (i.e. only one token is missing) and the thread has at least two tokens for *data*, which is a

contradiction. Therefore this case cannot happen.

Case $v \leq n \land r = 1$: For *pd* the invariant will claim one token from the thread. This is fine, because the thread has at least one token for *pd*. For *data* the invariant will change from $(2^{-}, write)$ to $(1^{-}, none)$, which means the thread is left with $(1^{+}, write)$.

Case $v \leq n \land r > 1$: As above, the thread will give back one *pd*-token. For *data*, the invariant will change from $(r)^{-}$ to $(r-1)^{-}$, which means that the thread will have to release a *data*-token.

Case $v = n + 1 \land r = 0$: The invariant has $(0)^{-}$ tokens for *pd* (i.e. no tokens are missing) and the thread has at least one token for *pd*, which is a contradiction. Therefore this case cannot happen.

Case $v = n + 1 \land r > 0$: Since $r > 0$, we can reduce *bits* and still $v == n + 1$. So we are for sure still in the "$v > n$"-branch of the invariant. Reducing bits by one will cause the location invariant to contain one less missing token for both *pd* and *data*, which means the thread will have to give up one token each of *pd* and *data*. This is possible because as observed above, the thread owns at least one token of each.

Case $v > n+1$: Since $v > n+1$, reducing bits by one cannot lead to $v <= n$, which means that we are still in the "$v > n$"-branch of the invariant.

A.6 Full details for Atomic Reference Counter

A.6.1 C++ Code

```
#include "weak-memory-frontend.h"

class Arc {
public:
  int data = 0;
  Invariant Q = [this] (int w) -> bool {
    return (w == 0 ? Source(data, 2, none) : Source(data, 3*w, none));
  };
  atomic<int> count = atomic<int>(Q);

  void drop()
  {
    ////////////////////////////////////////////////
    Requires(Tokens(data, 3, read) && Rmw(count,Q));
    ////////////////////////////////////////////////

    int y;
    y = count.fetch_add(-1, memory_order_release);
    if (y == 1) {
      atomic_thread_fence(memory_order_acquire);
      delete this;
    }
  }

  Arc(int v)
  {
```

```
    /////////////////////////////////////////////////////////////
    Requires(RMWAcq(count, Q) && Rel(count, Q) && Uninit(data));
    Ensures(Tokens(data, 3, read) && Rmw(count,Q));
    /////////////////////////////////////////////////////////////
    data = v;
    count.store(1, memory_order_relaxed);
   }

  int arc_read()
  {
    //////////////////////////////////////////////
    Requires(Tokens(data, 3, read) && Rmw(count,Q));
    Ensures(Tokens(data, 3, read));
    //////////////////////////////////////////////

    int ret;
    ret = data;
    return ret;
  }
  void clone()
  {
    //////////////////////////////////////////////
    Requires(Tokens(data, 3, read) && Rmw(count,Q));
    Ensures(Tokens(data, 6, read));
    //////////////////////////////////////////////

    count.fetch_add(1, memory_order_relaxed);
  }

  void test()
  {
    Requires(Tokens(data, 3, read) && Rmw(count,Q));

    clone();
    Assert(Tokens(data,6,read));
    drop();
    drop();
    // int v;
    // v = arc_read();
  }
};

// needed for it to compile
int main() {
  return 0;
}
```

A.6.2 Generated Viper Code

```
import "include/imports.vpr"

function data(this:Ref): Ref
  ensures !is_ghost(result) && heap(result)==0

define InhaleInv0(w) {
  if (w == 0) {
    inhaleSource(data(this),2,NONE)
  }
```

```
  else {
    inhaleSource(data(this),3 * w,NONE)
  }
  inhale isValidLoc(data(this))
}
define ExhaleInv0(w, sync) {
  if (w == 0) {
    exhaleSource(data(this),2,NONE,sync)
  }
  else {
    exhaleSource(data(this),3 * w,NONE,sync)
  }
  if(!is_ghost(data(this))) {
    inhale acc(downOrReal(data(this),sync).val, perm(temp(data(this))
        .val))
    exhale acc(temp(data(this)).val, perm(temp(data(this)).val))
  }
}

function count(this:Ref): Ref
  ensures !is_ghost(result) && heap(result)==0

method drop(this:Ref)
{
  inhaleTokens(data(this),3,READ)
  inhale Rmw(count(this),0)
  var y:Int
  fetch_add(count(this),-1,y,REL)
  if (y == 1) {
    fence(ACQ)
    exhale acc(data(this).val)
  }
}
define exhale_precondition_drop(this) {
  exhaleTokens(data(this),3,READ)
  exhale Rmw(count(this),0)
}
define inhale_postcondition_drop(this) {
}

method Arc(this:Ref, v:Int)
{
  inhale RMWAcq(count(this),0)
  inhale Rel(count(this),0)
  inhale Uninit(data(this))
  nonAtomicWrite(data(this), v)
  store(count(this),1,RLX)
  exhaleTokens(data(this),3,READ)
  exhale Rmw(count(this),0)
}
define exhale_precondition_Arc(this, v) {
  exhale RMWAcq(count(this),0)
  exhale Rel(count(this),0)
  exhale Uninit(data(this))
}
define inhale_postcondition_Arc(this, v) {
  inhaleTokens(data(this),3,READ)
  inhale Rmw(count(this),0)
}

method arc_read(this:Ref) returns (res:Int)
{
  inhaleTokens(data(this),3,READ)
  inhale Rmw(count(this),0)
```

```
  var ret:Int
  nonAtomicRead(data(this), ret)
  res := ret
  exhaleTokens(data(this),3,READ)
}
define exhale_precondition_arc_read(this) {
  exhaleTokens(data(this),3,READ)
  exhale Rmw(count(this),0)
}
define inhale_postcondition_arc_read(this, res) {
  inhaleTokens(data(this),3,READ)
}

method clone(this:Ref)
{
  inhaleTokens(data(this),3,READ)
  inhale Rmw(count(this),0)
  fetch_add_discard(count(this),1,RLX)
  exhaleTokens(data(this),6,READ)
}
define exhale_precondition_clone(this) {
  exhaleTokens(data(this),3,READ)
  exhale Rmw(count(this),0)
}
define inhale_postcondition_clone(this) {
  inhaleTokens(data(this),6,READ)
}

method test(this:Ref)
{
  inhaleTokens(data(this),3,READ)
  inhale Rmw(count(this),0)
  exhale_precondition_clone(this)
  inhale_postcondition_clone(this)
  assert(tokens(data(this),6,wildcard))
  exhale_precondition_drop(this)
  inhale_postcondition_drop(this)
  exhale_precondition_drop(this)
  inhale_postcondition_drop(this)
}
define exhale_precondition_test(this) {
  exhaleTokens(data(this),3,READ)
  exhale Rmw(count(this),0)
}
define inhale_postcondition_test(this) {
}

define InhaleInv(loc, v) {
  if (perm(Acq(loc,0)) > 0/1) {
    InhaleInv0(v)
  }
  else {
    assert false
  }
}
define ExhaleInv(loc, v, sync) {
  if (loc.rel==0) {
    ExhaleInv0(v, sync)
  }
  else {
    assert false
  }
}
```