A Separation Logic for Stacked Borrows

MSc Thesis *(Afstudeerscriptie)*

written by

Daniël Louwrink

under the supervision of **Robbert Krebbers (TU Delft, RU Nijmegen)** and **Alban Ponse**, and submitted to the Examinations Board in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
January 29, 2021	Robbert Krebbers (Daily supervisor)
	Alban Ponse (Supervisor)
	Benno van den Berg
	Marieke Huisman
	Christian Schaffner (Chair)

Institute for Logic, Language and Computation

Abstract

Rust is a systems programming language that uses a strong static type system to prevent memory safety issues (such "use-after-free" bugs and buffer overflows) that are common in programs written in languages such as C and C++. The RustBelt project (Jung et al. 2018a) has developed a machine-checked proof of the safety guarantees provided by the Rust language, formally showing that the Rust type system indeed prevents the issues that it claims to prevent. However, RustBelt uses an operational semantics that is incompatible with certain desirable compiler optimizations inspired by the guarantees that the Rust type system provides. Recently, Jung et al. have introduced Stacked Borrows (Jung et al. 2020), a new operational semantics for Rust that allows such compiler optimizations to be performed. However, this new operational semantics has not yet been integrated into the RustBelt safety proof, meaning it has not been formally shown that the safety guarantees of the Rust type system still hold for the new Stacked Borrows semantics.

This thesis takes the first step toward updating the RustBelt safety proof to account for the new Stacked Borrows semantics. Specifically, we develop a new program logic, which we call *Stacked Borrows Separation Logic* (SBSL), for reasoning about the behavior of programs executed according to the Stacked Borrows semantics. Building on RustBelt, we have developed a machine-checked proof of the soundness of SBSL using the Coq proof assistant. The key conceptual contribution in SBSL is the notion of *ghost stacks*, which enables SBSL to abstract away irrelevant implementation details and allows for better reasoning about concurrency compared to a naive approach.

Acknowledgements

I would like to thank Robbert Krebbers for supervising several of my individual research projects during my Master of Logic program and ultimately supervising my thesis. During the projects, I learned a lot about programming languages and the Iris logic, and this was a good preparation for writing this thesis.

Moreover, I would also like to thank Ralf Jung, Derek Dreyer, and Hai Dang at the Max Planck Institute for Software Systems for having several meetings with me during the first few months of my thesis project, where we had interesting technical discussions and where I received useful feedback on some of my early ideas.

Contents

Chapter 1

Introduction

Systems programming languages like C and C++ provide a high degree of control over memory management, making them suitable for applications that are performance-sensitive or that require precise control over the underlying hardware, such as web browsers and operating systems. However, programs written in these languages are often plagued by memory safety issues (such as "use-after-free" bugs and buffer overflows), a class of bugs that is a frequent cause of security issues. For example, Microsoft estimates that around 70% of the security issues that it fixes are caused by memory safety issues (Thomas 2019).

Rust (Matsakis et al. 2014; Rust team 2020) is a new systems programming language that provides programmers with the control and performance benefits of C and C++ while preventing memory safety issues using a strong static type system. An important feature of Rust is the extensible nature of its type system: Rust allows programmers to use clearly-marked `unsafe` operations, to which the safety guarantees of the type system do not apply, in order to implement additional libraries that cannot be implemented within the constraints of the Rust type system.

The safety guarantees provided by the Rust type system have also been the subject of academic research. The most prominent work in this direction is the RustBelt project (Dang et al. 2020; Jung 2020; Jung et al. 2018a), which provided a machine-checked proof of the safety guarantees (such as type safety and memory safety) of the core Rust type system and parts of the Rust standard library that rely on `unsafe` code.

However, the operational semantics used in RustBelt is incompatible with certain desirable compiler optimizations, such as reordering of memory accesses across function calls to unknown code. Recently, Jung et al. have proposed *Stacked Borrows* (Jung et al. 2020), a new operational semantics for Rust that allows these types of optimizations. The Stacked Borrows semantics is directly inspired by the Rust type system, and this makes it at least plausible that the safety guarantees claimed by Rust still hold for the new

Stacked Borrows semantics. However, this claim has not yet been formally established, since the Stacked Borrows semantics has not been integrated into RustBelt.

This thesis takes a first step toward re-establishing the safety guarantees of Rust for the new Stacked Borrows semantics. Before stating our contributions, we briefly describe the setup of RustBelt. RustBelt established the safety guarantees of Rust by means of a *semantic type safety* proof. Specifically, RustBelt gives a *model* for the Rust type system that describes when a program is type-safe based on the program's behavior, instead of based on a fixed set of typing rules. The main benefit of the semantic approach to type safety is that it easily accounts for libraries that are implemented using `unsafe` code: one can use the semantic notion of type safety to prove that such programs behave in a type-safe manner, despite the fact that they internally rely on unsafe operations that cannot be typed according to the ordinary rules of the type system.

The RustBelt semantic model is built on top of a program logic that provides high-level rules for reasoning about the behavior of Rust programs. The program logic used for RustBelt is a variant of *concurrent separation logic* (Brookes 2007; O'Hearn 2007; O'Hearn et al. 2001; Reynolds 2002), which is a type of program logic that simplifies reasoning about programs that use concurrency and mutable state. Specifically, the RustBelt program logic is built on top of *Iris* (Jung et al. 2016, 2018b, 2015; Krebbers et al. 2017a), a modern higher-order concurrent separation logic that provides general facilities for building program logics for programming languages with a wide range of expressive features, such as concurrency, mutable state, higher-order functions, and recursive types. Iris is implemented inside the Coq proof assistant (Coq Development Team 2020) and it enjoys the support of *MoSeL* (Krebbers et al. 2018) (formerly *Iris Proof Mode* (Krebbers et al. 2017b)), a proof mode that simplifies the process of writing separation logic proofs inside of Coq.

The rules of the program logic used in RustBelt are not sound for the new Stacked Borrows operational semantics, because the behavior of some programs is different according to the new semantics. This means that the RustBelt safety proof does not directly apply to the new Stacked Borrows semantics. Therefore, a key requirement for integrating Stacked Borrows into RustBelt is to develop a new program logic for Stacked Borrows. In this thesis, we develop such a logic, which we call *Stacked Borrows Separation Logic* (SBSL for short). The current version of SBSL only considers a simplified version of Stacked Borrows, but this simplified version nevertheless contains some of the key mechanisms present in the full version of Stacked Borrows.

We now briefly describe the key idea behind SBSL. Standard separation logic, as used in RustBelt, provides the *points-to assertion* $\ell \mapsto v$, which is a logical proposition that roughly speaking states that the location ℓ currently holds the value v in the current (implicit) program state. In the

new Stacked Borrows semantics, each memory location ℓ has an additional piece of information associated to it in addition to its current value v: namely, a *borrow stack* S, which is used to determine what kind of memory accesses are allowed to that memory location.

A naive approach to extending separation logic to account for Stacked Borrows would be to introduce the equivalent of the points-to assertion for borrow stacks, by introducing a logical proposition that tracks the current borrow stack S for each memory location ℓ. However, this naive approach leads to a program logic that is sensitive to irrelevant details of a program's implementation and incapable of reasoning about simple forms of concurrency allowed by the Rust type system.

Instead, SBSL takes a different approach: it introduces the notion of *ghost stacks* [1], which do not precisely track the borrow stack for each location, but instead track an approximation of it. This is the main conceptual contribution of this thesis, and it enables SBSL to abstract away irrelevant implementation details and additionally enables better reasoning about concurrency.

1.1 Contributions

This thesis makes the following contributions:

- We have developed a new program logic for Stacked Borrows, called Stacked Borrows Separation Logic (SBSL), that introduces the notion of *ghost stacks*.
- We demonstrate the usefulness of SBSL by applying it to several simple Rust programs that contain patterns commonly used in Rust. We also use examples to show that SBSL is better able to reason about abstraction and concurrency compared to a naive approach.
- Building on top of the Iris logic, we provide a soundness proof for SBSL.
- We have fully mechanized the rules of SBSL and its soundness proof inside of the Coq proof assistant. Our Coq development builds on the existing RustBelt Coq development, and this provides a path forward for further integrating SBSL into RustBelt.

1.2 Thesis outline

We now provide a brief outline of the chapters of this thesis.

Chapter 2 provides an introduction to the Rust programming language. It first gives an informal introduction of the Rust language, introducing important concepts of the language such as *ownership*, *borrowing*, and *undefined*

[1] Ghost stacks are implemented using Iris' *ghost state* mechanism, from which they derive their name.

behavior. Understanding these concepts is mainly important to understand the motivation for the new Stacked Borrows operational semantics. At the end of the chapter, we give a formal account of λ_{Rust}, the simplified Rust-like language that was used for the original RustBelt safety proof.

Chapter 3 introduces the simplified version of the Stacked Borrows operational semantics that is used in throughout this thesis. First, we provide the motivation for Stacked Borrows based on one of the examples from the original Stacked Borrows paper (Jung et al. 2020). The remainder of the chapter gives a formal account of $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$, a variant of λ_{Rust} that includes the new Stacked Borrows semantics.

Chapter 4 is the core chapter of this thesis. It introduces Stacked Borrows Separation Logic, the new program logic that we have developed. It begins by introducing ordinary concurrent separation logic, as used in RustBelt. Next, it describes a naive approach to extending separation logic to account for Stacked Borrows. Although this approach is not the one adopted by SBSL, it provides a useful intermediate step for understanding SBSL. We describe the shortcomings of the naive approach using two examples based on abstraction and concurrency. The remainder of the chapter is devoted to describing SBSL. We apply SBSL to the two examples used for the naive approach and show that SBSL is better able to reason about them. Finally, we discuss the relation between SBSL and the Rust type system.

Chapter 5 describes the Iris model of SBSL. We describe how the assertions of SBSL are expressed in terms of more primitive Iris notions and how they are related to the operational semantics of the programming language. Finally, we state the adequacy theorem for SBSL, which ensures that the rules of the logic are sound with respect to the operational semantics.

1.3 Notation and conventions

Throughout this thesis, we frequently use the name of a (meta)variable to indicate the kind of mathematical object that can be substituted for or is referred to by the (meta)variable. For example, when we provide a new definition such as $v \in \mathit{Val} ::= \ldots$, then the (meta)variable v should be subsequently understood to stand for mathematical objects in *Val*, where the different syntactic forms for *Val* are listed in $\ldots$.

We use the following notation for lists: $[]$ is the empty list, and $x :: xs$ (where x is an element and xs is a list of elements) is the list xs with the single element x added to the front of it. Lists of multiple elements are also written as $[x_1, x_2, \ldots, x_n]$, which is equivalent to writing $x_1 :: x_2 :: \ldots :: x_n :: []$.

The notation $t[s/x]$ is used to denote substitution of the term s for all occurrences of the variable x in the term t. Substitution of a term for a variable is defined in the usual way (performing renaming of bound variables to avoid variable capture where necessary), and we do not provide

an explicit definition. Simultaneous substitution of several terms $s_1, \ldots, s_n$ for the corresponding variables $x_1, \ldots, x_n$ is written $t[\overline{s}/\overline{x}]$, where $\overline{s}$ and $\overline{x}$ are abbreviations for the aforementioned sequences.

Propositions and functions that are applied to several arguments are occasionally written without parentheses. For example, the notation $\mathsf{P}\ 1\ 2$, where P is a binary relation (applied to two arguments), is an alternative notation for $\mathsf{P}(1, 2)$.

Chapter 2

Rust

This chapter gives an introduction to the Rust programming language. We do not discuss all details of the language, but merely focus on those aspects of Rust which are most relevant to this thesis. This chapter entirely consists of background material. The first sections of this chapter provides a fairly high-level and informal introduction to important concepts in Rust. Section 2.1 describes how the notion of *ownership* allows the Rust type system to ensure that memory is managed properly. Section 2.2 describes how borrowing works in Rust. Borrowing is an extension of the notion of ownership that allows one to create *references* to values. Section 2.4 discusses *lifetimes*, which allow the Rust compiler to ensure that references are used properly. Section 2.5 discusses *unsafe code* and *raw pointers*, which can be used to provide more flexibility when references are not sufficient. Improperly using raw pointers can lead to *undefined behavior*, a type of undesirable program behavior that is discussed in Section 2.6.

In Section 2.7, we give a formal description of the λ_{Rust} programming language, which is a programming language based on Rust that is easier to reason about formally. The λ_{Rust} language was developed for the RustBelt project (Jung et al. 2018a) in order to formalize the safety guarantees of Rust. We introduce only a simplified fragment of λ_{Rust}, leaving out aspects of the language which are orthogonal to the results in this thesis.

2.1 Ownership

An important aspect of the design of a programming language is how it deals with *memory management*. Memory management refers to the allocation and deallocation of memory while a program executes. Programs *allocate* (request) regions of memory in order to store intermediate values, and they *deallocate* (give up) regions of memory once those values will no longer be used, allowing the deallocated part of memory to be re-used in later allocations. Memory deallocation is important, since memory is a finite resource that would be

exhausted if programs did not re-use memory by deallocating.

There are several ways of dealing with memory management in a programming language. Some languages, like C, require *manual memory management*, meaning the programmer has to explicitly allocate and deallocate memory. Manual memory management is error-prone: improper memory management can lead to *memory leaks* and *use-after-free* bugs. A memory leak occurs when the programmer allocates memory to store values but forgets to deallocate the memory once those values no longer need to be stored. This can cause the program to consume more memory than necessary. A use-after-free bug occurs when the programmer deallocates memory too early, i.e. when the value that is stored there is still in use by the program. Since deallocated memory can be re-used by (other parts of) the program, this can cause the memory holding the value to be overwritten with values produced by another part of the program, leading the program to behave erratically. We will discuss the effect of use-after-free bugs further in Section 2.6, when we discuss the notion of undefined behavior.

To avoid the pitfalls of manual memory management, some languages, like Java and Python, make use of a *garbage collector*, which is an auxiliary program that runs alongside the program written by the programmer, and ensures that memory is automatically deallocated once the garbage collector has determined that the value stored there will no longer be used. Garbage collection is convenient, because the programmer can simply allocate memory and let the garbage collector take care of deallocation.

However, there are also downsides to using a garbage collector: the garbage collector runs alongside the original program, and performs checks to see whether memory can be deallocated. This can slow down the original program, and make it more unpredictable when memory will be deallocated, leading to latency characteristics that are hard to predict. For this reason, garbage-collected languages are typically avoided for performance-sensitive applications, such as operating systems and web browsers.

Rust takes a different approach to memory management, which does not incur the runtime overhead of garbage collection, while still handling memory management mostly automatically. The approach that Rust takes to memory management is based on the notion of *ownership*. We illustrate the idea of ownership using the following example:

```
1  fn print_vec() {
2      let mut vec = Vec::new();
3      vec.push(1);
4      vec.push(2);
5      vec.push(3);
6      println!("{:?}", vec); // Prints [1, 2, 3]
7  }
```

This code sample defines a function called `print_vec`, which does not take any arguments (indicated by the empty parentheses after the function name).

The body of the function, which is executed when the function is called, is enclosed in curly brackets.

This function calls the `Vec::new` function in order to create an empty *vector*. A vector stores a sequence of elements where the size of the sequence is not determined in advance. We use the `let` construct to introduce a variable `vec` and bind it to the vector we have created, allowing us to refer to the vector in the following lines. In Rust, variables are immutable by default, and therefore we use the `mut` modifier to indicate that we wish the variable to be mutable, allowing us to later add elements to the vector.

Next, the function invokes the `push` function three times on the vector in order to append the numbers 1, 2, and 3 to the back of the vector. Finally, it prints the resulting vector to the screen using `println!`, which displays the elements currently contained in the vector.

What is interesting about this example is that the vector is required to allocate memory in order to store the elements it contains. This raises an important question: when is the memory allocated by the vector deallocated?

In order to understand when memory is deallocated in Rust, it is necessary to introduce the idea of *variable scope*. The scope of a variable is the portion of the program in which we can refer to the variable by writing its name. In Rust, the scope of a (`let`-bound) variable lasts from the point where it is introduced using the `let` construct, until the *end of the enclosing block* in which the `let` is contained. Generally speaking, a block is a sequence of statements enclosed in curly brackets. In particular, the body of a function is a block.

In the above example, the variable `vec` is introduced in line 2 using `let`, and we can therefore refer to it until line 7, since that is where the enclosing block ends. While we can still refer to the variable, the variable is said to be *in scope*. At the end of the enclosing block, the variable is said to go *out of scope*, and we can no longer refer to it.

Rust uses the scope of variables to determine when memory should be deallocated. In Rust, every value is *owned* by some variable. Ownership is *exclusive*, meaning that a value cannot be owned by multiple variables. When the owning variable goes out of scope, the value is *dropped*, and the memory associated to that value is deallocated. In general, dropping a value can also cause different kinds of resource cleanup, but in this thesis we only focus on memory deallocation.

If we bind a value to a variable using `let`, then that variable becomes the exclusive owner of that value. Therefore, in the above example, the variable `vec` becomes the exclusive owner for the vector. Since `vec` goes out of scope at line 7, that is the point in which the memory used by the vector is (implicitly) deallocated. In other words, the vector is deallocated at the end of the function.

Ownership of a value can also be *transferred* by passing values as arguments to functions or returning values from functions. Transferring ownership

of a value is called *moving*. When moving a value, the original owner loses ownership and ownership is transferred to a different owner. For example, the following program transfers ownership of a vector into a separate function, which adds some elements to the vector before returning ownership of it:

```
1  fn print_vec_move() {
2      let vec = Vec::new();
3      let vec2 = add_elems_to_vec_move(vec);
4      println!("{:?}", vec2); // Prints [1, 2, 3]
5  }
6
7  fn add_elems_to_vec_move(mut vec3: Vec<i32>) -> Vec<i32> {
8      vec3.push(1);
9      vec3.push(2);
10     vec3.push(3);
11     return vec3;
12 }
```

Here, the vector `vec` is passed as an argument to `add_elems_to_vec_move` (which expects an argument of type `Vec<i32>` (a vector of integers), and returns a value of the same type). This transfers ownership from the variable `vec` into the function. At the end of `add_elems_to_vec_move`, ownership is returned by returning the vector from the function. Since `add_elems_to_vec_move` *receives* ownership of a vector (received as an argument) and subsequently *gives up* ownership of the vector (by returning it from the function), the function `add_elems_to_vec_move` does not deallocate the vector at the point where `vec3` goes out of scope. Instead, the vector is deallocated when `vec2` goes out of scope, because that variable receives (and keeps) ownership of the vector returned by `add_elems_to_vec_move` (printing the vector does not take ownership of the vector for reasons we do not explain here).

The ownership discipline in Rust ensures that memory is properly deallocated, and prevents memory leaks (with some exceptions) and use-after-free bugs. Moreover, by looking at the places where ownership transfers occur, the Rust compiler is able to determine where memory should be deallocated entirely at compile-time, meaning the overhead of garbage collection is avoided.

2.2 Borrowing

The function `add_elems_to_vec_move` in the previous section does not keep ownership of the vector. Instead, it receives the vector as an argument, modifies the vector, and then gives back the vector by returning it. Effectively, the function has only briefly "borrowed" the vector for the purpose of making some modifications to it. It is quite tedious to have to explicitly return ownership in such cases.

In order to simplify code like this, Rust also allows one to *borrow* a value by creating a *reference* to it. A reference to a value allows one to temporarily use the value, without receiving ownership of the value. References are an instance of a more general concept called *pointers*, which are values that refer to (or *point to*) a location in memory holding a value. The way that pointers are represented is made more precise in Section 2.7.

The following example demonstrates how references can be used to allow a function to modify a vector without receiving ownership of it:

```
1  fn print_vec_borrow() {
2      let mut vec = Vec::new();
3      add_elems_to_vec_borrow(&mut vec);
4      println!("{:?}", vec);
5  }
6
7  fn add_elems_to_vec_borrow(vec2: &mut Vec<i32>) {
8      vec2.push(1);
9      vec2.push(2);
10     vec2.push(3);
11 }
```

In this example, a vector `vec` is created, but this vector is not moved into the function by passing it as an argument directly. Instead, we create a *mutable reference* to the vector using `&mut vec`, which we then pass to `add_elems_to_vec_borrow`. A mutable reference provides temporary, mutable (read-write) access to the value it refers to.

The function `add_elems_to_vec_borrow` expects an argument of type `&mut Vec<i32>`, indicating that it expects a mutable reference to a vector of integers. It uses the mutable reference to the vector to add three elements to the vector referred to by the reference.

Crucially, creating a reference does not affect ownership: in the above example, the variable `vec` remains the owner of the vector, and the vector will still be dropped when `vec` goes out of scope (i.e., at the end of `print_vec_borrow`). Because a reference does not confer ownership, the `add_elems_to_vec_borrow` does not have to explicitly "return" ownership of the vector.

In fact, we have already seen code that uses mutable references before, since the `push` function we have used to add elements to the vector expects a mutable reference as well, which is created implicitly.

One of the most important operations on references is *dereferencing*, written `*r`, where `r` is a reference. If `r` is a reference, then `*r` is used to access the value *referred to* by the reference. The meaning of dereferencing is dependent on the context in which it appears. For example, `*r = ...` is used to *write* a value to a reference, and `*r` by itself is typically used to *read* the value referred to by a reference. The dereferencing operation is not always required to be written explicitly, which is why it does not appear in all of the examples.

2.3 Borrow checking

In order to prevent use-after-free bugs, it is important to ensure that a reference is not used after the value it originally referred to has been destroyed. The Rust compiler includes a component called the *borrow checker*, which ensures this by enforcing restrictions on how references can be used: in particular, it ensures that references are only used while they are still valid. Moreover, it ensures that mutable references are *exclusive*, meaning that there is at most one (usable) mutable reference to each value. Ensuring the exclusivity of mutable references prevents use-after-free bugs, as illustrated by the following example program (slightly modified from Section 2 of Jung et al. (2020)), which is rejected by the borrow checker:

```
1  fn invalid_ref_use() {
2      let mut vec = Vec::new();
3      vec.push(1);
4
5      let vec_ref = &mut vec;
6      let first_item_ref = &mut vec_ref[0];
7      *vec_ref = Vec::new();
8      *first_item_ref = 5; // Use-after-free!
9  }
```

This program creates a vector holding a single element. Then it creates a mutable reference `vec_ref` to the entire vector. It then obtains a mutable reference `first_item_ref` to just the first element of the vector. Then, it overwrites the vector with the empty vector using `vec_ref`. Doing this means that `first_item_ref` is no longer valid, since it refers to the first element of a vector that has been dropped (due to being replaced by an empty vector). Hence, subsequently writing to `first_item_ref` constitutes a use-after-free bug. The borrow checker prevents this error by imposing constraints that ensure that the reference `first_item_ref` is no longer used after the write to `vec_ref`. The key point is that if we are able to obtain two mutable references to the same value, then we are sometimes able to use one of the mutable references in order to make the other mutable reference invalid (e.g., by dropping the value it refers to).

The exclusivity of mutable references can also be re-stated in terms of *pointer aliasing*. Two pointers or references are said to *alias* when they refer to the same value. Hence, we can say that the Rust borrow checker prevents undesirable aliasing of mutable references.

Rust also allows one to create *shared references*. In contrast to mutable references, shared references are allowed to alias, but they only provide read-only access (with some exceptions). Since shared references only provide read-only access, it is not possible to reproduce the example `invalid_ref_use` above with shared references, because that example relies on overwriting the vector in order to invalidate a mutable reference.

The following is an example of a program that creates and uses multiple

(aliased) shared references to the same value:

```
fn aliasing_shared_refs() {
    let mut vec = Vec::new();
    vec.push(1);

    let vec_ref = &vec;
    let first_item_ref_1 = &vec[0];
    let first_item_ref_2 = &vec[0];
    println!("{}", first_item_ref_1); // Prints 1
    println!("{:?}", vec_ref); // Prints [1]
    println!("{}", first_item_ref_2); // Prints 1
}
```

This example shows that the syntax `&vec` is used to create a shared reference (of type `&Vec<i32>`) to the vector of integers. Additionally, two aliasing references are created to the first element of the vector and all of the references are subsequently used in a read-only manner.

2.4 Lifetimes

The borrow checker ensures that references are used properly using an analysis based on *lifetimes*. Every reference in a Rust program has a lifetime associated to it, which is the portion of the program where that reference can be used. By imposing constraints on the lifetimes of references, the borrow checker is able to ensure that references are only used while valid. In particular, lifetimes are also used to prevent undesirable aliasing of mutable references.

References in Rust can generally be created in two ways: by creating a reference to an owned value directly (borrowing), and by *deriving* a new reference from an existing reference. The latter operation is called *reborrowing*, and it is typically useful for creating a reference to a *part* of some value from a reference to the *entire* value (say, for creating a reference to the first element of the vector given a reference to the entire vector), although it is also possible for the derived reference to have the same type as the original reference. Consider the following program:

```
fn borrows() {
   let mut x = 42;

   // Borrowing, x is a variable holding an owned value
   let x_ref1 = &mut x;
   // Reborrowing, x_ref1 is an existing mutable reference
   let x_ref2 = &mut *x_ref1;
}
```

Here, we have that the reference `x_ref1` is created by borrowing `x`, whereas the reference `x_ref2` is *derived* from the another reference `x_ref1` by reborrowing. There is no special syntax for reborrowing: `&mut *x_ref1` should just be read as "create a mutable reference to the value referred to by `x_ref1`". In this

example, both `x_ref1` and `x_ref2` are mutable references of type `&mut i32` (mutable reference to an integer), and they are aliasing, since they both refer to the contents of the variable `x`. However, as we shall see, the constraints imposed by the borrow checker prevent these aliasing mutable references from being used "at the same time".

We now describe the constraints imposed by the borrow checker in order to ensure that references are used properly, preventing issues such as use-after-free bugs. The following rules are used to determine the lifetime of each reference (these rules are slightly rephrased from (Jung et al. 2020)):

- A reference (and any reference derived from it by reborrowing) can only be used during its lifetime.
- Newly-created mutable references: The original value or reference (from which the new reference is derived) cannot be used until the lifetime of the newly-created reference has ended.
- Newly-created shared references: The original value or reference (from which the new reference is derived) cannot be *mutated* (but *can* be read) until the lifetime of the newly-created reference has ended.

The borrow checker applies these constraints to determine the lifetimes of the references in the program: for each reference, it chooses the lifetime to be the smallest portion of the program that contains all uses of a reference (and uses of the references derived it), while ensuring that the latter two constraints are also respected. If it is not possible to satisfy all constraints, then the program is rejected by the borrow checker. Note that once a lifetime has ended, it cannot "start" again at some later point in the program.

Apart from the lifetime constraints, we have that mutable references can be used for reading and writing, whereas shared references can only be used for reading (with an exception that we do not discuss here). Moreover, creating a mutable reference counts as a write access, whereas creating a shared reference counts as a read access. This means that it is not possible to create a mutable reference from a shared reference.

Applying these rules to the program `invalid_ref_use` on page 14, the borrow checker determines the following lifetimes:

```
1 fn invalid_ref_use() {
2     let mut vec = Vec::new();
3     vec.push(1);
4
5     let vec_ref = &mut vec; // 'a
6     let first_item_ref = &mut vec_ref[0]; // 'a, 'b
7     *vec_ref = Vec::new(); // 'a, 'b
8     *first_item_ref = 5; // 'a, 'b
9 }
```

Lifetimes in this program are written `'a` (single quote followed by alphabetic character). The same convention is used in the Rust language to refer to lifetimes. The references `vec_ref` and `first_item_ref` have lifetimes `'a` and `'b`, respectively. The comments on the side show what portion of the program each lifetime corresponds to. The borrow checker picks the smallest lifetime that contains all uses of a reference (and references derived from it). Hence, the lifetime `'a` contains all uses of `vec_ref` and `first_item_ref`, since `first_item_ref` is derived from `vec_ref`. The lifetime `'b` simply contains all uses of `first_item_ref`.

Now, we can see that this program does not satisfy all the constraints: the program writes to `vec_ref`, while the lifetime `'b` of `first_item_ref` (derived from `vec_ref`) is *has not yet ended*. This is a violation of the lifetime constraints, since the original reference cannot be used until the lifetime of the derived reference has ended (for mutable references). Hence, the program is rejected, which in this case prevents a use-after-free bug.

For shared references, the lifetime constraints are more lenient than for mutable references, since they allow multiple aliasing shared references to co-exist (with overlapping lifetimes) and be read from in an arbitrary interleaved fashion until the point where the original value or reference is written to.

Just as with ownership, it is important to note that the lifetime analysis does not involve runtime checks: lifetimes are determined entirely at compile-time, and therefore borrow checking does not have a performance impact while a program executes.

2.5 Unsafe code and raw pointers

Based on ownership and borrowing, the borrow checker is able to ensure the absence of memory safety issues such as use-after-free bugs in the majority of Rust code. However, sometimes the ownership and borrowing constraints imposed by the borrow checker are too strict, rejecting programs that use references properly (i.e., without causing issues such as use-after-free bugs either directly or indirectly), but where the analysis is not powerful enough to determine that. For example, the Rust standard library includes reference-counted pointers, which provide a more flexible notion of ownership than that allowed by the Rust type system. Reference-counted pointers allow a single value to have multiple owners, by keeping track of the number of owners at runtime, and destroying the value once the number of owners reaches 0.

Implementing reference-counted pointers is not possible within the constraints enforced by the Rust type system, which requires each value to have a unique owner. However, Rust provides programmers with the ability to use *raw pointers*, which are similar to references in the sense that they refer to a value, but which are not subject to the lifetime checks performed by the borrow checker. Using raw pointers improperly can lead to memory safety

issues such as use-after-free bugs, and therefore Rust requires most operations on raw pointers to appear inside `unsafe` blocks, signifying to the reader that "unsafe" operations are being used.

Using raw pointers makes it possible to implement additional abstractions like reference-counted pointers. An important point is that reference-counted pointers are entirely safe to use, as long as the client code does not contain any `unsafe` blocks itself. That is, despite the fact that the implementation of reference-counted pointers relies on `unsafe` code, the interface is entirely safe to use. This means it is possible to write libraries that use `unsafe` code internally without compromising the safety guarantees of the Rust type system, assuming that such libraries are implemented correctly.

2.6 Undefined behavior

Since the borrow checker does not provide guarantees about raw pointers, it is possible to use raw pointers to cause use-after-free bugs. Consider the following program:

```
1  fn use_evil_pointer() {
2      let evil_pointer = create_evil_pointer();
3      unsafe { *evil_pointer = 10 };
4  }
5
6  fn create_evil_pointer() -> *mut i32 {
7      let mut x = 42;
8      let x_ref = &mut x;
9      let evil_pointer = x_ref as *mut i32;
10     return evil_pointer;
11 }
```

The function `create_evil_pointer` creates a variable `x` holding the value 42, and creates a mutable reference `x_ref` to it. Then, it *casts* (converts) the mutable reference to a mutable raw pointer (of type `*mut i32`) using `as *mut i32`. While mutable references are checked using the lifetime analysis, raw pointers are not checked at all. Therefore, we are able to return `evil_pointer` from the function, despite the fact that the variable `x` is deallocated at the end of the function. This means that the pointer returned by the function refers to a value that was already dropped, and hence a use-after-free bug occurs when we write to it in `use_evil_pointer`.

Here, the borrow checker does not provide any help: the Rust compiler compiles this program without complaining, and we can even run the program. This raises an interesting question: how does a program containing a use-after-free bug behave when it is executed?

The behavior of programs written in a programming language is specified by the *semantics* of the programming language. The semantics can be described in an informal manner in a document describing the programming language (a language specification), or it can be made much more precise by

giving a rigorous mathematical definition of program behavior. Typically, the goal of a semantics is to describe the behavior of programs in a way that is not dependent on specific hardware or a specific implementation of the language. Rust does not yet have an full, official language specification, but some aspects of the behavior of Rust programs are generally agreed upon (and relied on by compiler implementers).

One way of dealing with programs that contain use-after-free bugs is to stipulate that such programs should terminate with an error message. This approach has the advantage that erroneous programs behave in a predictable way. However, this approach can be costly in terms of performance, since it requires programs to keep track of administrative information and perform checks to determine when use-after-free bugs are occurring.

Instead of requiring the program to be aborted or to have some other fixed behavior, the Rust language (similar to other performance-oriented languages, like C and C++) stipulates that programs that contain use-after-free bugs have *undefined behavior*. A program that has undefined behavior is allowed to have *any behavior whatsoever*. Moreover, this behavior does not have to be consistent between different runs of the same program, different machines, or different compiler versions.

Why is it desirable to let certain programs have undefined behavior? This means that the compiler implementer is given maximum freedom in mapping the constructs of the language to the underlying hardware in the most efficient way, i.e. without adding any checks to ensure that the program is not performing "illegal" operations. Assuming that programs do not have undefined behavior can also be tremendously useful for justifying compiler optimizations, as we will see in Chapter 3.

However, this freedom comes at a significant cost: if the programmer makes a mistake and writes a program that has undefined behavior, then they no longer have any guarantee about how their program will behave. Perhaps the program will work correctly most of the time (or even all of the time), but be subtly broken when switching to different hardware, or when upgrading to a new compiler version that performs additional compiler optimizations.

Since the behavior of programs with undefined behavior cannot be relied upon, the Rust language goes to great lengths to avoid it. Hence, Rust requires every piece of code that could cause undefined behavior if used incorrectly to be enclosed in an `unsafe` block. Using the type system (mainly the borrow checker), the Rust compiler is able to ensure the absence of undefined behavior in the vast majority of Rust code, whereas the programmer is responsible for manually auditing a small number of `unsafe` blocks in the code (auditing an `unsafe` block might require looking at the surrounding code as well). For this reason, code that does not contain any `unsafe` blocks is called *safe* Rust code, whereas code that does contain `unsafe` blocks is called *unsafe* Rust code.

2.7 The λ_{Rust} programming language

This section introduces the λ_{Rust} programming language, which is a programming language that is similar to Rust, but considerably simplified in order to make it easier to reason about. The λ_{Rust} language was created for the RustBelt project (Jung et al. 2018a) in order to prove core properties of the Rust type system. We present λ_{Rust} as an *untyped* language here, even though RustBelt defines a type system for λ_{Rust} on top of the untyped language.

In contrast to Rust, λ_{Rust} has a *formal* semantics, which specifies how programs written in λ_{Rust} behave. In particular, the semantics specifies when a λ_{Rust} program has undefined behavior. As mentioned in the beginning of this chapter, we describe only a fragment of the language: in particular, we leave out mostly orthogonal aspects of the language like *non-atomic memory accesses* and the *compare-and-set* instruction, although we do take these instructions into account in the Coq formalization developed for this thesis. Moreover, we simplify memory allocation to single memory locations, instead of blocks of memory locations. This simplification to single memory locations is also applied in the Coq formalization for this thesis.

Section 2.7.1 gives a brief, informal introduction to the language. This introduction is not fully precise about every detail of the language, but merely meant to give an overview of the language constructs. In Section 2.7.2, we give the formal semantics (specifically, a small-step operational semantics) for the language. Section 2.7.3 discusses when a λ_{Rust} program has undefined behavior. Finally, Section 2.7.4 discusses how to translate Rust programs into λ_{Rust}.

2.7.1 Informal introduction

The syntax of λ_{Rust} is shown in Fig. 2.1. Some language constructs that are not relevant to this thesis have been omitted. The syntax is divided into *values* v and *expressions* e. Values represent the results of terminated programs, whereas expressions also include programs that have not (yet) terminated. Expressions can be *evaluated* in order to obtain a final value. For example, $2 + 3$ is an expression that evaluates to the value 5.

The λ_{Rust} language has four kinds of values: locations $\ell \in \mathit{Loc}$ (where Loc is an unspecified, countably infinite set), integers $z \in \mathbb{Z}$, recursive functions $\texttt{rec}\, f([x_1, \ldots, x_n]) := e$ and the "poison" value $\lightning$. The role of each of these will be explained as we explain the operations of the language.

The addition operation $e + e$ can be used to add two integers. For example, $(2 + 3) + 5$ evaluates to 10.

Function application $e([e_1, \ldots, e_n])$ applies the function e to the arguments $e_1, \ldots, e_n$. For example, $(\texttt{rec}\, f([x, y]) := x + y)([2, 3])$ applies a function f (which produces the sum of its two arguments) to the values 2 and 3, and therefore evaluates to 5. A function can call itself recursively using its own

$$
\begin{array}{rcl}
\ell \in \mathit{Loc} & & \\
z \in \mathbb{Z} & & \\
v \in \mathit{Val} & ::= & \lightning \mid \ell \mid z \mid \texttt{rec}\, f([x_1, \ldots, x_n]) := e \\
e \in \mathit{Expr} & ::= & v \\
& & \mid x \\
& & \mid e + e \mid \ldots \\
& & \mid e([e_1, \ldots, e_n]) \\
& & \mid \texttt{case}\, e\, \texttt{of}\, [e_1, \ldots, e_n] \\
& & \mid \texttt{fork}\, \{\, e \,\} \\
& & \mid \texttt{alloc}() \\
& & \mid \texttt{free}(e) \\
& & \mid {}^{*}e \\
& & \mid e := e \\
& & \mid \ldots
\end{array}
$$

$$
\begin{array}{rcl}
\lambda x.e & := & \texttt{rec}\, \_([x]) := e \\
\texttt{let}\, x = e_1\, \texttt{in}\, e_2 & := & (\lambda x.e_2)\, e_1 \\
e_1; e_2 & := & (\texttt{let}\, \_ = e_2\, \texttt{in}\, e_1) \\
\texttt{true} & := & 1 \\
\texttt{false} & := & 0 \\
\texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 & := & \texttt{case}\, e_1\, \texttt{of}\, [e_3, e_2]
\end{array}
$$

Figure 2.1: The syntax of λ_{Rust}.

name. Functions are treated as ordinary values in λ_{Rust}, just like in the λ-calculus, from which it derives its name. We write _ for variables that are not used anywhere.

The case expression $\texttt{case}\, e\, \texttt{of}\, [e_0, \ldots, e_{n-1}]$ is used to choose between evaluating different expressions. It evaluates e to an integer z, and subsequently evaluates e_z (the z-th expression, counting from 0) from $e_0, \ldots, e_{n-1}$. Hence, $\texttt{case}\, 1\, \texttt{of}\, [2, 3+3]$ evaluates to 6. Based on the case construct, it is also possible to define the more familiar if expression $\texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3$ using the constants $\texttt{true} := 1$ and $\texttt{false} := 0$.

Programs written in λ_{Rust} can read from and write to memory locations $\ell \in \mathit{Loc}$, where the set of locations Loc is an unspecified, countably infinite set. The idea is that each location in the memory can hold a single value $v \in \mathit{Val}$. Allocation $\texttt{alloc}()$ finds an unused memory location ℓ and marks it for use, storing the value ☣ at that location. The poison value ☣ is an ordinary value, that is used here to represent uninitialized memory (memory to which we have not yet written an initial value). Programs that attempt to perform certain operations (such as arithmetic) on poison values have undefined behavior.

It is possible to read from a memory location using ${}^*\ell$, which evaluates to the value currently stored at ℓ. Writing to a memory location is written $\ell := v$, which updates the value stored at location ℓ in the memory to v. Finally, deallocating a location is written $\texttt{free}(\ell)$. After deallocation, the memory location ℓ is marked as unused, allowing it to be re-used in future allocations.

Based on functions and function application, it is possible to define the let-binding construct $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$, which evaluates e_1 to a value v that is substituted for the variable x in e_2. Hence, $\texttt{let}\, x = 2 + 3\, \texttt{in}\, x + x$ evaluates to 10. Similarly, we can implement sequencing $e_1; e_2$, which evaluates e_1 and e_2 in sequence. These constructs are primarily useful in programs that access memory locations. For example,

$$\texttt{let}\, x = \texttt{alloc}()\, \texttt{in}\, x := 10; x := 5; \texttt{free}(x)$$

is a program that allocates a memory location and performs two writes to it before deallocating the location again.

In λ_{Rust}, it is also possible to have multiple programs executing concurrently and communicating with each other by reading from and writing to memory. The programs that are executed concurrently are referred to as *threads*. The fork operation $\texttt{fork}\, \{\, e\, \}$ creates an additional thread that starts executing the program e concurrently.

2.7.2 Operational semantics

This section presents the formal semantics for λ_{Rust}, which is a formal description of how programs written in λ_{Rust} behave. The semantics that we

describe here is a slimmed-down version of the full semantics developed for RustBelt (Jung et al. 2018a), merely containing those aspects of the language that we describe in this thesis.

The λ_{Rust} semantics is given as a *small-step operational semantics*. In a small-step operational semantics, the behavior of programs is specified by defining an *abstract machine* that executes programs in a step-by-step fashion. The behavior of this machine is given by a *transition relation*, that describes how the state of the abstract machine changes as a program is executed. States of the abstract machine are called *configurations*.

The λ_{Rust} language supports concurrency and mutable state. Therefore, a configuration of the λ_{Rust} abstract machine consists of the contents of the memory, as well as a list containing the threads that are executing concurrently:

$$m \in \mathit{Mem} \triangleq \mathit{Loc} \xrightarrow{\mathsf{fin}} \mathit{Val}$$
$$c \in \mathit{Config} \triangleq \mathit{Mem} \times \mathit{List}(\mathit{Expr})$$

The memory keeps track of the value stored at each location, and therefore it is represented as a finite partial function from locations $\ell \in \mathit{Loc}$ to values $v \in \mathit{Val}$. The set of locations is left unspecified, but is assumed to be countably infinite. This means that a memory can hold an arbitrary (but finite) number of values. Since memories are finite, they can be written as lists of (location, value)-pairs. For example,

$$\{\ell_1 := 2, \ell_2 := 3\}$$

is a memory in which the location ℓ_1 holds the value 2 and the location ℓ_2 holds the value 3. Locations which are not mapped represent unused locations.

We will write machine configurations as

$$(m \mid \langle e_1 \mid\mid \ \ldots \ \mid\mid e_n \rangle)$$

where m is the current memory state and $e_1, \ldots, e_n$ are the programs that are being executed concurrently. Hence,

$$(\{\ell_1 := 0\} \mid \langle 2 + 3 \mid\mid 4 + 5 \rangle)$$

represents a state where the programs $2 + 3$ and $4 + 5$ are being executed concurrently and the memory has a single location ℓ_1 holding the value 0.

The behavior of the λ_{Rust} abstract machine is given by a transition relation

$$\cdot \to_{\mathsf{tp}} \cdot \subseteq \mathit{Config} \times \mathit{Config}$$

where $c_1 \to_{\mathsf{tp}} c_2$ can be read as "if the current configuration of the machine is c_1, then the machine can take a step to end up in the next configuration c_2".

O-Add
$$\frac{z_1 + z_2 = z}{(m \mid z_1 + z_2) \to (m \mid z)}$$

O-App
$$(m \mid (\texttt{rec}\, f(\overline{x}) := e)(\overline{v})) \to (m \mid e[(\texttt{rec}\, f(\overline{x}) := e)/f, \overline{v}/\overline{x}])$$

O-Case
$$(m \mid \texttt{case}\, z\, \texttt{of}\, [e_0, \ldots, e_z, \ldots, e_{n-1}]) \to (m \mid e_z)$$

O-Fork
$$(m \mid \texttt{fork}\, \{\, e\, \}) \to (m \mid \text{☣}, e)$$

O-ECtx
$$\frac{(m_1 \mid e_1) \to (m_2 \mid e_2, e_f^?)}{(m_1 \mid K[\, e_1\,]) \to (m_2 \mid K[\, e_2\,], e_f^?)}$$

O-Mem-Alloc
$$\frac{\ell \notin \mathsf{dom}(m)}{(m \mid \texttt{alloc}()) \to (m[\ell \leftarrow \text{☣}] \mid \ell)}$$

O-Mem-Free
$$\frac{\ell \in \mathsf{dom}(m)}{(m \mid \texttt{free}(\ell)) \to ((m \setminus \ell) \mid \text{☣})}$$

O-Mem-Write
$$\frac{\ell \in \mathsf{dom}(m)}{(m \mid \ell := v) \to (m[\ell \leftarrow v] \mid \text{☣})}$$

O-Mem-Read
$$\frac{\ell \in \mathsf{dom}(m)}{(m \mid {}^*\ell) \to (m \mid m(\ell))}$$

Figure 2.2: Operational semantics of λ_{Rust}.

This relation is called the *thread-pool reduction relation* because it manages the machine configuration, which contains a list (pool) of threads. When executing the program e, the initial configuration of the machine is $(\varnothing \mid \langle e \rangle)$, i.e. e starts out as the only thread and the memory is initially empty.

The behavior of individual threads is given by a transition relation

$$(\cdot \mid \cdot) \to (\cdot \mid \cdot, \cdot) \subseteq \mathit{Mem} \times \mathit{Expr} \times \mathit{Mem} \times \mathit{Expr} \times \mathit{Expr}^?$$

where $(m_1 \mid e_1) \to (m_2 \mid e_2, e_f^?)$ can be read as "if the current memory is m_1, then the thread e_1 can take a step to become e_2, where the memory changes to m_2 and a new thread e_f is created". The set $\mathit{Expr}^? \triangleq \mathit{Expr} \uplus \{\bot\}$ is used to indicate that creating a new thread is optional. This relation, which is called the *per-thread reduction relation*, defines the actual behavior of the language constructs. The rules for the relation are shown in Fig. 2.2. For language constructs which do not create threads, e_f is omitted.

The machine takes steps by having one of the threads e_i (the choice i is

$$
\begin{aligned}
K \in \mathit{ECtx} ::= {} & \bullet \\
& \mid K + e \\
& \mid v + K \\
& \mid K([e_1, \ldots, e_n]) \\
& \mid v([v_1, \ldots, v_i, K, e_j, \ldots, e_n]) \\
& \mid \texttt{case}\ K\ \texttt{of}\ [e_1, \ldots, e_n] \\
& \mid \texttt{free}(K) \\
& \mid {}^*K \\
& \mid K := e \\
& \mid v := K
\end{aligned}
$$

Figure 2.3: Evaluation contexts of λ_{Rust}.

non-deterministic) in its thread-pool take a step:

Tp-Step

$$
\frac{(m_1 \mid e_i) \to (m_2 \mid e_i', e_f^?)}{(m_1 \mid \langle e_1 \mid\mid \ \ldots \ \mid\mid e_i \mid\mid \ \ldots \ \mid\mid e_n \rangle) \to_{\mathsf{tp}} (m_2 \mid \langle e_1 \mid\mid \ \ldots \ \mid\mid e_i' \mid\mid \ \ldots \ \mid\mid e_n \mid\mid e_f^? \rangle)}
$$

If a new thread e_f is created, then that thread is added to the thread pool.

We now briefly discuss each of the rules in Fig. 2.2. The rule O-Add describes the behavior of addition. It allows us to derive transitions such as $(m \mid 2 + 3) \to (m \mid 5)$. According to O-App, function applications are evaluated by substituting the argument values $\overline{v}$ for the parameters $\overline{x}$ of the function, and substituting the function itself for the name f of the function. The rule O-Case states that case expressions are evaluated by evaluating the chosen branch e_z. The rule O-Fork states that the fork expression $\texttt{fork}\,\{\, e \,\}$ evaluates to ☣ and creates a new thread executing e.

The rule O-ECtx describes the evaluation of expressions that contain subexpressions that need to be evaluated before applying an operation. For example, in $(2 + 3) + 5$, $2 + 3$ needs to be evaluated before adding 5 to it. The evaluation order of subexpressions in λ_{Rust} is described using *evaluation contexts*. An evaluation context is a program with a "hole" $\bullet$, where the hole describes where evaluation is taking place. The syntax of evaluation contexts K is shown in Fig. 2.3. These evaluation contexts indicate a left-to-right, call-by-value (strict) evaluation strategy. For example, $\bullet + 5$ is an evaluation context of the form $K + e$. We can plug an expression into the hole of an evaluation context by writing $K[\, e \,]$ where K is an evaluation context and e is an expression. For example, $(\bullet + 5)[\, 2 + 3 \,]$ is the expression $(2 + 3) + 5$. Based on O-ECtx and O-Add, we are able to derive steps such as

$$(m \mid (2 + 3) + 5) \to (m \mid 5 + 5)$$

The rules O-Mem-Alloc, O-Mem-Read, O-Mem-Write and O-Mem-Free, combined with a description of how the state of the memory is represented (e.g., as a finite partial mapping from locations to values), constitute the *memory model* of the λ_{Rust} language. The memory model defines the behavior of the memory-related operations. These rules rely on some additional notation for describing changes to the memory: the notation $m[\ell \leftarrow v]$ is the memory m where ℓ has been updated to hold v, $m \setminus \ell$ is the memory m with the mapping for ℓ removed, $m(\ell)$ is the value currently stored at ℓ in the memory m, and $\mathsf{dom}(m)$ is the *domain* of the memory, or the set of locations mapped to some value.

The rule O-Mem-Alloc states that `alloc()` non-deterministically picks an unused location $\ell \notin \mathsf{dom}(m)$, stores ☣ at that location and evaluates to ℓ. It is always possible to find a location $\ell \notin \mathsf{dom}(m)$, since the set of locations is infinite whereas $\mathsf{dom}(m)$ is always finite. This means that memory allocation always succeeds. The rule O-Mem-Read states that $^*\ell$ evaluates to the value stored at location ℓ in the memory, if $\ell \in \mathsf{dom}(m)$. The rule O-Mem-Write states that $\ell := v$ (where $\ell \in \mathsf{dom}(m)$) updates the value stored at location ℓ to v, and evaluates to ☣. Finally, the rule O-Mem-Free states that $\texttt{free}(\ell)$ removes the mapping for ℓ from the memory and evaluates to ☣, if $\ell \in \mathsf{dom}(m)$.

Using the rules of the operational semantics, we can derive machine transitions such as

$$(\{\ell_1 := 2\} \mid \langle \texttt{alloc}() \parallel 4 + 5 \rangle) \rightarrow_{\mathsf{tp}} (\{\ell_1 := 2, \ell_2 := ☣\} \mid \langle \ell_2 \parallel 4 + 5 \rangle)$$

By repeatedly performing machine transitions, we can keep evaluating a thread until it has reduced to a value, at which point that thread has terminated. For example, if we evaluate $(2+3)+5$ by starting from an initial configuration $(\varnothing \mid \langle (2+3)+5 \rangle)$, then the machine executes the following transitions:

$$\begin{aligned} & (\varnothing \mid \langle (2+3)+5 \rangle) \\ \rightarrow_{\mathsf{tp}} & (\varnothing \mid \langle 5+5 \rangle) \\ \rightarrow_{\mathsf{tp}} & (\varnothing \mid \langle 10 \rangle) \end{aligned}$$

In the last configuration, the initial (and only) thread has terminated with the value 10. The machine can no longer make any transitions from the last configuration, because none of the rules for making transitions apply in that configuration. However, if there were additional threads, then it would be possible for the machine to keep making transitions despite the fact that the initial thread has terminated.

We can express that a machine configuration c_2 is reachable from the configuration c_1 using a sequence of thread-pool reductions by taking the reflexive, transitive closure $\rightarrow^*_{\mathsf{tp}}$ of the thread-pool reduction relation. Hence, we have that $(\varnothing \mid \langle (2+3)+5 \rangle) \rightarrow^*_{\mathsf{tp}} (\varnothing \mid \langle 10 \rangle)$.

2.7.3 Undefined behavior in λ_{Rust}

In Section 2.7.2, we have presented the operational semantics of λ_{Rust} (originating in the RustBelt work (Jung et al. 2018a)), which defines an abstract machine that executes λ_{Rust} programs. We have seen how the abstract machine evaluates programs by repeatedly making transitions. This section gives a precise description of how undefined behavior is modeled in the λ_{Rust} operational semantics.

The per-thread reduction relation $(m_1 \mid e_1) \to (m_2 \mid e_2, e_f^?)$ implicitly specifies when a program has undefined behavior. For example, consider the program ☣$+5$ (executing as one of the threads in some machine configuration). This program is not a value, and therefore it has not terminated yet. Hence, we would like to keep applying per-thread reduction steps to it in order to reduce it to a value. However, none of the rules for per-thread reduction steps apply to this program: the only rule for addition is O-Add, and O-Add can only be applied when adding two integers. Moreover, applying O-ECtx also does not work here, since there are no subexpressions that need to be evaluated. Therefore, this program cannot take a per-thread reduction step. Threads which are not values but nevertheless cannot take a per-thread reduction step (in the current memory of the machine configuration) are called *stuck*.

More formally, a combination $(m_1 \mid e_1)$ of a program e_1 and a memory m_1 is *stuck* if $e_1 \notin \mathit{Val}$ and there does not exist a memory m_2, an expression e_2, and an (optional) thread $e_f^?$ such that $(m_1 \mid e_1) \to (m_2 \mid e_2, e_f^?)$.

Based on the notion of stuckness, we can give a formal definition of undefined behavior: a λ_{Rust} program e has undefined behavior if it is possible to reach a machine configuration $(m' \mid \langle e_1 \mid\mid \ldots \mid\mid e_n \rangle)$ (using $\to_{\mathsf{tp}}^*$, starting from the initial configuration $(\varnothing \mid \langle e \rangle)$) such that for some i, $(m' \mid e_i)$ is stuck. That is, if it is possible to end up in a machine configuration where one of the threads is stuck, then the program has undefined behavior.

As mentioned before, programs that have undefined behavior are allowed to have *arbitrary* behavior. This means that the program does not have to stop running or crash (although it is allowed for it to do so) when implemented on a real machine, despite the intuition that seems to be suggested by the word "stuck". The notion of stuckness is just a formal way of describing when a program has undefined behavior. Moreover, if one of the threads is stuck in some reachable configuration of a program, then the program *in its entirety* has undefined behavior: undefined behavior is not local to the thread that gets stuck.

The main type of undefined behavior that can be caused by memory accesses is accessing (reading, writing, or deallocating) a memory location $\ell \notin \mathsf{dom}(m)$ that is not allocated in the current memory m. This type of undefined behavior corresponds to use-after-free bugs. For example, the

```
1  fn f(x: &mut i32) -> i32 {
2      let mut y = *x;
3
4      // Borrowing (mutable)
5      let y1 = &mut y;
6      // Reborrowing (mutable)
7      let y2 = &mut *y1;
8      // Reborrowing (shared)
9      let y3 = &*y2;
10     // Reading and writing
11     *x = *y3 + 1;
12     *y1 = 10;
13     // Cast to raw pointer
14     let yp = y1 as *mut i32;
15     // More writing
16     unsafe { *yp = 20 };
17     y = 30;
18
19     return y;
20 }
```

$$
\begin{array}{l}
\texttt{rec}\ f([x]) := \\
\quad \texttt{let}\ y = \texttt{alloc}()\ \texttt{in} \\
\quad y := {}^*x; \\
\quad \texttt{let}\ y_1 = y\ \texttt{in} \\
\quad \texttt{let}\ y_2 = y_1\ \texttt{in} \\
\quad \texttt{let}\ y_3 = y_2\ \texttt{in} \\
\quad x := {}^*y3 + 1; \\
\quad y_1 := 10; \\
\quad \texttt{let}\ y_p = y_1\ \texttt{in} \\
\quad y_p := 20; \\
\quad y := 30; \\
\quad \texttt{let}\ r = {}^*y\ \texttt{in} \\
\quad \texttt{free}(y); \\
\quad r
\end{array}
$$

Figure 2.4: A Rust program (left) and its λ_{Rust} counterpart (right).

program

$$\texttt{let}\ x = \texttt{alloc}()\ \texttt{in}\ x := 0; \texttt{free}(x); {}^*x$$

has undefined behavior because of the following possible sequence of thread-pool reductions:

$$
\begin{aligned}
& (\varnothing \mid \langle \texttt{let}\ x = \texttt{alloc}()\ \texttt{in}\ x := 0; \texttt{free}(x); {}^*x \rangle) \\
\to_{\mathsf{tp}} & (\{\ell_1 := \text{☣}\} \mid \langle \texttt{let}\ x = \ell_1\ \texttt{in}\ x := 0; \texttt{free}(x); {}^*x \rangle) \\
\to_{\mathsf{tp}} & (\{\ell_1 := \text{☣}\} \mid \langle \ell_1 := 0; \texttt{free}(\ell_1); {}^*\ell_1 \rangle) \\
\to_{\mathsf{tp}} & (\{\ell_1 := 0\} \mid \langle \text{☣}; \texttt{free}(\ell_1); {}^*\ell_1 \rangle) \\
\to_{\mathsf{tp}} & (\{\ell_1 := 0\} \mid \langle \texttt{free}(\ell_1); {}^*\ell_1 \rangle) \\
\to_{\mathsf{tp}} & (\varnothing \mid \langle \text{☣}; {}^*\ell_1 \rangle) \\
\to_{\mathsf{tp}} & (\varnothing \mid \langle {}^*\ell_1 \rangle)
\end{aligned}
$$

Here, the machine ends up in a state where one of the threads attempts to execute ${}^*\ell_1$ in a memory m where $\ell_1 \notin \mathsf{dom}(m)$. Since the rule O-Mem-Read can only be applied for locations $\ell \in \mathsf{dom}(m)$, the thread is stuck, and therefore the program has undefined behavior.

2.7.4 Translating programs from Rust to λ_{Rust}

This section demonstrates how Rust programs can be translated to λ_{Rust} programs that are (roughly) equivalent. A side-by-side comparison of a Rust program and a corresponding λ_{Rust} program is shown in Fig. 2.4.

The Rust program has function `f` taking a single argument of type `&mut i32`. It declares a variable `y`, which is initialized with the value read from the reference `x`. It then uses borrowing and reborrowing to create a few references: the references `y1`, `y2`, `y3`, and the raw pointer `yp` all refer to the same location: the memory location holding the contents of the variable `y`. The mutable reference `y1` is obtained by borrowing from the variable `y`, the mutable reference `y2` is reborrowed from `y1`, and similarly the shared reference `y3` is reborrowed from `y2`. Several reads and writes to these references and the variable `y` are performed: for example, `*x = *y3 + 1` updates the value stored at the location referred to by `x` to the value read from `y3`, incremented by one. The write using the raw pointer is required to be enclosed in an `unsafe` block.

The λ_{Rust} program is also a function that takes an argument x. This function *explicitly* allocates a memory location for the variable y, which is deallocated near the end of the function f. The λ_{Rust} program performs the initialization by writing ${}^*x$ (the value read from x) to the allocated location. Note that in λ_{Rust}, ${}^*x$ simply means reading from the location x in memory, whereas in Rust, the `*` has a more complicated meaning that depends on the context where it appears.

Whereas in the Rust program, the variable `y` holds an integer directly, in λ_{Rust}, the variable y holds a location, referring to a place in memory where the actual integer is stored. This is because in λ_{Rust}, variables are immutable, and it is only possible to perform mutation by reading and writing to memory locations.

In Rust, `&mut y` creates a (mutable) reference to a value stored in memory. Because the variable y in λ_{Rust} is *already* reference (it holds a location that refers to a value stored in memory, as opposed to holding the value directly), there is no need for a separate operation that "creates" references in λ_{Rust}: we can simply have the variable y_1, y_2, y_3, and y_p hold the same location as y. This means that the constructs for creating references in Rust, such as `&mut` and `&`, do not really show up in λ_{Rust}. There also is not any real difference between mutable references, shared references, and raw pointers in λ_{Rust}: values of type `&mut i32`, `&i32`, and `*mut i32` are simply locations ℓ where $m(\ell)$ is an integer $z \in \mathbb{Z}$. The difference between those types of pointers only matters to the type system, and is not manifested at runtime in λ_{Rust}.

The translation illustrated here is not fully general (in particular, the original RustBelt paper (Jung et al. 2018a) uses a more involved translation capable of dealing with more language constructs, such as indirect control flow transfers in loops), but it works for all the example Rust programs in the rest of this thesis. Moreover, this translation does not use non-atomic memory accesses, which we left out of our description of λ_{Rust}, even though those types of accesses more accurately reflect memory accesses in Rust. The use of non-atomic memory accesses is mostly not relevant for our purposes, since we

only consider programs where non-atomic memory accesses behave exactly the same as the memory accesses described in the operational semantics in Section 2.7.2.

Chapter 3

Stacked Borrows

The previous chapter discussed the Rust language, and introduced the λ_{Rust} language and its operational semantics. This chapter introduces the *Stacked Borrows aliasing model*, a new operational semantics for memory accesses in Rust developed by Jung et al. (2020). The motivation for Stacked Borrows is to enable more compiler optimizations compared to an operational semantics like the one discussed in Section 2.7.2.

The outline of this chapter is as follows: Section 3.1 gives the motivation for Stacked Borrows by discussing a useful optimization that is not allowed by an operational semantics like that of λ_{Rust}, but which is allowed under the Stacked Borrows semantics. The example that we discuss is one of the motivating examples from (Jung et al. 2020). We reproduce the motivation in order to provide the reader with a better understanding of the purpose of Stacked Borrows.

Section 3.2 describes the simplified version of the Stacked Borrows aliasing model that we consider in this thesis, which leaves out some aspects of the full Stacked Borrows model described in (Jung et al. 2020).

Finally, Section 3.3 gives the operational semantics of $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$, which is a variant of λ_{Rust} that incorporates the Stacked Borrows aliasing model. The operational semantics for $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ is based on the original Stacked Borrows paper (Jung et al. 2020), although we use a somewhat different (but equivalent) representation for some parts of the program state.

3.1 Optimizations for references

This section gives an example of a useful compiler optimization on references that cannot be performed according to an operational semantics like the one used for λ_{Rust}. This example is taken directly from the original paper on Stacked Borrows (Jung et al. 2020), where it is used as one of the motivating examples for introducing the Stacked Borrows semantics. We reproduce it here in order to make the motivation for Stacked Borrows more clear, and our

discussion of the example follows the Stacked Borrows paper fairly closely.

In Section 2.7, we have described the λ_{Rust} language and its operational semantics. The semantics of a programming language is particularly important when considering compiler optimizations. A compiler optimization is a transformation applied to a program in order to improve its performance characteristics. Generally, many compiler optimizations are performed to a program during the process of transforming the source program written by the programmer into a form suitable for efficient execution on the actual hardware.

An important property of compiler optimizations is that they should be *behavior-preserving*. The optimized version of program should behave the same as the unoptimized version of a program, because the intent of compiler optimizations is only to improve the performance of a program, and not to change its observable behavior.

Optimizations are not required to be "behavior-preserving" on programs with undefined behavior, since programs with undefined behavior are allowed to have arbitrary behavior in the first place, and therefore compiler implementers have no obligation toward those programs. This means that compiler implementers are effectively allowed to *assume* that programs do not contain undefined behavior while optimizing a program. Being able to make such additional assumptions can allow for more powerful optimizations. It does mean, however, that programs that *violate* those assumptions by causing undefined behavior can be transformed in unexpected ways.

We now consider one of the motivating examples of optimizations from the original paper on Stacked Borrows (Jung et al. 2020). Consider the following two functions:

```
1  fn write_both(x: &mut i32, y: &mut i32) -> i32 {
2      *x = 3;
3      *y = 5;
4      return *x;
5  }
6
7  fn write_both_optimized(x: &mut i32, y: &mut i32) -> i32 {
8      *x = 3;
9      *y = 5;
10     return 3;
11 }
```

The function `write_both` takes two references, writes 3 to the first and 5 to the second, and then reads the value stored in the first reference. Suppose that we know that `x` and `y` do not alias, i.e. they do not point to the same location. In that case, we know that the writes to `x` and `y` are writing to different locations. Hence, the write to `y` does not affect the value stored at `x`. This means that when we go to read `x`, the value stored there will still be 3 (for simplicity, assume here that the program is single-threaded), and therefore we can replace the read of `x` by the value 3, as we have done

in `write_both_optimized`. This optimization removes an unnecessary read operation from the program, replacing it by a constant.

The above optimization relies on the fact that `x` and `y` are known not to alias. The borrow checker ensures that mutable references do not alias, and hence the above optimization would seem to be justified. However, using unsafe code, it is possible to effectively circumvent the borrow checker and obtain two mutable references to the same location in memory:

```
1  fn duplicate_ref() -> i32 {
2      let mut x = 0;
3      let ptr = &mut x as *mut i32;
4
5      let ref1 = unsafe { &mut *ptr };
6      let ref2 = unsafe { &mut *ptr };
7
8      let result = write_both(ref1, ref2);
9      return result;
10 }
```

The function `duplicate_ref` obtains two mutable references to the same location by first casting a mutable reference to a raw pointer using `as *mut i32`, and then converting that back to a mutable reference *twice* using `&mut *ptr`. Since the borrow checker does not track what happens to raw pointers, it does not reject this program.

What is the behavior of `duplicate_ref`? According to a memory model like the one used for λ_{Rust}, `duplicate_ref` always returns the value 5. This is because reads and writes just update locations in memory (according to O-Mem-Read and O-Mem-Write), and the program does not have undefined behavior because it does not access unallocated locations. Hence, if both `x` and `y` in `write_both` point to the same location, then the read from `x` inside `write_both` will read the value that was written using `y`, i.e. the value 5.

This means that replacing `write_both` by `write_both_optimized` is not a valid optimization (according to a naive memory model like the one in λ_{Rust}), because it would change the behavior of `duplicate_ref`, making `duplicate_ref` always return 3 instead of 5.

This example shows that it is not possible (when using an operational semantics similar to λ_{Rust}) to use the aliasing guarantees on mutable references provided by the borrow checker to perform aliasing-related optimizations, because it is possible to circumvent those guarantees using unsafe code. The core issue here is that the λ_{Rust} operational semantics says that `duplicate_ref` should always output 5, while we want to perform an optimization that would make it output 3. This issue can be resolved by declaring programs like `duplicate_ref` (that intuitively violate the aliasing restrictions on mutable references in Rust) to have undefined behavior. In that case, it would be allowed by the semantics for `duplicate_ref` to output 3, since it would be allowed to have any behavior whatsoever.

3.2 Stacked Borrows aliasing model

In Section 3.1, we have seen how certain optimizations are prevented by the λ_{Rust} operational semantics, because the operational semantics does not enforce the aliasing restrictions imposed by the borrow checker. This section describes a variant of the *Stacked Borrows* aliasing model, which adds additional aliasing-related undefined behavior to Rust for the purpose of enabling optimizations like the one described in Section 3.1. The Stacked Borrows aliasing model was introduced in (Jung et al. 2020).

In this thesis, we consider a simplified variant of the full Stacked Borrows model. In particular, we leave out *protectors* and support for *interior mutability*. The original paper on Stacked Borrows describes Stacked Borrows in a step-by-step version, and this simplified version almost exactly corresponds to the version described in Section 3 of (Jung et al. 2020). The only difference compared to the model described in Section 3 of that paper is that we also apply a relaxation of the model described later in the paper, where the rule for reading is relaxed by introducing Disabled items (although we implement that rule without explicit Disabled items).

The basic idea behind Stacked Borrows is to extend the operational semantics of Rust with additional "checks" to ensure that programs do not violate certain restrictions on pointer aliasing, similar to the restrictions enforced by the borrow checker. The Stacked Borrows checks are performed by the abstract machine while the program executes on the abstract machine, and cause programs that violate the aliasing restrictions to become stuck, meaning those programs have undefined behavior. However, unlike the borrow checker, these "checks" are applied to the runtime behavior of the program, and therefore they are also capable of taking into account raw pointers and unsafe code.

While performing program optimizations, it is allowed to *assume* that a program does not have undefined behavior. Therefore, adding the Stacked Borrows checks to the abstract machine allows the compiler to assume that programs do not violate the aliasing restrictions, and this in turn enables more powerful optimizations.

For example, the optimization of replacing `write_both` by `write_both_optimized` is allowed according to the Stacked Borrows aliasing model, because programs like `duplicate_ref` that violate the aliasing restrictions using unsafe code get "caught" by the abstract machine and therefore have undefined behavior.

It is important to note that despite the fact that Stacked Borrows is formulated as a "checker" on the runtime behavior of a program, there is no need to execute those checks while a program executes on a *real* machine (as opposed to an abstract machine), because programs with undefined behavior are not required to crash or produce an error. Hence using the Stacked Borrows semantics does not lead to runtime overhead. The purpose of the Stacked Borrows is mostly to give a precise definition of what constitutes a

"violation of the aliasing restrictions", which can be used to justify compiler optimizations.

However, as a debugging tool, it can still be useful to execute the Stacked Borrows checks even when executing a program on a real machine. For this reason, the Stacked Borrows aliasing model has been implemented in the Miri interpreter (Jung et al. 2020), a program that executes Rust programs and is capable of detecting some types of undefined behavior.

Stacked Borrows rules In order to enforce aliasing restrictions, Stacked Borrows tags each pointer with a *tag*. In the λ_{Rust} memory model, pointer values are simply locations $\ell \in \mathit{Loc}$. In contrast, in the Stacked Borrows aliasing model, a pointer value consists of a pair $\langle \ell, t \rangle$ of a location $\ell \in \mathit{Loc}$ and a tag $t \in \mathit{Tag}$. Pointers are either tagged with a "pointer ID" p (a natural number), or with the special tag $\bot$ ("untagged").

$$\begin{aligned} p \in \mathit{PtrId} &\triangleq \mathbb{N} \\ t \in \mathit{Tag} &::= p \mid \bot \end{aligned}$$

By tagging pointers, Stacked Borrows is able to distinguish between multiple pointers to the same location (i.e., aliasing pointers). Stacked Borrows uses this tag to determine what kind of accesses the pointer is allowed to perform. Stacked Borrows keeps track of the access permissions of each pointer using some additional state: it associates a *borrow stack* (stack, for short) to each allocated memory location, which tracks which tags are currently allowed to access that memory location and what kind of accesses those tags are allowed to perform. Similar to the way the memory is represented, the stacks are stored in a finite, partial mapping from locations to stacks:

$$\xi \in \mathit{Stacks} \triangleq \mathit{Loc} \xrightarrow{\mathsf{fin}} \mathit{Stack}$$

Each stack consists of a list of items, where each item consists of a permission and a tag:

$$\begin{aligned} S \in \mathit{Stack} &\triangleq \mathit{List}(\mathit{Item}) \\ \iota \in \mathit{Item} &::= \mathsf{Unique}(t) \mid \mathsf{SharedRO}(t) \mid \mathsf{SharedRW}(\bot) \end{aligned}$$

There are three kinds of items, and each item grants the permission to perform certain types of memory access to the tag mentioned in the item:

- $\mathsf{Unique}(t)$, which stands for unique mutable access. This item grants read and write access to the tag t.
- $\mathsf{SharedRO}(t)$, which stands for shared read-only access. This item merely grants read access to the tag t.
- $\mathsf{SharedRW}(\bot)$, which stands for shared mutable (read-write) access. This item grants read and write access to the special tag $\bot$.

For example, if the item $\mathsf{Unique}(5)$ appears in the stack for location ℓ, then it is allowed to perform reads and writes to that location using the tagged pointer $\langle \ell, 5 \rangle$. If $\mathsf{Unique}(5)$ does not appear in the stack for location ℓ, then it is not allowed to perform write accesses using $\langle \ell, 5 \rangle$ (although it might be allowed to perform read accesses if $\mathsf{SharedRO}(5)$ *does* appear in the stack).

The ordering of the items in the stack is used to keep track of the order in which references were derived from each other. This means that new items are *added* to the stack when additional references to a location ℓ are created by reborrowing or casting a reference to a raw pointer. Stacked Borrows uses this to ensure that derived references are invalidated when the original reference is used again, similar to the lifetime restrictions enforced by the borrow checker. Stacked Borrows invalidates pointers (disallowing them from being used again) by *removing* items from the stack.

Items are mostly added to and removed from the stacks in a LIFO (last-in, first-out) order, which is why borrow stacks are called stacks. That is, when a new item is added, it is generally added to the front of the list (the front of the list represents the top of the stack) and when items are removed they are generally removed at the front of the list. Adding items to the front of the list is called *pushing*, and removing items from the front is called *popping*.

For example, creating new mutable references to a location ℓ by reborrowing causes additional $\mathsf{Unique}(t)$ items to be pushed onto the stack for location ℓ. Writing to the location ℓ using the tagged pointer $\langle \ell, 5 \rangle$ while the item $\mathsf{Unique}(5)$ appears in the stack for location ℓ causes all items above the $\mathsf{Unique}(5)$ to be popped from the stack, effectively invalidating all references derived from $\langle \ell, 5 \rangle$.

Having explained the general outline of the Stacked Borrows rules, we now present the Stacked Borrows semantics, describing precisely how the stacks are affected when allocation, reading, writing, deallocation, and pointer creation (by borrowing, reborrowing or casting a reference to a raw pointer) occur. Pointer creation is called *retagging*, since it involves generating a new tag and adding corresponding items to the stack. The location of a pointer remains unchanged during retagging.

The Stacked Borrows semantics is closely inspired by the lifetime rules in the Rust type system, as discussed in (Jung et al. 2020). Specifically, the Rust type system only allows a reference to be used during its lifetime. Similarly, Stacked Borrows only allows a pointer to be used while its tag occurs in the stack with the right permission. The reader might find it useful to keep this correspondence in mind while reading the semantics.

The rules of the semantics are as follows:

- Allocation: When allocating a new location ℓ, initialize the stack for location ℓ to $[\mathsf{Unique}(t)]$ for some freshly generated tag t, and return the tagged pointer $\langle \ell, t \rangle$.

- Writing: When writing using the pointer $\langle \ell, t \rangle$, find the topmost item

in the stack for location ℓ that grants write access to t. Pop *all* items above the granting item from the stack. If there is no granting item, the program has undefined behavior.

- Reading: When reading using the pointer $\langle \ell, t \rangle$, find the topmost item in the stack for location ℓ that grants read access to t. Remove all Unique items above the granting item (SharedRO and SharedRW items are not removed). If there is no granting item, the program has undefined behavior.

- Deallocation: When deallocating the location ℓ using the pointer $\langle \ell, t \rangle$, find the topmost item in the stack for location ℓ that grants write access to t. Then, remove the stack for location ℓ from the mapping holding the stacks. If there is no granting item, the program has undefined behavior.

- Retagging: When deriving a new pointer from the pointer $\langle \ell, t_{\text{old}} \rangle$:

  - If the new pointer is a *mutable reference*, perform the effects of a *write* access with tag t_{old} on the stack for location ℓ. Next, push a new item Unique(t_{new}) on top of the stack for location ℓ, where t_{new} is a freshly generated tag. Return the tagged pointer $\langle \ell, t_{\text{new}} \rangle$.
  - If the new pointer is a *shared reference*, perform the effects of a *read* access with tag t_{old} on the stack for location ℓ. Next, push the item SharedRO(t_{new}) on top of the stack, where t_{new} is a freshly generated tag. Return the tagged pointer $\langle \ell, t_{\text{new}} \rangle$.
  - If the new pointer is a *raw pointer* being created from a reference using a cast (e.g., `myref as *mut i32`), then perform the effects of a *write* access with tag t_{old} on the stack for location ℓ. Next, push a new item SharedRW($\perp$) on top of the stack for location ℓ. Return the tagged pointer $\langle \ell, \perp \rangle$.

- Functions arguments of type `&mut T` for some type `T` are implicitly retagged as mutable references at the start of the function, and function arguments of type `&T` are implicitly retagged as shared references at the start of the function. This ensures that references passed as arguments to functions receive fresh tags and is necessary in general to justify optimizations. In the full Stacked Borrows model, there are more cases like this, but this case is the only one relevant to the examples in this thesis.

Example (no undefined behavior) We now show an example of how the stack for a location changes during execution of a typical Rust program. We do this by showing the state of the stack for a particular location in a comment between the lines, and we use a comment on the side to show the

tag of each newly-created pointer (we also name the variables after the tags to emphasize which tag each pointer has). Consider the following example:

```
fn sb_example_defined() {
    let mut x0 = 42; // Tag: 0
    // [Unique(0)]
    let x1 = &mut x0; // Tag: 1
    // [Unique(1), Unique(0)]
    let x2 = &mut *x1; // Tag: 2
    // [Unique(2), Unique(1), Unique(0)]
    *x2 = 10;
    // [Unique(2), Unique(1), Unique(0)]
    *x1 = 5;
    // [Unique(1), Unique(0)]
    x0 = 3;
    // [Unique(0)]
    let x3 = &x0; // Tag: 3
    // [SharedRO(3), Unique(0)]
    let x4 = &x0; // Tag: 4
    // [SharedRO(4), SharedRO(3), Unique(0)]
    x0 = 5;
    // [Unique(0)]
}
```

This program shows how additional Unique items are added to the stack for the location holding the variable x0 as mutable references are created. Writing to the location causes all items above the granting item to be removed, invalidating the pointers corresponding to those items. The program also shows that it is possible to create multiple shared references without invalidating any of the existing shared references. The shared references are only invalidated once a write occurs. This is very similar to the lifetime rules in Rust.

Example (undefined behavior) The following is a simple example (based on the motivating example in Section 3.1) of a program that has undefined behavior according to Stacked Borrows:

```
fn sb_example_undefined() {
  let mut x0 = 42; // Tag: 0
  // [Unique(0)]
  let x1 = &mut x0; // Tag: 1
  // [Unique(1), Unique(0)]
  let xraw = x1 as *mut i32; // Tag: _
  // [SharedRW(_), Unique(1), Unique(0)]
  let x2 = unsafe { &mut *xraw }; // Tag: 2
  // [Unique(2), SharedRW(_), Unique(1), Unique(0)]
  let x3 = unsafe { &mut *xraw }; // Tag: 3
  // N.B.: Unique(2) gets popped!
  // [Unique(3), SharedRW(_), Unique(1), Unique(0)]
  *x2 = 10; // Undefined behavior! Tag 2 does not have write access
  *x3 = 20;
}
```

This program creates a mutable reference `x1` to the variable `x0`, and casts that mutable reference to a raw pointer `xraw`. Subsequently the program derives *two* (aliasing) mutable references from the raw pointer: when deriving a mutable reference from the raw pointer with tag $\perp$, all items above the granting item for $\perp$ are removed. This means that when the second mutable reference is created, the item $\mathsf{Unique}(2)$ is removed, which effectively means that `x2` can no longer be used after `x3` has been created. Hence, the program has undefined behavior because it violates the Stacked Borrows rules by writing to `x2` after `x3` has been created. In this way, Stacked Borrows ensures that programs like the above, which create aliasing mutable references using raw pointers, do not have to be considered while optimizing.

3.3 $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$: extending λ_{Rust} with Stacked Borrows

This section gives a formal operational semantics $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ ("lambda-Rust Stacked"), which is a version of λ_{Rust} that has been extended with the Stacked Borrows aliasing model. We only describe the changes with respect to the operational semantics for λ_{Rust} given in Section 2.7.2. The general style of the operational semantics given here is the same as in the original description of Stacked Borrows (Jung et al. 2020), although there are some minor implementation differences: in particular, we use recursively-defined functions on lists instead of functions that explicitly manipulate list indices in order to describe changes to borrow stacks. Using recursively-defined functions makes it simpler to perform proofs by induction on lists.

Since, pointers in Stacked Borrows are tagged, the syntax of values $v \in \mathit{Val}$ is different: instead of

$$v \in \mathit{Val} ::= \ell \mid \dots$$

we have

$$v \in \mathit{Val} ::= \langle \ell, t \rangle \mid \dots$$

Moreover, the language is extended with a *retagging instruction*:

$$\begin{aligned} r \in \mathit{RetagKind} &::= \texttt{\&mut} \mid \texttt{\&} \mid \texttt{*mut} \\ e \in \mathit{Expr} &::= \texttt{retag}[r](e) \mid \dots \\ K \in \mathit{ECtx} &::= \bullet \mid \texttt{retag}[r](K) \mid \dots \end{aligned}$$

The retagging instruction derives a new pointer $\langle \ell, t_{\text{new}} \rangle$ (for some t_{new}) from another pointer $\langle \ell, t_{\text{old}} \rangle$: the derived pointer is assigned a new tag and the stack for the location is updated accordingly. The *retag kind* $r \in \mathit{RetagKind}$ indicates what kind of pointer is being derived: a mutable reference (`&mut`), a shared reference (`&`), a mutable raw pointer (`*mut`). Retagging corresponds to reborrowing in Rust.

For λ_{Rust}, the machine configuration consisted of the memory $m \in \mathit{Mem}$, combined with the list of threads. Stacked Borrows keeps track of a borrow

stack for each memory location, and therefore the machine configuration now also keeps track of the borrow stacks $\xi \in \mathit{SBStacks}$, as well as a counter $p \in \mathit{PtrId}$ that can be incremented for generating new tags:

$$
\begin{aligned}
p \in \mathit{PtrId} &\triangleq \mathbb{N} \\
t \in \mathit{Tag} &::= p \mid \bot \\
\iota \in \mathit{Item} &::= \mathsf{Unique}(t) \\
&\quad \mid \mathsf{SharedRO}(\{t_1, \ldots, t_n\}) \\
&\quad \mid \mathsf{SharedRW}(\bot) \\
S \in \mathit{Stack} &\triangleq \mathit{List}(\mathit{Item}) \\
\xi \in \mathit{SBStacks} &\triangleq \mathit{Loc} \xrightarrow{\mathsf{fin}} \mathit{Stack} \\
\varsigma \in \mathit{SBState} &\triangleq \mathit{PtrId} \times \mathit{SBStacks} \\
\sigma \in \mathit{ProgramState} &\triangleq \mathit{Mem} \times \mathit{SBState} \\
c \in \mathit{Config} &\triangleq \mathit{ProgramState} \times \mathit{List}(\mathit{Expr})
\end{aligned}
$$

The formal operational semantics uses a slightly different representation for SharedRO items than described in Section 3.2 (the representation in that section is based on the original Stacked Borrows paper (Jung et al. 2020)): adjacent SharedRO items are grouped into a set, which simplifies reasoning about the semantics. That is, the stack $[\mathsf{SharedRO}(2), \mathsf{SharedRO}(1), \mathsf{Unique}(0)]$ is written as $[\mathsf{SharedRO}(\{1, 2\}), \mathsf{Unique}(0)]$ instead. This representation also corresponds to the intuitive "set-like" way in which adjacent SharedRO items are treated: their relative ordering does not matter (this is easy to verify by looking at each of the rules in the semantics).

Now we describe the changes to the per-thread reduction relation $(m_1 \mid e_1) \to (m_2 \mid e_2, e_f^?)$. The semantics of operations which do not access or affect the memory are left essentially unchanged: the only change to the rules O-Add, O-App, O-Case, O-Fork, and O-ECtx is that the state now consists of a (m, ς) pair of a memory $m \in \mathit{Mem}$ and the Stacked Borrows state $\varsigma \in \mathit{SBState}$, which is left unchanged by each of those operations. The memory model, consisting of the rules O-Mem-Alloc, O-Mem-Read, O-Mem-Write, and O-Mem-Free is replaced with new rules given in Fig. 3.1. These rules have additional premises that are responsible for updating the borrow stacks and "checking" that the Stacked Borrows rules are respected.

The additional premises related to Stacked Borrows are of the form $\varsigma \xrightarrow{\varepsilon} \varsigma'$, which is a transition relation that describes how the Stacked Borrows state changes from ς to ς' when the *Stacked Borrows event* ε occurs. The purpose of the event is to indicate what kind of operation is being performed, and this determines which of the Stacked Borrows rules should apply. The syntax

O-Mem-SB-Alloc

$$\frac{\ell \notin \mathsf{dom}(m) \qquad \varsigma \xrightarrow{\mathsf{SBAlloc}(\langle \ell, t \rangle)} \varsigma'}{((m, \varsigma) \mid \texttt{alloc}()) \to ((m[\ell \leftarrow \text{☣}], \varsigma') \mid \langle \ell, t \rangle)}$$

O-Mem-SB-Read

$$\frac{\ell \in \mathsf{dom}(m) \qquad \varsigma \xrightarrow{\mathsf{SBRead}(\langle \ell, t \rangle)} \varsigma'}{((m, \varsigma) \mid {}^*\langle \ell, t \rangle) \to ((m, \varsigma') \mid m(\ell))}$$

O-Mem-SB-Free

$$\frac{\ell \in \mathsf{dom}(m) \qquad \varsigma \xrightarrow{\mathsf{SBFree}(\langle \ell, t \rangle)} \varsigma'}{((m, \varsigma) \mid \texttt{free}(\langle \ell, t \rangle)) \to ((m \setminus \ell, \varsigma') \mid \text{☣})}$$

O-Mem-SB-Write

$$\frac{\ell \in \mathsf{dom}(m) \qquad \varsigma \xrightarrow{\mathsf{SBWrite}(\langle \ell, t \rangle)} \varsigma'}{((m, \varsigma) \mid \langle \ell, t \rangle := v) \to ((m[\ell \leftarrow v], \varsigma') \mid \text{☣})}$$

O-Mem-SB-Retag

$$\frac{\varsigma \xrightarrow{\mathsf{SBRetag}(r, \langle \ell, t_{\text{old}} \rangle, t_{\text{new}})} \varsigma'}{((m, \varsigma) \mid \texttt{retag}[r](\langle \ell, t_{\text{old}} \rangle)) \to ((m, \varsigma') \mid \langle \ell, t_{\text{new}} \rangle)}$$

Figure 3.1: Operational semantics for memory-related operations under Stacked Borrows.

of events is as follows:

$$\begin{aligned}\varepsilon \in \mathit{SBEvent} ::= {} & \mathsf{SBAlloc}(\langle \ell, t \rangle) \\ \mid {} & \mathsf{SBFree}(\langle \ell, t \rangle) \\ \mid {} & \mathsf{SBWrite}(\langle \ell, t \rangle) \\ \mid {} & \mathsf{SBRead}(\langle \ell, t \rangle) \\ \mid {} & \mathsf{SBRetag}(r, \langle \ell, t_{\text{old}} \rangle, t_{\text{new}})\end{aligned}$$

The Stacked Borrows transition relation $\varsigma \xrightarrow{\varepsilon} \varsigma'$ also has the notion of being "stuck", which describes the situation where it is not possible to derive the premise $\varsigma \xrightarrow{\varepsilon} \varsigma'$. In that case, the thread executing the operation that causes the event also gets stuck, and hence the program has undefined behavior.

Each of the rules in Fig. 3.1 has a premise with an event corresponding to the operation (allocation, reading, writing, deallocation, or retagging) that the program is performing, and the parameters of the event are (partially) determined by the operation: for example, writing to $\langle \ell, t \rangle$ "causes" an $\mathsf{SBWrite}(\langle \ell, t \rangle)$ event. However, for some events not all of the event parameters are determined by the program: for example, when allocating, the tag is not determined by the program, but it is determined based on the current Stacked Borrows state. Hence, in an event such as $\mathsf{SBAlloc}(\langle \ell, t \rangle)$, t is more properly viewed as an "output" of Stacked Borrows. Similarly, the tag t_{new} for $\mathsf{SBRetag}(r, \langle \ell, t_{\text{old}} \rangle, t_{\text{new}})$ is also not determined by the program.

The effect of the events is determined by the rules in Fig. 3.2. The rule E-SB-Alloc for allocation requires that the location does not already have a stack ($\ell \notin \mathsf{dom}(\xi)$), and produces the tagged pointer $\langle \ell, p \rangle$ (shown in the event), where the tag p is freshly generated by incrementing the counter in the Stacked Borrows state. The stack for location ℓ is initialized to $[\mathsf{Unique}(p)]$.

The rules E-SB-Read, E-SB-Write, and E-SB-Free deal with reading, writing, and deallocation events, respectively. The rules E-SB-Retag-Mutable, E-SB-Retag-Shared, E-SB-Retag-Raw deal with the retagging for mutable references, shared reference, and mutable raw pointers, respectively. All of these rules have a similar shape: they each look up the stack for the location ℓ under consideration using $\xi(\ell) = S$. Then they apply an auxiliary function to that stack (the functions have names including the word Single since they are applied to single stacks), which either produces a new stack S' or *fails* by producing the special value **fail**. The purpose of that function is to add or remove items from the stack S in accordance with the Stacked Borrows rules, and the function can produce **fail** to indicate a Stacked Borrows violation.

If the function did not produce **fail**, the new stack S' is stored in the Stacked Borrows state using $\xi[\ell \leftarrow S']$ (or the stack is removed, in the case of deallocation). Furthermore, the rules for retagging always generate a fresh tag by incrementing the counter, which is passed to the auxiliary function for that rule if the function requires a fresh tag.

E-SB-Alloc

$$\frac{\ell \notin \mathsf{dom}(\xi)}{(p,\xi) \xrightarrow{\mathsf{SBAlloc}(\langle \ell, p \rangle)} (p+1, \xi[\ell \leftarrow [\mathsf{Unique}(p)]])}$$

E-SB-Write

$$\frac{\xi(\ell) = S \qquad \mathsf{WriteSingle}(t, S) = S'}{(p,\xi) \xrightarrow{\mathsf{SBWrite}(\langle \ell, t \rangle)} (p, \xi\big[\ell \leftarrow S'\big])}$$

E-SB-Read

$$\frac{\xi(\ell) = S \qquad \mathsf{ReadSingle}(t, S) = S'}{(p,\xi) \xrightarrow{\mathsf{SBRead}(\langle \ell, t \rangle)} (p, \xi\big[\ell \leftarrow S'\big])}$$

E-SB-Free

$$\frac{\xi(\ell) = S \qquad \mathsf{WriteSingle}(t, S) = S'}{(p,\xi) \xrightarrow{\mathsf{SBFree}(\langle \ell, t \rangle)} (p, \xi \setminus \ell)}$$

E-SB-Retag-Mutable

$$\frac{\xi(\ell) = S \qquad \mathsf{RetagUniqueSingle}(t_{\text{old}}, p, S) = S'}{(p,\xi) \xrightarrow{\mathsf{SBRetag}(\texttt{\&mut}, \langle \ell, t_{\text{old}} \rangle, p)} (p+1, \xi\big[\ell \leftarrow S'\big])}$$

E-SB-Retag-Shared

$$\frac{\xi(\ell) = S \qquad \mathsf{RetagSharedROSingle}(t_{\text{old}}, p, S) = S'}{(p,\xi) \xrightarrow{\mathsf{SBRetag}(\texttt{\&}, \langle \ell, t_{\text{old}} \rangle, p)} (p+1, \xi\big[\ell \leftarrow S'\big])}$$

E-SB-Retag-Raw

$$\frac{\xi(\ell) = S \qquad \mathsf{RetagSharedRWSingle}(t_{\text{old}}, S) = S'}{(p,\xi) \xrightarrow{\mathsf{SBRetag}(\texttt{*mut}, \langle \ell, t_{\text{old}} \rangle, \bot)} (p+1, \xi\big[\ell \leftarrow S'\big])}$$

Figure 3.2: Operational semantics for Stacked Borrows events.

$$\begin{aligned}
\mathsf{WriteSingle} &: \mathit{Tag} \times \mathit{Stack} \to \mathit{Stack} \uplus \{\mathbf{fail}\} \\
\mathsf{WriteSingle}(t, []) &= \mathbf{fail} \\
\mathsf{WriteSingle}(t, \iota :: \bar{\iota}) &= \iota :: \bar{\iota} \quad (\text{if } \mathsf{ItemGrantsWrite}\ \iota\ t) \\
\mathsf{WriteSingle}(t, \iota :: \bar{\iota}) &= \mathsf{WriteSingle}(t, \bar{\iota})
\end{aligned}$$

$$\begin{aligned}
\mathsf{ReadSingle} &: \mathit{Tag} \times \mathit{Stack} \to \mathit{Stack} \uplus \{\mathbf{fail}\} \\
\mathsf{ReadSingle}(t, []) &= \mathbf{fail} \\
\mathsf{ReadSingle}(t, \iota :: \bar{\iota}) &= \iota :: \bar{\iota} \quad (\text{if } \mathsf{ItemGrantsRead}\ \iota\ t) \\
\mathsf{ReadSingle}(t, \mathsf{Unique}(t') :: \bar{\iota}) &= \mathsf{ReadSingle}(t, \bar{\iota}) \\
\mathsf{ReadSingle}(t, \iota :: \bar{\iota}) &= \iota :: \mathsf{ReadSingle}(t, \bar{\iota})
\end{aligned}$$

$$\begin{aligned}
\mathsf{RetagUniqueSingle} &: \mathit{Tag} \times \mathit{Tag} \times \mathit{Stack} \to \mathit{Stack} \uplus \{\mathbf{fail}\} \\
\mathsf{RetagUniqueSingle}(t_{\text{old}}, t_{\text{new}}, S) &= \mathsf{Unique}(t_{\text{new}}) :: \mathsf{WriteSingle}(t_{\text{old}}, S)
\end{aligned}$$

$$\begin{aligned}
\mathsf{RetagSharedRWSingle} &: \mathit{Tag} \times \mathit{Stack} \to \mathit{Stack} \uplus \{\mathbf{fail}\} \\
\mathsf{RetagSharedRWSingle}(t_{\text{old}}, S) &= \mathsf{SharedRW}(\bot) :: \mathsf{WriteSingle}(t_{\text{old}}, S)
\end{aligned}$$

$$\begin{aligned}
\mathsf{PushSharedRO} &: \mathit{Tag} \times \mathit{Stack} \to \mathit{Stack} \\
\mathsf{PushSharedRO}(t_{\text{new}}, \mathsf{SharedRO}(\{t_1, \ldots, t_n\}) :: \bar{\iota}) &= \mathsf{SharedRO}(\{t_{\text{new}}, t_1, \ldots, t_n\}) :: \bar{\iota} \\
\mathsf{PushSharedRO}(t_{\text{new}}, \bar{\iota}) &= \mathsf{SharedRO}(\{t_{\text{new}}\}) :: \bar{\iota}
\end{aligned}$$

$$\begin{aligned}
\mathsf{RetagSharedROSingle} &: \mathit{Tag} \times \mathit{Tag} \times \mathit{Stack} \to \mathit{Stack} \uplus \{\mathbf{fail}\} \\
\mathsf{RetagSharedROSingle}(t_{\text{old}}, t_{\text{new}}, S) &= \mathsf{PushSharedRO}(t_{\text{new}}, \mathsf{ReadSingle}(t_{\text{old}}, S))
\end{aligned}$$

Figure 3.3: Auxiliary functions describing how the stack for a location is updated. The first case that matches from top to bottom determines the output of the function, and failure propagates (see text).

ItemGrantsWrite-Unique
ItemGrantsWrite Unique(t) t

ItemGrantsWrite-SharedRW
ItemGrantsWrite SharedRW($\bot$) $\bot$

ItemGrantsRead-Unique
ItemGrantsRead Unique(t) t

ItemGrantsRead-SharedRW
ItemGrantsRead SharedRW($\bot$) $\bot$

ItemGrantsRead-SharedRO
$$\frac{t \in \{t_1, \ldots, t_n\}}{\text{ItemGrantsRead SharedRO}(\{t_1, \ldots, t_n\})\ t}$$

Figure 3.4: Permissions assigned to tags by stack items.

The definitions for the auxiliary functions are shown in Fig. 3.3. We use some convenient notation in order to make the definitions less verbose: the equations should be read from *top to bottom*, where the first equation that applies to the given inputs determines the output of the function. Furthermore, we assume that failure *propagates*, meaning that if a occurrence of a function on the right-hand side of an equation produces **fail**, then the entire right-hand side should become **fail**. This notation helps to reduce the number of cases that need to be explicitly written out, but it would also be possible to write out all cases explicitly using more traditional mathematical notation.

Several functions are defined recursively on lists: they distinguish between the empty list [] and a non-empty list $\iota :: \bar{\iota}$ consisting of a single item ι at the front of the list followed by the rest of the list $\bar{\iota}$, and are applied recursively to the rest of the list in some cases. It is easy to see that these recursive definitions are well-founded, since the recursive occurrences of each function are applied to structurally smaller lists.

The auxiliary functions use the ItemGrantsWrite (respectively ItemGrantsRead) judgments shown in Fig. 3.4 to determine whether an item grants write (respectively read) permissions to a tag.

The function WriteSingle specifies how the stack for a location changes during a write access: it removes items from the top of the stack (by recursing over the structure of the stack) until it encounters an item that grants write access to the given tag t (as indicated by ItemGrantsWrite ι t). If it does not encounter a write-granting item, then it produces **fail**, indicating that the program has undefined behavior.

Similarly, the function ReadSingle specifies the effect of read accesses on the stack: remove Unique items from the stack (SharedRO and SharedRW items are left in the stack) until it encounters a read-granting item (according to ItemGrantsRead ι t) for the tag t. Note that a Unique item can also be a read-granting item, hence the case for ItemGrantsRead can also match a Unique, which will then not be removed. If it does not find a read-granting

```
1  fn f(x: &mut i32) -> i32 {
2      let mut y = *x;
3
4      // Borrowing (mutable)
5      let y1 = &mut y;
6      // Reborrowing (mutable)
7      let y2 = &mut *y1;
8      // Reborrowing (shared)
9      let y3 = &*y2;
10     // Reading and writing
11     *x = *y3 + 1;
12     *y1 = 10;
13     // Cast to raw pointer
14     let yp = y1 as *mut i32;
15     // More writing
16     unsafe { *yp = 20 };
17     y = 30;
18
19     return y;
20 }
```

$$
\begin{array}{l}
\texttt{rec}\ f([x]) := \\
\quad \texttt{let}\ x = \texttt{retag}[\texttt{\&mut}](x)\ \texttt{in} \\
\quad \texttt{let}\ y = \texttt{alloc}()\ \texttt{in} \\
\quad y := {}^{*}x; \\
\quad \texttt{let}\ y_1 = \texttt{retag}[\texttt{\&mut}](y)\ \texttt{in} \\
\quad \texttt{let}\ y_2 = \texttt{retag}[\texttt{\&mut}](y_1)\ \texttt{in} \\
\quad \texttt{let}\ y_3 = \texttt{retag}[\texttt{\&}](y_2)\ \texttt{in} \\
\quad x := {}^{*}y_3 + 1; \\
\quad y_1 := 10; \\
\quad \texttt{let}\ y_p = \texttt{retag}[\texttt{*mut}](y_1)\ \texttt{in} \\
\quad y_p := 20; \\
\quad y := 30; \\
\quad \texttt{let}\ r = {}^{*}y\ \texttt{in} \\
\quad \texttt{free}(y); \\
\quad r
\end{array}
$$

Figure 3.5: A Rust program (left) and its $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ counterpart (right).

item, it produces **fail**.

The functions RetagUniqueSingle and RetagSharedRWSingle specify the effect of creating a mutable reference and casting to a raw pointer, respectively: they apply the effects of a write access with t_{old} to the stack and push a new item on top of the stack. Similarly, RetagSharedROSingle gives the effect of creating a shared reference or a read-only raw pointer: it applies the effects of a *read* access with t_{old} to the stack and pushes a new SharedRO item, which is added into the existing set of SharedRO items where possible (using the function PushSharedRO).

3.4 Translating programs from Rust to $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$

This section gives a short example of how to translate a simple Rust program into a similar $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ program. The only addition to the translation illustrated in Section 2.7.4 consists of the insertion of retagging instructions in the proper places: when creating references by borrowing or reborrowing, when casting references to raw pointers, and when receiving references as function arguments. Again, we only illustrate the translation using an example, and the translation as illustrated here is not meant to cover all Rust language features.

Consider the two programs shown in Fig. 3.5. The program on the left is a Rust program which has been translated into the $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ program on the right. Retagging instructions have been inserted where borrowing, reborrowing, or

casting to a raw pointer occurs. Furthermore, the arguments of type `&mut T` or `&T` are retagged at the start of the function. Just as in Section 2.7.4, the pointers y_1, y_2, y_3, and y_p all refer to the same location, but now they have different tags, and therefore the difference between the types `&mut i32`, `&i32`, and `*mut i32` is now manifested at runtime as well: the types determine which kind of retagging instructions are used, and this also determines what kind of stack items are added to the borrow stacks.

Chapter 4

Separation Logic for Stacked Borrows

In Chapter 2, we have introduced the Rust language, describing how the Rust type system applies the notions of ownership and borrowing to ensure the absence of issues such as use-after-free bugs (leading to undefined behavior) in the majority of Rust code. The RustBelt project formalized these safety guarantees for the λ_{Rust} language based on a simplified version of the Rust type system that includes the core notions of ownership, borrowing, and lifetimes. Specifically, the RustBelt project shows that well-typed λ_{Rust} programs (corresponding to safe Rust programs, i.e. programs that do not contain `unsafe` blocks and only make use of safe-to-use libraries) do not cause any undefined behavior according to the λ_{Rust} operational semantics.

The RustBelt safety proof is built on top of a program logic for reasoning about the behavior of λ_{Rust} programs. However, this logic is not sound for $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$, since $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ has a different semantics with more undefined behavior. Therefore, we have developed a new, sound logic for reasoning about $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ programs, which we call *Stacked Borrows Separation Logic* (SBSL). Part of the motivation behind this logic is to provide a path for updating the RustBelt safety proof to account for Stacked Borrows, although the program logic is also useful in its own right for reasoning about the correctness (in particular, the absence of undefined behavior) of programs written in $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$.

Just like the logic used for RustBelt, SBSL is based on *concurrent separation logic* (Brookes 2007; O'Hearn 2007; O'Hearn et al. 2001; Reynolds 2002), a family of program logics that simplify reasoning about the behavior of programs with concurrency and mutable state. Specifically, SBSL builds on top of the Iris concurrent separation logic framework (Jung et al. 2016, 2018b, 2015; Krebbers et al. 2017a), a modern variant of concurrent separation logic. We will explain the basic concepts of separation logic and Iris throughout this chapter.

We build up to the SBSL logic in several steps. In Section 4.1, we discuss

$$
\begin{aligned}
\tau &::= \mathit{Val} \mid \mathit{Expr} \mid \mathit{Tag} \mid \mathit{Loc} \mid \mathbb{Z} \mid \dots \\
s &::= x \mid v \mid e \mid t \mid \ell \mid n \mid \dots \\
P, Q &::= P \land Q \\
&\mid P \lor Q \\
&\mid P \Rightarrow Q \\
&\mid \neg P \\
&\mid \mathsf{True} \\
&\mid \mathsf{False} \\
&\mid s = s \\
&\mid \exists x{:}\tau. P \\
&\mid \forall x{:}\tau. P \\
&\mid \{P\}\, e\, \{v.\, Q\} \\
&\mid P * Q \\
&\mid \ell \mapsto v \\
&\mid P \Rrightarrow Q \\
&\mid \dots
\end{aligned}
$$

Figure 4.1: The syntax of Iris types, terms and propositions.

the logic that is used in RustBelt (Jung et al. 2018a) for reasoning about λ_{Rust} programs. In Section 4.2, we describe the *physical stack assertion*, which is a naive way of extending the λ_{Rust} program logic to reason about $\lambda_{\mathsf{Rust}}^{\mathsf{SB}}$ programs. In Section 4.3, we discuss some shortcomings of the physical stack assertion, showing that the naive logic is too sensitive to irrelevant implementation details, which makes it unsuitable for reasoning about simple forms of concurrency.

In Section 4.4, we start introducing the rules of Stacked Borrows Separation Logic, a separation logic for reasoning about $\lambda_{\mathsf{Rust}}^{\mathsf{SB}}$ programs, and which does not suffer from the shortcomings of the naive approach described in Section 4.2. This logic is based on the notions of *ghost stacks* and *ghost forgetting*, which allow for more abstract reasoning about $\lambda_{\mathsf{Rust}}^{\mathsf{SB}}$ programs, in turn enabling better reasoning about concurrency. These notions are the main conceptual contribution of this thesis.

The rules of SBSL are introduced in a step-by-step fashion: Section 4.4 describes the rules for mutable references, Section 4.5 describes the rules for raw pointers, and Section 4.6 describe the rules for shared references. In each section, we illustrate the rules using simple examples.

Finally, we briefly discuss the connection between SBSL and the Rust type system in Section 4.7.

4.1 Concurrent separation logic

This section describes the concurrent separation logic used for reasoning about λ_{Rust} programs in the RustBelt project (Jung et al. 2018a). The logic is built on top of the Iris concurrent separation logic (Jung et al. 2016, 2018b, 2015; Krebbers et al. 2017a), which is a generic framework for building program logics.

The syntax of Iris is shown in Fig. 4.1. Iris is an intuitionistic higher-order logic, and therefore it includes all of the usual logical connectives, such as conjunction, disjunction, negation, implication, equality, and universal and existential quantification. It also includes some additional connectives, which will be introduced as required. Some connectives shown in Fig. 4.1 are not primitive notions in Iris, but are instead defined in terms of more basic logical primitives.

The main connective used in Iris for reasoning about programs is the *Hoare triple*. Hoare triples were first introduced in Hoare logic (Hoare 1969). Hoare triples are written

$$\{P\}\, e\, \{v.\, Q\}$$

and consist of a *precondition* P, a *program* e, and a *postcondition* Q, where P and Q are propositions. The meaning of such a Hoare triple is that when the program e is executed in an initial state satisfying P, then the program e executes *safely* (without undefined behavior) and if it terminates, it does so with a value v such that Q is satisfied in the final state. Note that the variable v may appear in Q in order to refer to the result of the computation and we can omit v when it does not appear in Q. A Hoare triple gives a *specification* of the behavior of a program, and proving that a Hoare triple holds shows that a program behaves (operationally) according to its specification.

A simple example of a valid Hoare triple for λ_{Rust} is the following:

$$\{\mathsf{True}\}\, 2 + (3 + 5)\, \{v.\, v = 10\}$$

expressing that the program $2 + (3 + 5)$ does not cause undefined behavior regardless of the initial state (indicated by the precondition True), and when it terminates, it does so with the value 10. It is easy to see based on the intuitive definitions of Hoare triples and the operational semantics of λ_{Rust} that this Hoare triple holds.

A Hoare triple $\{P\}\, e\, \{v.\, Q\}$ does not imply termination of the program e: therefore, these Hoare triples guarantee only *partial correctness*, meaning that a state satisfying Q might never be reached, as opposed to *total correctness*, where termination is also guaranteed.

Separation logic (O'Hearn et al. 2001; Reynolds 2002) is an extension of Hoare logic that adds the *points-to assertion* $\ell \mapsto v$ and the *separating conjunction* $P * Q$. Separation logic is intended to allow reasoning about programs that use pointers to mutable memory locations. The intuitive

reading of $\ell \mapsto v$ is that in the current state, the memory location ℓ holds the value v. The intuitive reading of the separating conjunction $P * Q$ is that the propositions P and Q hold for *disjoint* parts of the program state. Therefore, $P * Q$ is generally (but not always) stronger than $P \wedge Q$, which has the usual meaning that P and Q both hold (but not necessarily for disjoint parts of the state).

When the program state simply consists of a memory $m \in \mathit{Mem}$ (as in λ_{Rust}), then disjoint parts of the program state are simply non-overlapping parts of memory. That is, a memory is split into disjoint parts by splitting its domain: for example, the memory $m = \{\ell_1 := 2, \ell_2 := 3\}$ can be split into $m_1 = \{\ell_1 := 2\}$ and $m_2 = \{\ell_2 := 3\}$. This means that points-to assertions on either sides of a separating conjunction cannot have the same location:

$$\ell_1 \mapsto v_1 * \ell_2 \mapsto v_2 \Rightarrow \ell_1 \neq \ell_2$$

because disjoint parts of memory cannot contain the same location. In particular, this implies

$$\ell \mapsto v * \ell \mapsto v' \Rightarrow \mathsf{False}$$

and thus the following does *not* hold

$$\ell \mapsto v \Rightarrow \ell \mapsto v * \ell \mapsto v$$

That is, given a $\ell \mapsto v$, we cannot obtain two separate copies of it: the $\ell \mapsto v$ proposition cannot be "duplicated". This means that the proposition $\ell \mapsto v$ is often viewed as expressing ownership of a *resource*, instead of simply expressing a *fact* or *knowledge*. This gives rise to the *ownership reading* (O'Hearn 2007) of $\ell \mapsto v$: the proposition $\ell \mapsto v$ expresses exclusive ownership (where ownership is the right to access) of the location ℓ, which holds the value v.

Iris has a general mechanism for defining resources, and resources do not necessarily have to correspond to parts of the program state directly. We will see further examples of resources later in this chapter. This means that $P * Q$ can be read more generally as: "P and Q hold for disjoint resources".

Based on the points-to assertion, we can state Hoare triples about programs that use pointers to access memory. For instance, the following is a valid Hoare triple:

$$\{\ell \mapsto 0\}\ \ell := 1\ \{\ell \mapsto 1\}$$

We can read this Hoare triple as: if we are given exclusive ownership of $\ell \mapsto 0$ (a location ℓ that holds the value 0 in the current state), then the write $\ell := 1$ executes without undefined behavior and ends in a state where we have exclusive ownership of $\ell \mapsto 1$ (a location ℓ that holds the value 1).

The notion of ownership in separation logic enables *local reasoning* about programs in the presence of pointers and mutable state: parts of a program

that operate on disjoint parts of the state do not affect each other indirectly, and can thus be reasoned about in isolation. The local reasoning principles of separation logic are captured by rules such as the *frame rule*:

$$\frac{\{P\}\, e\, \{v.\, Q\}}{\{P * R\}\, e\, \{v.\, Q * R\}}$$

This states that if we have proved a Hoare triple for the program e with precondition P and postcondition Q, then the same Hoare triple will still hold in the presence of additional disjoint resources R. This means that program specifications only need to mention those parts of the program state that are actually accessed or modified by the program e: using the frame rule, it is possible to generalize a Hoare triple to a larger program state.

The use of the separating conjunction is crucial in the frame rule. If we replace the separating conjunction by an ordinary conjunction, then the frame rule is no longer sound, as we can derive conclusions such as

$$\frac{\dfrac{\{\ell \mapsto 2\}\, \ell := 3\, \{\ell \mapsto 3\}}{\{\ell \mapsto 2 \land \ell \mapsto 2\}\, \ell := 3\, \{\ell \mapsto 3 \land \ell \mapsto 2\}}}{\{\ell \mapsto 2\}\, \ell := 3\, \{\ell \mapsto 2\}}$$

which states that if ℓ holds 2 initially ($\ell \mapsto 2 \land \ell \mapsto 2$ is implied by $\ell \mapsto 2$ since $P \Rightarrow P \land P$), then after $\ell := 3$, ℓ will still hold 2. For the separating conjunction, we do *not* have $\ell \mapsto 2 \Rightarrow \ell \mapsto 2 * \ell \mapsto 2$, preventing a similar line of reasoning.

Separation logic rules for λ_{Rust} Having introduced the basic notions of separation logic, we show the separation logic rules for reasoning about λ_{Rust} programs in Fig. 4.2. Some additional derived rules are shown in Fig. 4.3. These rules are part of the program logic developed for λ_{Rust} in the RustBelt project (Jung et al. 2018a).

The most interesting rules for our purposes are those related to memory accesses: H-Mem-Alloc, H-Mem-Write, H-Mem-Read, and H-Mem-Free. Allocation produces a location ℓ for which we obtain a points-to assertion $\ell \mapsto \lightning$ in the postcondition. Deallocation requires a points-to assertion $\ell \mapsto v$, which no longer appears in the postcondition. That is, we give up ownership of $\ell \mapsto v$ when deallocating, which prevents accessing the location ℓ after deallocation. Moreover, reading and writing both require a points-to assertion (the superscript q in H-Mem-Read can be ignored for now), which still appears in the postcondition (with an updated value in case of writing).

The rule H-Value is a fairly trivial rule that can be applied when the program is already a value. The rule H-Add can be used to reason about addition of two numbers. The rule H-If allows proving a Hoare triple for an if-expression by proving the Hoare triple for each of the branches separately

H-Mem-Alloc
$\{\mathsf{True}\}\ \mathtt{alloc}()\ \{\ell.\ \ell \mapsto \text{☣}\}$

H-Mem-Write
$\{\ell \mapsto v\}\ \ell := v'\ \{w.\ (w = \text{☣}) * \ell \mapsto v'\}$

H-Mem-Read
$\{\ell \overset{q}{\mapsto} v\}\ {}^*\ell\ \{w.\ (w = v) * \ell \overset{q}{\mapsto} v\}$

H-Mem-Free
$\{\ell \mapsto v\}\ \mathtt{free}(\ell)\ \{w.\ w = \text{☣}\}$

PointsTo-Split
$\ell \overset{q_1+q_2}{\longmapsto} v \iff \ell \overset{q_1}{\longmapsto} v * \ell \overset{q_2}{\longmapsto} v$

H-Frame
$$\frac{\{P\}\ e\ \{v.\ Q\}}{\{P * R\}\ e\ \{v.\ Q * R\}}$$

H-Value
$\{\mathsf{True}\}\ w\ \{v.\ v = w\}$

H-Add
$\{\mathsf{True}\}\ z_1 + z_2\ \{z.\ z = z_1 + z_2\}$

H-If
$$\frac{\{P * (b = \mathtt{true})\}\ e_1\ \{v.\ Q\} \qquad \{P * (b = \mathtt{false})\}\ e_2\ \{v.\ Q\}}{\{P * (b \in \{\mathtt{true}, \mathtt{false}\})\}\ \mathtt{if}\ b\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ \{v.\ Q\}}$$

H-App
$$\frac{\{P\}\ e[\overline{v}/\overline{x}]\ \{w.\ Q\}}{\{P\}\ (\mathtt{rec}\ \_(\overline{x}) := e)(\overline{v})\ \{w.\ Q\}}$$

H-ECtx
$$\frac{\{P\}\ e\ \{u.Q\} \qquad \forall v.\ \{Q[v/u]\}\ K[\,v\,]\ \{w.\ R\}}{\{P\}\ K[\,e\,]\ \{w.\ R\}}$$

H-Fork
$$\frac{\{P\}\ e\ \{\mathsf{True}\}}{\{P\}\ \mathtt{fork}\{\ e\ \}\ \{v.\ v = \text{☣}\}}$$

H-Conseq
$$\frac{P \Rightarrow P' \qquad \{P'\}\ e\ \{v.\ Q'\} \qquad \forall v.\ Q' \Rightarrow Q \qquad (P \Rightarrow P' \text{ persistent}) \qquad (\forall v. Q' \Rightarrow Q \text{ persistent})}{\{P\}\ e\ \{v.\ Q\}}$$

H-Exists
$$\frac{(x \text{ not free in } e \text{ and } Q) \qquad \forall x.\{P\}\ e\ \{v.\ Q\}}{\{\exists x.\ P\}\ e\ \{v.\ Q\}}$$

H-Eq
$$\frac{\{P[s/r]\}\ e[s/r]\ \{v.Q[s/r]\}}{\{r = s * P\}\ e\ \{v.Q\}}$$

Figure 4.2: Separation logic rules for reasoning about λ_{Rust} programs.

H-Seq
$$\frac{\{P\}\, e_1\, \{Q\} \qquad \{Q\}\, e_2\, \{v.\, R\}}{\{P\}\, e_1; e_2\, \{v.\, R\}}$$

H-Let
$$\frac{\{P\}\, e_1\, \{v.\, Q\} \qquad \forall v. \{Q\}\, e_2[v/x]\, \{w.\, R\}}{\{P\}\, \texttt{let}\ x = e_1\ \texttt{in}\ e_2\, \{w.\, R\}}$$

H-Par
$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\}}{\{P_1 * P_2\}\, e_1 \mid\mid e_2\, \{Q_1 * Q_2\}}$$

Figure 4.3: Derived rules for reasoning about λ_{Rust} programs.

(the rule for `case` is a straightforward generalization of this rule). The rule H-App allows reasoning about a (non-recursive) function applied to some argument values by reasoning about the *body* of the function, where the argument values have been substituted for the parameters. For simplicity, we do not present the rule for recursive functions here.

The rule H-ECtx can be used when an expression $K[\,e\,]$ contains a subexpression e that is evaluated first. According to the operational semantics, in an expression of the form $K[\,e\,]$, the subexpression e is first evaluated to a value v before proceeding with the evaluation of $K[\,v\,]$. The rule H-ECtx captures this by allowing us to prove an intermediate postcondition Q for the subexpression e, and then letting us reason about $K[\,v\,]$ where v is the value resulting from e.

We can reason about a sequence $e_1; e_2$ of expressions using the H-Seq rule. This rule requires the postcondition of e_1 and the precondition of e_2 to be the same, and therefore we typically have to apply the *rule of consequence* H-Conseq, which allows replacing the precondition P' by a stronger precondition P ($P \Rightarrow P'$) and the postcondition Q' by a weaker postcondition Q ($\forall v.\, Q' \Rightarrow Q$). This also allows performing reasoning inside preconditions and postconditions. The rule of consequence has a technical side-condition, requiring the propositions $P \Rightarrow P'$ and $\forall v.\, Q' \Rightarrow Q$ to be *persistent*. This side condition can mostly be ignored due to our particular presentation of the rules of the program logic. [1]

The rules H-Exists and H-Eq, when read from bottom to top, allow us

[1] Requiring the implications to be persistent rules out implications that express ownership of resources, like the proposition $\mathsf{True} \Rightarrow \ell \mapsto v$ that "contains" a $\ell \mapsto v$ resource that we can get by applying modus ponens with True. The side condition is necessary to avoid resources like $\ell \mapsto v$ from being duplicated. Due to our presentation of the rules, implications like $\mathsf{True} \Rightarrow \ell \mapsto v$ cannot appear outside the pre- or postconditions of Hoare triples, which also ensures that we cannot violate the side condition.

to eliminate equalities and existentials occurring in preconditions of a Hoare triple while proving a Hoare triple. These rules are necessary in order to use equalities $r = s$ and existentials $\exists x{:}\tau.\, P$ occurring in the precondition. Equalities appear in the postconditions of rules such as H-Mem-Read, and rewriting using an equality also replaces occurrences in the program e. We will also see rules that have existentials in their postcondition.

Based on the rules discussed so far, we can derive Hoare triples about more complicated λ_{Rust} programs. We will generally not be focusing on programs that contain loops (indeed, we did not show the rule for recursive functions) or branches, since we are mainly interested in reasoning about memory accesses. For example, we can derive the following Hoare triple:

$$\{\mathsf{True}\}\, \mathtt{let}\, x = \mathtt{alloc}()\, \mathtt{in}\, (x := 0); (x := {}^*x + 1); \mathtt{free}(x)\, \{\mathsf{True}\}$$

The program in this Hoare triple allocates a memory location ℓ, writes 0 to it, increments the value stored in the location by 1 (by reading and then writing), and finally frees the location. Proving this Hoare triple shows that the program does not have undefined behavior, regardless of the initial state. The postcondition is trivial, meaning that we do not show any interesting fact about the behavior of this program aside from the fact that it does not have undefined behavior.

Since derivations are generally quite large, we generally show derivations by merely annotating the program with the intermediate pre- and postconditions Q for rules like H-Seq and H-Let, and being implicit about applications of rules such as H-Frame, H-Conseq, etc. Based on this, it is generally easy to see how to derive the Hoare triple. For example, the derivation of the above Hoare triple can be shown as follows:

$$\begin{array}{l}
\{\mathsf{True}\} \\
\mathtt{let}\, x = \mathtt{alloc}()\, \mathtt{in} \\
\{\ell_x \mapsto \not{\mathsf{W}}\} \\
x := 0; \\
\{\ell_x \mapsto 0\} \\
x := {}^*x + 1; \\
\{\ell_x \mapsto 1\} \\
\mathtt{free}(x) \\
\{\mathsf{True}\}
\end{array}$$

Here, we have applied the convention of writing ℓ_x for the location produced by the allocation and bound to the variable x. Note that by annotating programs in this way, we can see that the separation logic rules are essentially "symbolically" or "logically" executing the program.

Concurrency The local reasoning principles enabled by the separating conjunction, as enshrined in rules like the frame rule, also lead to powerful

principles for reasoning about *concurrent* programs. This is because threads that operate on disjoint parts of the program state do not affect each other indirectly, and means that we can reason about each thread in isolation. This is the main idea behind concurrent separation logic (Brookes 2007; O'Hearn 2007), which applies the ownership reading of the points-to assertion to give rules for reasoning about concurrent programs. For example, we have the rule H-Par for reasoning about the *parallel composition* of two programs e_1 and e_2:

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\}}{\{P_1 * P_2\}\, e_1 \mid\mid e_2\, \{Q_1 * Q_2\}}$$

The parallel composition construct $e_1 \mid\mid e_2$ executes e_1 and e_2 concurrently, waiting for both expressions to terminate, and discards their results. Parallel composition is not a primitive operation in λ_{Rust}, but instead it is implemented in terms of `fork` and the "compare-and-set" instruction which allows threads to communicate. We do not discuss the implementation here, merely showing the rule for reasoning about it. The H-Par rule can be derived from H-Fork (discussed below), but doing so requires more advanced Iris techniques that we do not discuss here.

The parallel composition rule allows reasoning about e_1 and e_2 independently, as long as they operate on disjoint resources P_1 and P_2. We obtain the resources Q_1 and Q_2 produced by both threads in the postcondition. An example application of this rule is as follows:

$$\frac{\{\ell_1 \mapsto 2\}\, \ell_1 := 3\, \{\ell_1 \mapsto 3\} \qquad \{\ell_2 \mapsto 3\}\, \ell_2 := 4\, \{\ell_2 \mapsto 4\}}{\{\ell_1 \mapsto 2 * \ell_2 \mapsto 3\}\, \ell_1 := 3 \mid\mid \ell_2 := 4\, \{\ell_1 \mapsto 3 * \ell_2 \mapsto 4\}}$$

Here, we have one thread which writes to the location ℓ_1 and another that writes to the location ℓ_2. The separating conjunction implicitly encodes that these locations are not the same, and therefore we can reason about the behavior of each thread in isolation. Effectively, the rule forces us to decide which of the threads receives ownership of which points-to assertions. This means that $\ell \mapsto v$ can also be read as "the current *thread* has exclusive ownership of the location ℓ (holding the value v)."

We also have the more primitive rule H-Fork. This rule lets us (logically) pass resources P to the newly-created thread, but it has a trivial postcondition because the fork operation does not wait for the thread to terminate, and therefore we cannot get resources "back" from the forked-off thread.

Just as for sequential programs, we can show derivations by annotating a program with intermediate pre- and postconditions, as in the following

example:

$$
\begin{array}{l}
\{\mathsf{True}\} \\
\texttt{let}\ x = \texttt{alloc}()\ \texttt{in} \\
\{\ell_x \mapsto \text{☣}\} \\
\texttt{let}\ y = \texttt{alloc}()\ \texttt{in} \\
\{\ell_x \mapsto \text{☣} * \ell_y \mapsto \text{☣}\} \\
x := 0; \\
\{\ell_x \mapsto 0 * \ell_y \mapsto \text{☣}\} \\
y := 0; \\
\{\ell_x \mapsto 0 * \ell_y \mapsto 0\} \\
\begin{array}{l|l}
\{\ell_x \mapsto 0\} & \{\ell_y \mapsto 0\} \\
x := {}^*x + 1 & y := {}^*y + 1 \\
\{\ell_x \mapsto 1\} & \{\ell_y \mapsto 1\}
\end{array} \\
\{\ell_x \mapsto 1 * \ell_y \mapsto 1\} \\
\texttt{free}(y); \\
\{\ell_x \mapsto 1\} \\
\texttt{free}(x) \\
\{\mathsf{True}\}
\end{array}
$$

Here, we put programs side-by-side separated by vertical lines to indicate parallel composition. We can also see how the resources $P_1 * P_2$ are divided across threads: the thread on the left gets the $\ell_x \mapsto 0$ and the thread on the right gets the $\ell_y \mapsto 0$. After the parallel composition, the postconditions of both threads are combined.

Fractional permissions The rules as we have presented them thus far do not allow two threads to access the same memory location. This is because $\ell \mapsto v$ expresses exclusive ownership: it is not possible to duplicate the $\ell \mapsto v$ resource into $\ell \mapsto v * \ell \mapsto v$. In general, allowing this duplication is not sound, as we saw for the frame rule.

However, the rules can be slightly relaxed in order to allow different threads to concurrently access the same location, as long as all threads only *read* from that location, and do not perform writes. As long as threads only read from shared locations, we can still reason about threads in isolation. This is because threads cannot "interfere" with each other merely by reading from memory.

The notion of *fractional permissions* (Bornat et al. 2005; Boyland 2003) allows for this. The idea of fractional permissions is to allow a points-to assertion $\ell \mapsto v$ to be split into *fractional* points-to assertions $\ell \overset{q}{\mapsto} v$, where $q \in (\mathbb{Q} \cap (0, 1])$ is a fraction between 0 (exclusive) and 1 inclusive. This is done using the rule PointsTo-Split:

$$\ell \overset{q_1+q_2}{\mapsto} v \iff \ell \overset{q_1}{\mapsto} v * \ell \overset{q_2}{\mapsto} v$$

where we $\ell \mapsto v$ is simply an abbreviation for $\ell \overset{1}{\mapsto} v$.

The fraction q in $\ell \overset{q}{\mapsto} v$ indicates the *degree of ownership*. When $q = 1$, the points-to assertion indicates exclusive ownership, meaning there are no other threads with access to the location ℓ. When $q < 1$, the points-to assertion indicates fractional ownership, meaning that there might be other threads with access to the same location. The rules of the logic do not allow writing or deallocation using a fractional points-to assertion. Only reading (using H-Mem-Read) is allowed, because that merely requires a fractional points-to assertion. The fractions can also be thought of as percentages: $q = 1$ means 100% ownership, whereas $q < 1$ means less than 100% ownership. The specific fraction $q < 1$ does not matter for the H-Mem-Read rule.

The fractional points-to assertion allows reasoning about λ_{Rust} programs that have two threads concurrently reading the same location:

$$
\begin{array}{l}
\{\mathsf{True}\} \\
\texttt{let}\ x = \texttt{alloc}()\ \texttt{in} \\
\{\ell_x \mapsto \not{\circledast}\} \\
x := 0; \\
\{\ell_x \mapsto 0\} \\
\text{(rule of consequence)} \\
\{\ell_x \overset{1/4}{\longmapsto} 0 * \ell_x \overset{3/4}{\longmapsto} 0\} \\
\begin{array}{l|l}
\{\ell_x \overset{1/4}{\longmapsto} 0\} & \{\ell_x \overset{3/4}{\longmapsto} 0\} \\
{}^*x & {}^*x \\
\{\ell_x \overset{1/4}{\longmapsto} 0\} & \{\ell_x \overset{3/4}{\longmapsto} 0\}
\end{array} \\
\{\ell_x \overset{1/4}{\longmapsto} 0 * \ell_x \overset{3/4}{\longmapsto} 0\} \\
\text{(rule of consequence)} \\
\{\ell_x \mapsto 0\} \\
\texttt{free}(x) \\
\{\mathsf{True}\}
\end{array}
$$

Here we split the points-to assertion $\ell_x \mapsto 0$ into fractions $\ell_x \overset{1/4}{\longmapsto} 0$ and $\ell_x \overset{3/4}{\longmapsto} 0$, which are passed to the two different threads using H-Par. Again, the postconditions for both threads are combined according to H-Par.

4.2 Physical stack assertion

The separation logic presented in Section 4.1 allows reasoning about λ_{Rust} programs, and in particular we can show that a λ_{Rust} program e does not have undefined behavior by deriving a Hoare triple $\{\mathsf{True}\}\, e\, \{\mathsf{True}\}$.

However, $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ has a different operational semantics from λ_{Rust}. Pointers in $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ are tagged, and this tag is used to determine whether a particular memory access constitutes undefined behavior according to the borrow stack for that location. In addition, in $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$, there is an additional retagging operation that is not in λ_{Rust}. The separation logic for λ_{Rust} does not account for these stacks and tags at all, and it does not have a rule for the retagging

operation. Therefore, it obviously cannot be used to reason about $\lambda_{\mathsf{Rust}}^{\mathsf{SB}}$ programs.

Hence, in order to reason about $\lambda_{\mathsf{Rust}}^{\mathsf{SB}}$ programs, and in particular to show the absence of undefined behavior, the separation logic of Section 4.1 needs to be extended to account for Stacked Borrows.

In this section, we describe a naive extension of the logic in Section 4.1 that accounts for Stacked Borrows. This serves as an intermediate step toward Stacked Borrows Separation Logic, which is discussed in Section 4.4, because as we will see the naive extension is too weak to reason about certain programs.

As a first attempt to give separation logic rules for $\lambda_{\mathsf{Rust}}^{\mathsf{SB}}$, we introduce an assertion that is similar to the points-to assertion. For λ_{Rust}, the program state merely consists of a memory $m \in \mathit{Mem}$, which is a finite, partial map from locations to values. In $\lambda_{\mathsf{Rust}}^{\mathsf{SB}}$, the program state additionally keeps track of a borrow stack for each location. These stacks are stored in a finite, partial map $\xi \in \mathit{SBStacks}$ from locations to borrow stacks. Hence, the map holding the borrow stacks can be viewed as a special kind of memory that holds borrow stacks instead of ordinary values.

For the ordinary values, we have the fractional points-to assertion $\ell \overset{q}{\mapsto} v$ that keeps track of the value for each location. Hence, for the borrow stacks, we introduce a similar assertion called the *physical stack assertion* $\ell \overset{q}{\mapsto}_{\mathbf{p}} S$, which simply keeps track of the current borrow stack $S \in \mathit{Stack}$ of each location. The idea is that if we know what the current stack of the location ℓ is in the logic, then we can easily determine whether an access is allowed and what happens to the stack after the access. The reason why it is called the *physical* stack assertion is that the term "physical state" is often used to refer to the actual program state appearing in the operational semantics, and hence we refer to the borrow stacks in the operational semantics as *physical (borrow) stacks*. The reason for this distinction will become clearer in Section 4.4, where we introduce "ghost stacks".

The physical stack assertion inherits all the properties of the points-to assertion: $q = 1$ means the stack can be modified, while $q < 1$ merely provides read-only access. Moreover, the physical stack assertion can also be split:

$$\ell \overset{q_1+q_2}{\mapsto}_{\mathbf{p}} S \iff \ell \overset{q_1}{\mapsto}_{\mathbf{p}} S * \ell \overset{q_2}{\mapsto}_{\mathbf{p}} S$$

Based on the idea that $\ell \overset{q}{\mapsto}_{\mathbf{p}} S$ keeps track of the stack S for each location ℓ, we can formulate separation logic rules for the operations of allocation, deallocation, reading, writing, and retagging. These rules are shown in Fig. 4.4. These rules can be viewed as replacements for the memory-related rules for λ_{Rust} presented in Section 4.1: the rules of the λ_{Rust} separation logic which are not related to the memory (e.g., addition, if-expressions, function application) are left unchanged, since the operational semantics for those operations is the same in $\lambda_{\mathsf{Rust}}^{\mathsf{SB}}$ and λ_{Rust}.

H-SBP-Alloc
$\{\mathsf{True}\}\ \texttt{alloc}()\ \{v.\ \exists \ell, t.\ (v = \langle \ell, t\rangle) * \ell \mapsto \text{☣} * \ell \mapsto_{\mathbf{p}} [\mathsf{Unique}(t)]\}$

H-SBP-Write
$\{\ell \mapsto v * \ell \mapsto_{\mathbf{p}} S * \mathsf{WriteSingle}(t, S) = S'\}$
$\langle \ell, t\rangle := v'$
$\{w.\ (w = \text{☣}) * \ell \mapsto v' * \ell \mapsto_{\mathbf{p}} S'\}$

H-SBP-Read
$\{\ell \mapsto v * \ell \mapsto_{\mathbf{p}} S * \mathsf{ReadSingle}(t, S) = S'\}$
$^*\langle \ell, t\rangle$
$\{w.\ (w = v) * \ell \mapsto v * \ell \mapsto_{\mathbf{p}} S'\}$

H-SBP-Read-Fract
$\{\ell \overset{q}{\mapsto} v * \ell \overset{q}{\mapsto}_{\mathbf{p}} S * \mathsf{ReadSingle}(t, S) = S\}$
$^*\langle \ell, t\rangle$
$\{w.\ (w = v) * \ell \overset{q}{\mapsto} v * \ell \overset{q}{\mapsto}_{\mathbf{p}} S\}$

H-SBP-Free
$\{\ell \mapsto v * \ell \mapsto_{\mathbf{p}} S * \mathsf{WriteSingle}(t, S) = S'\}$
$\texttt{free}(\langle \ell, t\rangle)$
$\{w.\ w = \text{☣}\}$

H-SBP-Retag-Mutable
$\{\ell \mapsto_{\mathbf{p}} S * \mathsf{WriteSingle}(t_{\text{old}}, S) = S'\}$
$\texttt{retag}[\texttt{\&mut}](\langle \ell, t_{\text{old}}\rangle)$
$\{w.\ \exists t_{\text{new}}.\ (w = \langle \ell, t_{\text{new}}\rangle) * \ell \mapsto_{\mathbf{p}} (\mathsf{Unique}(t_{\text{new}}) :: S')\}$

H-SBP-Retag-Raw
$\{\ell \mapsto_{\mathbf{p}} S * \mathsf{WriteSingle}(t_{\text{old}}, S) = S'\}$
$\texttt{retag}[\texttt{*mut}](\langle \ell, t_{\text{old}}\rangle)$
$\{w.\ (w = \langle \ell, \bot\rangle) * \ell \mapsto_{\mathbf{p}} (\mathsf{SharedRW}(\bot) :: S')\}$

H-SBP-Retag-Shared
$\{\ell \mapsto_{\mathbf{p}} S * \mathsf{ReadSingle}(t_{\text{old}}, S) = S'\}$
$\texttt{retag}[\texttt{\&}](\langle \ell, t_{\text{old}}\rangle)$
$\{w.\ \exists t_{\text{new}}.\ (w = \langle \ell, t_{\text{new}}\rangle) * \ell \mapsto_{\mathbf{p}} \mathsf{PushSharedRO}(t_{\text{new}}, S')\}$

Figure 4.4: Rules for deriving Hoare triples based on the physical stack assertion.

The rules in Fig. 4.4 are based directly on the operational semantics for $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$, and in particular on the semantics for Stacked Borrows. In describing these rules, we mainly focus on the physical stack assertions, since the points-to assertions in the rules are the same as in the logic for λ_{Rust}.

According to the semantics for allocation, if we allocate a new memory location ℓ, then the stack for that location is initialized to $[\mathsf{Unique}(t)]$ for some tag t. This is reflected in H-SBP-Alloc, because we receive a physical stack assertion $\ell \mapsto_{\mathbf{p}} [\mathsf{Unique}(t)]$ in the postcondition, where t is an existentially quantified tag.

The rule H-SBP-Write indicates that when writing to the location ℓ using a tagged pointer $\langle \ell, t \rangle$, exclusive ownership of the points-to assertion $\ell \mapsto v$ and the physical stack assertion $\ell \mapsto_{\mathbf{p}} S$ for that location is required. This is because writing potentially changes the value and stack associated to a location. Moreover, we have to show in the precondition that $\mathsf{WriteSingle}(t, S) = S'$ for some stack S'. Recall that $\mathsf{WriteSingle}$ is the function used in the semantics to determine whether an access is allowed and what effect the access has on the stack. For example,

$$\mathsf{WriteSingle}(1, [\mathsf{Unique}(2), \mathsf{Unique}(1), \mathsf{Unique}(0)]) = [\mathsf{Unique}(1), \mathsf{Unique}(0)]$$

indicates that when the current stack is $[\mathsf{Unique}(2), \mathsf{Unique}(1), \mathsf{Unique}(0)]$, a write access using tag 1 is allowed and the stack is updated to $[\mathsf{Unique}(1), \mathsf{Unique}(0)]$. In contrast,

$$\mathsf{WriteSingle}(1, [\mathsf{Unique}(0)]) = \mathbf{fail}$$

indicates that writing using tag 1 is not allowed when the stack is $[\mathsf{Unique}(0)]$, because the tag 1 does not appear in the stack. Requiring that $\mathsf{WriteSingle}(t, S) = S'$ therefore ensures that the access is allowed (does not cause undefined behavior), and simultaneously gives us the new stack S' for the location ℓ. Hence, in the postcondition, the physical stack assertion in the postcondition is $\ell \mapsto_{\mathbf{p}} S'$, reflecting the updated stack.

An example instance of H-SBP-Write is as follows:

$$\begin{array}{l}\{\ell \mapsto 0 * \ell \mapsto_{\mathbf{p}} [\mathsf{Unique}(t_2), \mathsf{Unique}(t_1), \mathsf{Unique}(t_0)]\} \\ \quad \langle \ell, t_1 \rangle := 1 \\ \{w.\ (w = \textcircled{\scriptsize ✓}) * \ell \mapsto 1 * \ell \mapsto_{\mathbf{p}} [\mathsf{Unique}(t_1), \mathsf{Unique}(t_0)]\}\end{array}$$

where the $\mathsf{WriteSingle}$ precondition is not shown because it can be derived from the precondition using the rule of consequence.

The other rules are similar to H-SBP-Write: they each use the semantics to determine whether an access is allowed and what effect the access has on the stack for location ℓ. One rule deserves special attention: H-SBP-Read-Fract. This rule can be applied to reason about programs that perform concurrent reads, since it only requires *fractional* assertions. However, it has the fairly strict requirement $\mathsf{ReadSingle}(t, S) = S$, meaning that the stack is

not allowed to be changed by a concurrent read. This restriction is applied because the physical stack assertion, like the points-to assertion, does not allow for changes to the stack with only fractional ownership. However, as we shall see in Section 4.3, this makes the rules for reading rather weak and encourages us to abandon the physical stack assertion to enable more relaxed rules for reading.

To make the rules more concrete, we apply them to show the absence of undefined behavior in the following fragment of Rust code, translated into $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$:

```
1  let mut x0 = 42; // Tag: 0
2  // [Unique(0)]
3  let x1 = &mut x0; // Tag: 1
4  // [Unique(1), Unique(0)]
5  let xraw = x1 as *mut i32; // Tag: _
6  // [SharedRW(_), Unique(1), Unique(0)]
7  unsafe { *xraw = 10 };
8  // [SharedRW(_), Unique(1), Unique(0)]
9  x0 = 20;
10 // [Unique(0)]
```

The comments indicate how the stack for the location holding the contents of `x0` changes during execution. In fact, this already constitutes a proof sketch that the program is free from undefined behavior, since we can easily verify using the operational semantics that the annotated stacks are correct (up to different generated tags) and provide the right access permissions to each tag. Indeed, the derivation we now give closely mirrors it. This is the $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ equivalent of the program, annotated with preconditions and postconditions based on the rules in Fig. 4.4:

$$
\begin{array}{l}
\{\mathsf{True}\} \\
\texttt{let}\ x_0 = \texttt{alloc}()\ \texttt{in} \\
\{\ell_x \mapsto \not{\mathsf{x}} * \ell \mapsto_{\mathbf{p}} [\mathsf{Unique}(t_0)]\} \\
x_0 := 42; \\
\{\ell_x \mapsto 42 * \ell \mapsto_{\mathbf{p}} [\mathsf{Unique}(t_0)]\} \\
\texttt{let}\ x_1 = \texttt{retag}[\texttt{\&mut}](x_0)\ \texttt{in} \\
\{\ell_x \mapsto 42 * \ell \mapsto_{\mathbf{p}} [\mathsf{Unique}(t_1), \mathsf{Unique}(t_0)]\} \\
\texttt{let}\ x_\perp = \texttt{retag}[\texttt{*mut}](x_1)\ \texttt{in} \\
\{\ell_x \mapsto 42 * \ell \mapsto_{\mathbf{p}} [\mathsf{SharedRW}(\perp), \mathsf{Unique}(t_1), \mathsf{Unique}(t_0)]\} \\
x_\perp := 10; \\
\{\ell_x \mapsto 10 * \ell \mapsto_{\mathbf{p}} [\mathsf{SharedRW}(\perp), \mathsf{Unique}(t_1), \mathsf{Unique}(t_0)]\} \\
x_0 := 20; \\
\{\ell_x \mapsto 20 * \ell \mapsto_{\mathbf{p}} [\mathsf{Unique}(t_0)]\} \\
\texttt{free}(x_0) \\
\{\mathsf{True}\}
\end{array}
$$

Here, we can see that the physical stack assertion simply tracks the stack precisely (aside from the tags, which are existentially quantified and therefore

we do not know what the precise tags are), and it is easy to verify that each step in this proof can be derived using the rules in Fig. 4.4. The names of the variables have been chosen to reflect the tags: for example, the variable x_0 is replaced by the pointer $\langle \ell_x, t_0 \rangle$, x_1 is replaced by $\langle \ell_x, t_1 \rangle$, etc. Variables are replaced by their values during the proof of the Hoare triple based on the H-Let rule, and the tag of each location appears existentially quantified in rules such as H-SBP-Retag-Mutable and H-SBP-Alloc.

Just like the program logic for λ_{Rust} in Section 4.1, these proof rules allow programs to be "symbolically" executed by simply keeping track of the stacks. Note that the $\mathsf{WriteSingle}(t, S) = S'$ preconditions are not shown in the pre- and postconditions here: this is because those preconditions can easily be derived using the rule of consequence if the specific stacks are known.

4.3 Abstraction and concurrency

The separation logic for Stacked Borrows based on the physical stack assertion $\ell \overset{q}{\mapsto}_{\mathbf{p}} S$ works for reasoning about some simple $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ programs: it ensures the absence of undefined behavior by requiring in the precondition that an access is allowed according to the current stack for the location. However, the physical stack assertion has a major shortcoming: it leads to Hoare triple specifications that are too concrete, in the sense that they contain too many irrelevant details about how functions are implemented. Instead, we would like Hoare triples to be sufficiently abstract, describing mainly the externally visible behavior of a function, and not too many details about how the function works internally. By keeping Hoare triples sufficiently abstract, we can give the same specification to functions with a different implementation, allowing those functions to be interchanged without affecting the correctness proof of a larger program that uses those functions.

We will now demonstrate the lack of abstraction in the program logic rules based on the physical stack connective. The lack of abstraction also makes the rules weak for reasoning about concurrency, for which we also give an example.

Abstraction As an example, consider the following two Rust functions:

```
1  fn f1(x0: &mut i32) {
2      *x0 = 10;
3  }
4
5  fn f2(x0: &mut i32) {
6      let x1 = &mut *x0;
7      *x1 = 10;
8  }
```

Their $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ counterparts are as follows:

$$
\begin{array}{l}
\texttt{rec}\ f_1([x_0]) := \\
\quad \texttt{let}\ x_1 = \texttt{retag}[\texttt{\&mut}](x_0)\ \texttt{in} \\
\quad x_1 := 10
\end{array}
$$

$$
\begin{array}{l}
\texttt{rec}\ f_2([x_0]) := \\
\quad \texttt{let}\ x_1 = \texttt{retag}[\texttt{\&mut}](x_0)\ \texttt{in} \\
\quad \texttt{let}\ x_2 = \texttt{retag}[\texttt{\&mut}](x_1)\ \texttt{in} \\
\quad x_2 := 10
\end{array}
$$

Note the presence of the additional initial retagging operation on the argument x_0 in both functions: these are inserted implicitly for reference arguments (mutable references or shared references) passed to a function, as described in Section 3.2.

The behavior of f_1 and f_2 is essentially the same: they both write the value 10 to the mutable reference x_0. However, f_1 reborrows x_0 once before doing so, whereas f_2 reborrows x_0 twice. We would like to be able to use these functions interchangeably: intuitively, exactly how many reborrows a function performs in this case is an implementation detail, since all of the additional references (x_1, x_2) created inside the functions do not *escape* the function: they are not returned from the function or stored in external data structures. The additional references are merely used inside the function, and after the function returns those internally-created references will never be used again.

However, these functions do have a different effect on the physical stacks, and this is reflected in their Hoare triple specifications if we naively apply the rules of Fig. 4.4. For example, suppose that f_1 is applied to a mutable reference $\langle \ell_x, t_0 \rangle$ and the location ℓ_x currently has a stack $[\mathsf{Unique}(t_0)]$ and a value 42. We can give the following Hoare triple specification describing the behavior of f_1 in that situation:

$$
\begin{array}{l}
\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{p}} [\mathsf{Unique}(t_0)]\} \\
\quad f_1([\langle \ell_x, t_0 \rangle]) \\
\{\ell_x \mapsto 10 * \ell_x \mapsto_{\mathbf{p}} [\mathsf{Unique}(t_1), \mathsf{Unique}(t_0)]\}
\end{array}
$$

Here, the specification shows that the value stored at ℓ_x changes to 10 after the function f_1 has been applied to $\langle \ell_x, t_0 \rangle$, and the stack changes from $[\mathsf{Unique}(t_0)]$ to $[\mathsf{Unique}(t_1), \mathsf{Unique}(t_0)]$, since the function retags the mutable reference once, creating an additional reference x_1 with tag t_1. The

intermediate steps of the derivation for $f_1([\langle \ell_x, t_0 \rangle])$ are as follows:

$$
\begin{array}{l}
\texttt{rec}\, f_1([x_0]) := \\
\quad \{\ell_x \mapsto 42 * \ell \mapsto_{\mathbf{p}} [\mathsf{Unique}(t_0)]\} \\
\quad \texttt{let}\, x_1 = \texttt{retag}[\texttt{\&mut}](x_0)\, \texttt{in} \\
\quad \{\ell_x \mapsto 42 * \ell \mapsto_{\mathbf{p}} [\mathsf{Unique}(t_1), \mathsf{Unique}(t_0)]\} \\
\quad x_1 := 10 \\
\quad \{\ell_x \mapsto 10 * \ell \mapsto_{\mathbf{p}} [\mathsf{Unique}(t_1), \mathsf{Unique}(t_0)]\}
\end{array}
$$

Here, the application of H-App to reason about a function applied to an argument is left implicit.

We can obtain a similar Hoare triple for f_2, where the fact that it creates two mutable references is also reflected in the physical stack assertion:

$$
\begin{array}{l}
\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{p}} [\mathsf{Unique}(t_0)]\} \\
\quad f_2([\langle \ell_x, t_0 \rangle]) \\
\{\ell_x \mapsto 10 * \ell_x \mapsto_{\mathbf{p}} [\mathsf{Unique}(t_2), \mathsf{Unique}(t_1), \mathsf{Unique}(t_0)]\}
\end{array}
$$

Hence, the functions f_1 and f_2 end up with a specification that shows exactly how many reborrows they have performed internally, despite the fact that intuitively it does not matter for the externally visible behavior of the function. This is because the additional stack items $\mathsf{Unique}(t_2)$ and $\mathsf{Unique}(t_1)$ correspond to references $\langle \ell_x, t_2 \rangle$ and $\langle \ell_x, t_1 \rangle$ created inside the function that can *never be used again* after the function returns, because those references do not escape the function. Ideally, those references meant for "internal use" would not show up in the specification at all, allowing us to assign the same specification to f_1 and f_2.

Concurrency The lack of sufficient abstraction in Hoare triples based on the physical stack assertion also shows up when attempting to verify concurrent programs. For example, consider the following $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ program:

$$
\begin{array}{l}
\texttt{rec}\, f([]) := \\
\quad \texttt{let}\, x_0 = \texttt{alloc}()\, \texttt{in} \\
\quad x_0 := 42; \\
\quad \texttt{let}\, x_1 = \texttt{retag}[\texttt{\&mut}](x_0) \\
\quad \texttt{let}\, x_2 = \texttt{retag}[\texttt{\&mut}](x_1) \\
\quad x_2 := 10; \\
\quad ({}^*x_1 \parallel {}^*x_1); \\
\quad \texttt{free}(x_0)
\end{array}
$$

This program allocate a location ℓ_x and initializes that location with the value 42 before reborrowing twice to obtain mutable references x_1 and x_2 to the location, where x_2 is derived from x_1 and x_1 is derived from x_0. Then, it uses x_2 to write the value 10 to ℓ_x. Finally, it performs a *concurrent read* using x_1 by reading using that pointer in two threads concurrently.

This program never causes undefined behavior, although we cannot prove this based on the physical stack assertion, as we will demonstrate using the following incomplete (!) Hoare derivation:

$\{\mathsf{True}\}$
$\texttt{let}\ x_0 = \texttt{alloc}()\ \texttt{in}$
$\{\ell_x \mapsto \not{\mathfrak{B}} * \ell_x \mapsto_{\mathbf{p}} [t_0]\}$
$x_0 := 42;$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{p}} [t_0]\}$
$\texttt{let}\ x_1 = \texttt{retag}[\texttt{\&mut}](x_0)$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{p}} [t_1, t_0]\}$
$\texttt{let}\ x_2 = \texttt{retag}[\texttt{\&mut}](x_1)$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{p}} [t_2, t_1, t_0]\}$
$x_2 := 10;$
$\{\ell_x \mapsto 10 * \ell_x \mapsto_{\mathbf{p}} [t_2, t_1, t_0]\}$
$\{\ell_x \overset{1/2}{\mapsto} 10 * \ell_x \overset{1/2}{\mapsto} 10 * \ell_x \overset{1/2}{\mapsto}_{\mathbf{p}} [t_2, t_1, t_0] * \ell_x \overset{1/2}{\mapsto}_{\mathbf{p}} [t_2, t_1, t_0]\}$

$\{\ell_x \overset{1/2}{\mapsto} 10 * \ell_x \overset{1/2}{\mapsto}_{\mathbf{p}} [t_2, t_1, t_0]\}$	$\{\ell_x \overset{1/2}{\mapsto} 10 * \ell_x \overset{1/2}{\mapsto}_{\mathbf{p}} [t_2, t_1, t_0]\}$
$^*x_1$	$^*x_1$
$\{???\}$	$\{???\}$

$\{???\}$
$\texttt{free}(x_0)$
$\{???\}$

For brevity, we write $[t_1, t_0]$ instead of $[\mathsf{Unique}(t_1), \mathsf{Unique}(t_0)]$ in the above partial derivation. The pre- and postconditions show how the stack for location ℓ_x changes until we get to the concurrent read. At the point where the program reaches the concurrent read, the stack for ℓ_x is

$$[\mathsf{Unique}(t_2), \mathsf{Unique}(t_1), \mathsf{Unique}(t_0)]$$

as shown in the derivation.

Before the concurrent read, we split the points-to assertion and the physical stack assertion into fractions, and divide those fractions across the two threads using H-Par, because both threads access the location ℓ_x and therefore both threads need to have those assertions.

However, when attempting to prove a Hoare triple about $^*x_1$ in each thread, we run into a problem: we only have a fraction of the assertions in each thread, and therefore the only rule we could possibly use to reason about $^*x_1$ is H-SBP-Read-Fract. Unfortunately, this rule does not apply: it requires that $\mathsf{ReadSingle}(t, S) = S$, i.e. the stack must remain unchanged if we only have a fraction of the physical stack assertion. Hence, we cannot proceed with the proof, because reading using x_1 removes the item $\mathsf{Unique}(t_2)$ from the stack.

The mere fact that we cannot continue with a derivation is not a problem per se: it might be that we cannot derive a Hoare triple because the program

has undefined behavior. However, this program does not have undefined behavior, as we argue now. The concurrent reads using x_1 (holding the tagged pointer $\langle \ell_x, t_1 \rangle$) do not cause undefined behavior directly, because $\mathsf{Unique}(t_1)$ appears in the stack, granting read access to t_1, and neither of them removes that item. However, one of the two reads (depending on which is executed first) does end up removing $\mathsf{Unique}(t_2)$. Removing that item is not a problem, however, because the reference x_2 is *never used again* after the concurrent read.

This shows that the separation logic based on the physical stack assertion is not powerful enough to show the absence of undefined behavior in programs with concurrent reads that modify the stack. This is especially problematic, because the Rust equivalent of this program is accepted by the borrow checker, meaning that even the borrow checker with its relatively weak analysis is able to determine that this program does not have undefined behavior. This means that the separation logic based on the physical stack assertion is essentially weaker than the borrow checker for some programs.

How does the borrow checker determine that the program does not have undefined behavior? Intuitively, the borrow checker "knows" that the item $\mathsf{Unique}(t_2)$ (corresponding to x_2) can be safely removed during the concurrent read, because the lifetime of x_2 has ended by that point, meaning x_2 is never used again. Specifically, the lifetime of x_2 ends after $x_2 := 10$, since the borrow checker picks the shortest lifetime that contains all uses of x_2. Hence, the safety of the concurrent read is ensured by the constraint that a reference can only be used once the lifetimes of all derived references have ended. Since the lifetime of x_2 (derived from x_1) has ended by the point of the concurrent read, the concurrent read is accepted. This explanation is somewhat backwards, the borrow checker does not reason about stacks and instead Stacked Borrows was designed based on the borrow checker analysis, but it still serves as a useful intuition.

Again, the fact that the physical stack assertion is too concrete shows up here: the physical stack sometimes contains items which are no longer relevant and can be safely removed, because they will never be used again. Ideally, those irrelevant items would not appear in the specifications, just like specifications ideally would not describe references that are merely implementation details of functions.

4.4 Ghost stack assertion

In Section 4.3, we have argued that a separation logic based on the physical stack assertion is not sufficiently abstract, because Hoare triples reveal too many implementation details, which makes it unsuitable for reasoning about simple concurrent programs that can still be verified by the borrow checker. Specifically, the rule H-SBP-Read-Fract can only be used to reason about

concurrent reads that do not modify the stack for a location.

This section presents Stacked Borrows Separation Logic, a separation logic that allows for more abstract Hoare triples and is abstract enough to reason about some programs that perform concurrent reads that modify the stack. This logic is the main contribution of this thesis. We introduce SBSL in a step-by-step fashion: in this section, we introduce the rules for reasoning about programs that merely use mutable references (Unique items). In Section 4.5, we extend the logic with rules for reasoning about raw pointers (SharedRW items), which is a relatively simple extension. Finally, in Section 4.6, we add rules for reasoning about shared references (SharedRO items).

The main assertion of the logic is the *ghost stack assertion* $\ell \overset{q}{\mapsto}_{\mathbf{g}} G$. Whereas the physical stack assertion keeps track of the *exact* physical stack for each location, the ghost stack assertion keeps track of a *ghost stack* for each location. The main idea is that the ghost stack does not have to contain *all* of the items appearing in the physical stack: irrelevant items (corresponding to references that will never be used again) can be omitted in order to indicate that those items can be safely removed.

The name "ghost stack" derives from the *ghost state* mechanism of Iris (Jung et al. 2016, 2018b, 2015; Krebbers et al. 2017a), which allows one to define more abstract notions of resources on top of the physical state. This mechanism is used to implement ghost stacks.

The ghost stack assertion $\ell \overset{q}{\mapsto}_{\mathbf{g}} G$ should be read as "the ghost stack G is a *subsequence* of the physical stack for ℓ." The fraction q indicates whether the ghost stack can be modified, just as for the points-to assertion, and also allows splitting and combining into fractions:

$$\ell \overset{q_1+q_2}{\mapsto}_{\mathbf{g}} G \iff \ell \overset{q_1}{\mapsto}_{\mathbf{g}} G * \ell \overset{q_2}{\mapsto}_{\mathbf{g}} G$$

The syntax for ghost stacks G is as follows:

$$g \in \mathit{GItem} ::= \mathsf{GUnique}(t) \mid \ldots$$
$$G \in \mathit{GStack} ::= \mathit{List}(\mathit{GItem})$$

Note that the syntax of ghost stack is very similar to that of physical stacks. For example, the ghost item $\mathsf{GUnique}(t)$ corresponds to a physical item $\mathsf{Unique}(t)$. The reason for having a separate syntax is that SharedRO will have a different representation in ghost stacks compared to in physical stacks. The grammar will be extended in in Section 4.5 and Section 4.6, when we introduce the rules for raw pointers and shared references.

The idea is that the ghost stack merely contains those items from the physical stack which are still relevant, and should not be removed. For example,

$$\ell \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_2), \mathsf{GUnique}(t_0)]$$

means that the location ℓ has a physical stack that contains

$$[\mathsf{Unique}(t_2), \mathsf{Unique}(t_0)]$$

$$\textsc{SB-Forget}\ \frac{G' \text{ is a subsequence of } G}{\ell \mapsto_{\mathbf{g}} G \Rrightarrow \ell \mapsto_{\mathbf{g}} G'} \qquad \textsc{H-Vs}\ \frac{P \Rrightarrow P' \qquad \{P'\}\, e\, \{v.\, Q'\} \qquad \forall v.\, Q' \Rrightarrow Q}{\{P\}\, e\, \{v.\, Q\}}$$

$$\textsc{Vs-Frame}\ \frac{P \Rrightarrow Q}{P * R \Rrightarrow Q * R} \qquad \textsc{Vs-Seq}\ \frac{P \Rrightarrow Q \qquad Q \Rrightarrow R}{P \Rrightarrow R}$$

Figure 4.5: Stacked Borrows Separation Logic rules for ghost forgetting.

as a subsequence, but it might also include other items. For example, the physical stack for that location might be

$$[\mathsf{Unique}(t_3), \mathsf{Unique}(t_2), \mathsf{Unique}(t_1), \mathsf{Unique}(t_0)]$$

In this example, the items $\mathsf{Unique}(t_3)$ and $\mathsf{Unique}(t_1)$ do not appear in the ghost stack, despite the fact that they are still part of the physical stack for location ℓ. Those items have been "forgotten" in the program logic in order to indicate that they are no longer relevant and can be safely removed.

Ghost forgetting Stacked Borrows Separation logic includes a rule that allows one to forget about items in the physical stack by removing them from the ghost stack. This is called *ghost forgetting*, and the rules for it are shown in Fig. 4.5. The ghost forgetting rule SB-Forget allows removing an arbitrary subsequence of items from a ghost stack in the pre- or postcondition of Hoare triple. Ghost forgetting is essentially a logical operation: it changes a ghost stack that appears in a Hoare triple, while the corresponding physical stack does not change. This is unlike Hoare triples, which are used to reason about changes to the physical state caused by a program e. Hence, the ghost forgetting operation is not a Hoare triple, but a *view shift* $P \Rrightarrow Q$. A view shift $P \Rrightarrow Q$ should be read as: "the proposition P can be updated to become Q." View shifts are a general Iris mechanism used to describe changes to logical resources that are not necessarily accompanied by changes to physical resources. In some sense, a view shift can be seen as a Hoare triple without a program.

For example, consider the following instance of SB-Forget:

$$\ell \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_1), \mathsf{GUnique}(t_0)] \Rrightarrow \ell \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_0)]$$

This rule states that if we have exclusive ownership of the ghost stack assertion $\ell \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_1), \mathsf{GUnique}(t_0)]$, then we can update the ghost stack assertion to become $\ell \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_0)]$, effectively forgetting about the $\mathsf{GUnique}(t_1)$ item. We can apply this view shift while deriving a Hoare triple using H-Vs,

Vs-Seq, and Vs-Frame. For example, this allows the following reasoning step:

$$\frac{\{\ell \mapsto 0 * \ell \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_0)]\}\, e\, \{v.\, Q\}}{\{\ell \mapsto 0 * \ell \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_1), \mathsf{GUnique}(t_0)]\}\, e\, \{v.\, Q\}}$$

Read from bottom to top, here we are updating the ghost stack for ℓ in the precondition by forgetting an item before proving the Hoare triple for the program e. Similarly, we can also forget items from ghost stacks in the postconditions of Hoare triples.

Why is it useful to forget items from ghost stacks appearing in Hoare triples? This is because it allows hiding the fact that new references have been created from Hoare triples, leading to more abstract specifications, and enabling better reasoning about concurrency, as we shall see.

Rules for reasoning about $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ programs The SBSL rules for mutable references are shown in Fig. 4.6.

The rule for allocation, H-SB-Alloc, is essentially exactly the same as for the physical stack assertion. After allocation, we obtain the ghost stack assertion $\ell \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t)]$ and a points-to assertion for the tagged pointer $\langle \ell, t \rangle$ produced by the allocation. Hence, in this case the ghost stack is exactly the same as the physical stack.

The rule for writing, H-SB-Write, is more interesting. Recall that the intuition behind the ghost stack $\ell \overset{q}{\mapsto}_{\mathbf{g}} G$ is that G contains the items from the physical stack that are still relevant and should not be removed, where irrelevant items can be forgotten using SB-Forget. Hence, a write should not be allowed to remove any items from the physical stack that still appears in the ghost stack. Moreover, in order to avoid undefined behavior, writing should only be done using tags t that have write permissions according to the physical stack. These two constraints are captured by the $\mathsf{SBGrantsWrite}$ requirement in the precondition: $\mathsf{SBGrantsWrite}\, G\, t$ should be read as: "for a location with ghost stack G, the tag t can used for writing without causing undefined behavior, and doing so preserves G as a subsequence of the physical stack." Because a write is required to preserve G as a subsequence of the physical stack, the same ghost stack appears in the precondition and the postcondition.

So far, there is only a single rule SB-Grants-Write-Unique for deriving $\mathsf{SBGrantsWrite}\, G\, t$: it allows us to write using a tag t if the ghost stack has $\mathsf{GUnique}(t)$ *on top*. Why is this allowed? First of all, if $\mathsf{GUnique}(t)$ appears in the ghost stack, then $\mathsf{Unique}(t)$ appears in the physical stack (because the ghost stack is a subsequence of the physical stack), meaning the access does not cause undefined behavior. Moreover, according to the operational semantics, writing removes all items *above* the topmost granting item. Hence, none of the items in the ghost stack are removed since those are all below a granting item $\mathsf{GUnique}(t)$.

H-SB-Alloc
$\{\mathsf{True}\}$

$\texttt{alloc}()$

$\{w.\, \exists \ell, t.\, (w = \langle \ell, t \rangle) * \ell \mapsto \skull * \ell \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t)]\}$

H-SB-Write
$\{\ell \mapsto v * \ell \mapsto_{\mathbf{g}} G * \mathsf{SBGrantsWrite}\, G\, t\}$

$\langle \ell, t \rangle := v'$

$\{w.\, (w = \skull) * \ell \mapsto v' * \ell \mapsto_{\mathbf{g}} G\}$

H-SB-Read
$\{\ell \overset{q}{\mapsto} v * \ell \overset{q}{\mapsto}_{\mathbf{g}} G * \mathsf{SBGrantsRead}\, G\, t\}$

$^*\langle \ell, t \rangle$

$\{w.\, (w = v) * \ell \overset{q}{\mapsto} v * \ell \overset{q}{\mapsto}_{\mathbf{g}} G\}$

H-SB-Free
$\{\ell \mapsto v * \ell \mapsto_{\mathbf{g}} G * \mathsf{SBGrantsWrite}\, G\, t\}$

$\texttt{free}(\langle \ell, t \rangle)$

$\{w.\, w = \skull\}$

H-SB-Retag-Mutable
$\{\ell \mapsto_{\mathbf{g}} G * \mathsf{SBGrantsWrite}\, G\, t_{\text{old}}\}$

$\texttt{retag}[\texttt{\&mut}](\langle \ell, t_{\text{old}} \rangle)$

$\{w.\, \exists t_{\text{new}}.\, (w = \langle \ell, t_{\text{new}} \rangle) * \ell \mapsto_{\mathbf{g}} (\mathsf{GUnique}(t_{\text{new}}) :: G)\}$

SB-Grants-Write-Unique
$\mathsf{SBGrantsWrite}\, (\mathsf{GUnique}(t) :: G)\, t$

SB-Grants-Read-Unique
$\mathsf{SBGrantsRead}\, (\mathsf{GUnique}(t) :: G)\, t$

Figure 4.6: Stacked Borrows Separation Logic rules for reasoning about pointer-related operations.

For example, the following is an instance of H-SB-Write:

$$\begin{aligned}&\{\ell \mapsto 0 * \ell \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_1), \mathsf{GUnique}(t_0)]\}\\&\quad \langle \ell, t_1 \rangle := 1\\&\{\ell \mapsto 1 * \ell \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_1), \mathsf{GUnique}(t_0)]\}\end{aligned}$$

This instance can be derived using H-SB-Write, SB-Grants-Write-Unique, and the rule of consequence.

Similarly, the rule for reading, H-SB-Read, allows reading with a tag t if $\mathsf{GUnique}(t)$ appears on top of the ghost stack (SB-Grants-Read-Unique), and again the same ghost stack appears in the precondition and the postcondition. Contrary to H-SB-Write, only fractional ownership of the points-to assertion and the ghost stack assertion is required, since neither are modified. The intuition behind $\mathsf{SBGrantsRead}$ is similar to that behind $\mathsf{SBGrantsWrite}$: it means that t can be used for reading without causing undefined behavior while preserving G as a subsequence.

The rule for deallocation, H-SB-Free, is almost the same as H-SB-Write, since deallocation counts as a write to Stacked Borrows. The only difference is that the points-to assertion and ghost stack assertion no longer appear in the postcondition, disallowing the location from being accessed after deallocation.

Finally, the rule H-SB-Retag-Mutable allows reasoning about mutable references derived using retagging. It requires that t_{old} be allowed to write, and extends the ghost stack with an additional $\mathsf{GUnique}(t_{\text{new}})$ item, reflecting the new item that gets added to the physical stack when retagging.

In order to demonstrate the usefulness of the logic, we now apply it to the two examples from Section 4.3, for which we could not obtain satisfactory specifications using the physical stack assertion.

Example (abstraction) We again consider the following two functions:

$$\begin{aligned}&\texttt{rec}\ f_1([x_0]) :=\\&\quad \texttt{let}\ x_1 = \texttt{retag}[\texttt{\&mut}](x_0)\ \texttt{in}\\&\quad x_1 := 10\end{aligned}$$

$$\begin{aligned}&\texttt{rec}\ f_2([x_0]) :=\\&\quad \texttt{let}\ x_1 = \texttt{retag}[\texttt{\&mut}](x_0)\ \texttt{in}\\&\quad \texttt{let}\ x_2 = \texttt{retag}[\texttt{\&mut}](x_1)\ \texttt{in}\\&\quad x_2 := 10\end{aligned}$$

When naively applying the rules based on the physical stack assertion, we ended up with two different specifications, and the specification reflected the number of references created inside the function, even though those references did not escape the function.

Now, based on the ghost forgetting rule SB-Forget, we can assign the same specification to both of these functions by forgetting about the references

created inside the function. Specifically, we show the following Hoare triple for both functions $f \in \{f_1, f_2\}$:

$$\begin{array}{l}\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_0)]\} \\ \quad f([\langle \ell, t_0 \rangle]) \\ \{\ell_x \mapsto 10 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_0)]\}\end{array}$$

The outlines of the derivations are as follows:

$\texttt{rec}\ f_1([x_0]) :=$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_0)]\}$
$\texttt{let}\ x_1 = \texttt{retag}[\texttt{\&mut}](x_0)\ \texttt{in}$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_1), \mathsf{GUnique}(t_0)]\}$
$x_1 := 10$
$\{\ell_x \mapsto 10 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_1), \mathsf{GUnique}(t_0)]\}$
(ghost forgetting)
$\{\ell_x \mapsto 10 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_0)]\}$

$\texttt{rec}\ f_2([x_0]) :=$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_0)]\}$
$\texttt{let}\ x_1 = \texttt{retag}[\texttt{\&mut}](x_0)\ \texttt{in}$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_1), \mathsf{GUnique}(t_0)]\}$
$\texttt{let}\ x_2 = \texttt{retag}[\texttt{\&mut}](x_1)\ \texttt{in}$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_2), \mathsf{GUnique}(t_1), \mathsf{GUnique}(t_0)]\}$
$x_2 := 10$
$\{\ell_x \mapsto 10 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_2), \mathsf{GUnique}(t_1), \mathsf{GUnique}(t_0)]\}$
(ghost forgetting)
$\{\ell_x \mapsto 10 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_0)]\}$

Note that at every step, we only write or retag using the tag t that appears in a $\mathsf{GUnique}(t)$ item on top of the ghost stack, in accordance with the rules H-SB-Write and H-SB-Retag-Mutable. The derivations are quite similar to those based on the physical stack assertion, except we apply SB-Forget to forget about the items with tags t_1 and t_2, because those correspond to references created inside the function.

This shows how the notion of ghost forgetting can be used to obtain more abstract specifications for functions, in which implementation details are hidden. This makes it easier to maintain a proof of program correctness, because it means that changing minor implementation details of a function appearing in a larger program does not require reproving the correctness of the whole program.

Example (concurrency) Now we show that SBSL is also able to verify the absence of undefined behavior in a program that performs a concurrent read that modifies the physical stack, which was not possible using the physical stack assertion.

We again consider the concurrency example from Section 4.3. The following shows the concurrent read example annotated with preconditions and postconditions according to Stacked Borrows Separation Logic:

$\{\mathsf{True}\}$
$\texttt{let}\ x_0 = \texttt{alloc}()\ \texttt{in}$
$\{\ell_x \mapsto \not{\Omega} * \ell_x \mapsto_{\mathbf{g}} [t_0]\}$
$x_0 := 42;$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [t_0]\}$
$\texttt{let}\ x_1 = \texttt{retag}[\texttt{\&mut}](x_0)$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [t_1, t_0]\}$
$\texttt{let}\ x_2 = \texttt{retag}[\texttt{\&mut}](x_1)$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [t_2, t_1, t_0]\}$
$x_2 := 10;$
$\{\ell_x \mapsto 10 * \ell_x \mapsto_{\mathbf{g}} [t_2, t_1, t_0]\}$
(ghost forgetting)
$\{\ell_x \mapsto 10 * \ell_x \mapsto_{\mathbf{g}} [t_1, t_0]\}$
$\{\ell_x \overset{1/2}{\longmapsto} 10 * \ell_x \overset{1/2}{\longmapsto} 10 * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [t_1, t_0] * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [t_1, t_0]\}$

$\{\ell_x \overset{1/2}{\longmapsto} 10 * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [t_1, t_0]\}$	$\{\ell_x \overset{1/2}{\longmapsto} 10 * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [t_1, t_0]\}$
$^*x_1$	$^*x_1$
$\{\ell_x \overset{1/2}{\longmapsto} 10 * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [t_1, t_0]\}$	$\{\ell_x \overset{1/2}{\longmapsto} 10 * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [t_1, t_0]\}$

$\{\ell_x \overset{1/2}{\longmapsto} 10 * \ell_x \overset{1/2}{\longmapsto} 10 * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [t_1, t_0] * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [t_1, t_0]\}$
$\{\ell_x \mapsto 10 * \ell_x \mapsto_{\mathbf{g}} [t_1, t_0]\}$
(ghost forgetting)
$\{\ell_x \mapsto 10 * \ell_x \mapsto_{\mathbf{g}} [t_0]\}$
$\texttt{free}(x_0)$
$\{\mathsf{True}\}$

Here, we have again shortened the ghost stacks by writing $[t_1, t_0]$ instead of $[\mathsf{GUnique}(t_1), \mathsf{GUnique}(t_0)]$. Again, it is easy to verify that all writes, reads, and retags at each point in the program are performed using the tag t appearing in the $\mathsf{GUnique}(t)$ item on top of the ghost stack, as required by rules such as H-SB-Write, H-SB-Read, and H-SB-Retag-Mutable. Moreover, we can see that retagging adds additional tags to the stack.

This time, we are able to show the absence of undefined behavior in the program, despite the fact that it performs a concurrent read where one of the two reads ends up removing an item from the physical stack. The crucial point that makes the proof work is that we are able to apply the ghost forgetting rule (SB-Forget) *before* splitting the assertions into fractions, allowing the $\mathsf{GUnique}(t_2)$ item to be forgotten before it is actually removed from the physical stack by one of the two threads. Hence, in the proof each thread ends up with the ghost stack $[\mathsf{GUnique}(t_1), \mathsf{GUnique}(t_0)]$, which has t_1 on top and therefore we are able to apply H-SB-Read to show that the concurrent reads do not cause undefined behavior.

H-SB-Retag-Raw
$\{\ell \mapsto_{\mathbf{g}} G * \mathsf{SBGrantsWrite}\, S\, t_{\text{old}}\}$

$\texttt{retag}[\texttt{*mut}](\langle \ell, t_{\text{old}} \rangle)$

$\{w.\, w = \langle \ell, \bot \rangle * \ell \mapsto_{\mathbf{g}} (\mathsf{GSharedRW}(\bot) :: G)\}$

SB-Grants-Write-SharedRW
$\mathsf{SBGrantsWrite}\, (\mathsf{GSharedRW}(\bot) :: G)\, \bot$

SB-Grants-Read-SharedRW
$\mathsf{SBGrantsRead}\, (\mathsf{GSharedRW}(\bot) :: G)\, \bot$

SB-Grants-Read-Skip-SharedRW
$\mathsf{SBGrantsRead}\, G\, t \Rightarrow \mathsf{SBGrantsRead}\, (\mathsf{GSharedRW}(\bot) :: G)\, t$

Figure 4.7: Stacked Borrows Separation Logic rules for raw pointers.

After the concurrent part, the postconditions of both threads are combined according to H-Par, and subsequently we apply SB-Forget once more to bring t_0 to the top of the ghost stack, which allows applying the H-SB-Free rule.

This shows that the logic is more suited for reasoning about concurrent programs due to the fact that it decouples the point in the program where a stack item becomes irrelevant (because it will never be used again) from the point where it might actually be removed.

4.5 Raw pointers

In this section, we extend the logic with rules for reasoning about raw pointers. Having program logic rules that account for raw pointers is essential, because raw pointers are typically used for situations where the borrow checker cannot guarantee the absence of undefined behavior. Hence, by having reasoning rules for raw pointers we can allow such code to be manually shown to be free from undefined behavior using the program logic.

Fortunately, extending the logic to support raw pointers is a relatively straightforward extension of the ideas in Section 4.4. First, we extend the syntax of ghost stacks to account for $\mathsf{SharedRW}(\bot)$ items:

$$g \in \mathit{GItem} ::= \mathsf{GUnique}(t) \mid \mathsf{GSharedRW}(\bot) \mid \ldots$$

The additional rules for raw pointers are shown in Fig. 4.7. We have additional rules for the $\mathsf{SBGrantsRead}$ and $\mathsf{SBGrantsWrite}$ predicates, indicating that when $\mathsf{GSharedRW}(\bot)$ appears on top of the ghost stack, we are allowed to read and write using the tag $\bot$ (SB-Grants-Write-SharedRW

and SB-Grants-Read-SharedRW). The idea here is much the same as for $\mathsf{GUnique}(t)$ items: if $\mathsf{GSharedRW}(\bot)$ appears on top of the ghost stack, then reading and writing using $\bot$ does not cause undefined behavior (since it also appears in the physical stack) and also preserves the ghost stack as a subsequence of the physical stack (because none of the items occurring below $\mathsf{SharedRW}(\bot)$ are removed by reading and writing using $\bot$).

The rule SB-Grants-Read-Skip-SharedRW is more interesting: it says that if we are allowed to read with the tag t according to the ghost stack G, then we are also allowed to read with that tag according to $\mathsf{GSharedRW}(\bot) :: G$. Effectively, we are allowed to "skip" over $\mathsf{GSharedRW}(\bot)$ items on top of the ghost stack when considering whether reading using t is allowed. For example, according to the ghost stack $[\mathsf{GSharedRW}(\bot), \mathsf{GUnique}(t_0)]$, it is allowed to read using t_0. It is no longer necessarily the case that the tag t we are using has to appear in the topmost item of the ghost stack.

What is the justification for being allowed to skip over $\mathsf{GSharedRW}(\bot)$ items? The reason is that in the operational semantics, reading *never* removes any $\mathsf{SharedRW}(\bot)$ items, since it only removes Unique items above the granting item. Hence, even if we read using an item that appears below several $\mathsf{SharedRW}(\bot)$ items, we can still be sure that none of the items appearing in the ghost stack will be removed (i.e., none of the still-relevant items are removed).

Finally, we also have a rule for casting references to raw pointers: H-SB-Retag-Raw. This rule is very similar to H-SB-Retag-Mutable, which is used to reason about deriving new mutable references by retagging. The H-SB-Retag-Raw requires that t_{old} be allowed write access according to the current ghost stack for ℓ, since casting to a raw pointer counts as write. The ghost stack in the postcondition is extended with an additional $\mathsf{GSharedRW}(\bot)$ item, reflecting the new $\mathsf{SharedRW}(\bot)$ item added to the physical stack.

Example using raw pointers We now apply the new rules to a $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ program that uses raw pointers. Consider the following program:

$$
\begin{array}{l}
\texttt{let}\ x_0 = \texttt{alloc}()\ \texttt{in} \\
x_0 := 42; \\
\texttt{let}\ x_1 = \texttt{retag}[\texttt{\&mut}](x_0)\ \texttt{in} \\
\texttt{let}\ x_\bot = \texttt{retag}[\texttt{*mut}](x_1)\ \texttt{in} \\
{}^*x_\bot; \\
{}^*x_1; \\
{}^*x_\bot; \\
{}^*x_0; \\
{}^*x_\bot; \\
\texttt{free}(x_0)
\end{array}
$$

This roughly corresponds to the following Rust program:

```
1  let mut x0 = 42;
2  let x1 = &mut x0;
3  let xraw = x1 as *mut i32;
4  println!("{}", unsafe { *xraw });
5  println!("{}", *x1);
6  println!("{}", unsafe { *xraw });
7  println!("{}", x0);
8  println!("{}", unsafe { *xraw });
```

The prints are added to the Rust program such that the program does not contain a sequence of reads whose result is unused. Interestingly, note that the raw pointers remains usable (i.e., can be accessed without causing undefined behavior) even after reading from `x0` and `x1`. In contrast, for mutable references, Stacked Borrows only allows `x1` to be used from the point where it is created to the point where `x0` is read. Similarly, the borrow checker enforces that `x0` can only be used once the lifetime of `x1` has ended, leading to a similar constraint.

This program does not have undefined behavior, which we show using the following outline of a Hoare triple derivation:

$\{\mathsf{True}\}$
$\texttt{let}\ x_0 = \texttt{alloc}()\ \texttt{in}$
$\{\ell_x \mapsto \not{\mathfrak{B}} * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_0)]\}$
$x_0 := 42;$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_0)]\}$
$\texttt{let}\ x_1 = \texttt{retag}[\texttt{\&mut}](x_0)\ \texttt{in}$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_1), \mathsf{GUnique}(t_0)]\}$
$\texttt{let}\ x_\perp = \texttt{retag}[\texttt{*mut}](x_1)\ \texttt{in}$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GSharedRW}(\perp), \mathsf{GUnique}(t_1), \mathsf{GUnique}(t_0)]\}$
$^*x_\perp;$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GSharedRW}(\perp), \mathsf{GUnique}(t_1), \mathsf{GUnique}(t_0)]\}$
$^*x_1;$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GSharedRW}(\perp), \mathsf{GUnique}(t_1), \mathsf{GUnique}(t_0)]\}$
$^*x_\perp;$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GSharedRW}(\perp), \mathsf{GUnique}(t_1), \mathsf{GUnique}(t_0)]\}$
(ghost forgetting)
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GSharedRW}(\perp), \mathsf{GUnique}(t_0)]\}$
$^*x_0;$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GSharedRW}(\perp), \mathsf{GUnique}(t_0)]\}$
$^*x_\perp;$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GSharedRW}(\perp), \mathsf{GUnique}(t_0)]\}$
(ghost forgetting)
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\mathsf{GUnique}(t_0)]\}$
$\texttt{free}(x_0)$

It is easy to verify that at each point the preconditions SBGrantsWrite and SBGrantsRead for the applicable rule are satisfied, which in practice means

that we only perform writing and retagging (deriving mutable references or raw pointers) using the tag appearing in the topmost item of the ghost stack. In the case of reading, we are also able to read using t_1 while it appears below $\mathsf{GSharedRW}(\bot)$ in the ghost stack: this is allowed by the SB-Grants-Read-Skip-SharedRW rule.

Moreover, using ghost forgetting, we can take arbitrary subsequences of the ghost stack: this allows us to forget about the $\mathsf{GUnique}(t_1)$ item while leaving the $\mathsf{GSharedRW}(\bot)$ item on top of it. After that point, we are allowed to read using t_0, since it only appears below $\mathsf{GSharedRW}(\bot)$ in the ghost stack, and afterwards we are still able to read using $\bot$ because we did not forget about that item.

4.6 Shared references

In this section, we make the final extension to Stacked Borrows Separation Logic, adding rules for reasoning about shared references. This extension is not entirely trivial, because it is possible to derive additional shared references concurrently from multiple threads in safe Rust (i.e., the borrow checker allows this without requiring unsafe operations). The ghost forgetting rule enables reasoning about programs that remove items from the stack in a concurrent setting. Now, we need rules that allow items (specifically, $\mathsf{SharedRO}$ items corresponding to shared references) to be added to the stack concurrently.

Consider the following $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ program, which demonstrates concurrent retagging:

$$
\begin{array}{l}
\texttt{let}\, x_0 = \texttt{alloc}()\, \texttt{in} \\
x_0 := 42 \\
\texttt{let}\, x_1 = \texttt{retag}[\&](x_0)\, \texttt{in} \\
{}^*x_1 \\
\texttt{free}(x_0)
\end{array}
\;\Bigg\|\;
\begin{array}{l}
\texttt{let}\, x_2 = \texttt{retag}[\&](x_0)\, \texttt{in} \\
\texttt{let}\, x_3 = \texttt{retag}[\&](x_2)\, \texttt{in} \\
{}^*x_3
\end{array}
$$

Here, the physical stack for the allocated location starts out as $[\mathsf{Unique}(t_0)]$. Then, we have two threads that concurrently derive additional shared references and use the shared references to read the value stored in the location. This means that both threads concurrently add $\mathsf{SharedRO}(t)$ items to the stack, and then use the tags t of their respective items to read from the location. One possible ordering in which this might happen is that the left-most thread executes to completion, followed by the rightmost thread executing to completion. In that case, the physical stack would change in the following

way:

$[\mathsf{Unique}(t_0)]$ (initial)
$[\mathsf{SharedRO}(\{t_1\}), \mathsf{Unique}(t_0)]$ (after leftmost thread)
$[\mathsf{SharedRO}(\{t_3, t_2, t_1\}), \mathsf{Unique}(t_0)]$ (after rightmost thread)

There are other possible orderings as well: the rightmost thread might execute in its entirety first, or the threads might execute in an interleaved fashion.

The above program is in fact free from undefined behavior: intuitively, this is because all of the threads only create shared references from $\mathsf{Unique}(t_0)$ and from other shared references and perform reads. None of these operations removes the $\mathsf{Unique}(t_0)$ item or any SharedRO items from the stack (reading or deriving shared references only ever removes Unique items), and therefore we can be sure that regardless of the order in which instructions from threads execute, the required granting items will still be present in the stack.

The borrow checker is also able to determine that this program does not have undefined behavior: in short, this is because the lifetimes of shared references are only required to end once a write access to the original reference occurs. Since the program does not perform any writes, the shared references created in multiple threads can peacefully coexist.

Note in particular that the precise ordering in which threads are executed does not really matter: the leftmost thread adds a $\mathsf{SharedRO}(t_1)$ item to the stack and reads using t_1, but in reasoning about that thread it does not matter whether the rightmost thread has already added its $\mathsf{SharedRO}(t_2)$ item or not. Hence, in the separation logic spirit, we would like to be able to reason about the threads independently: the reasoning in each thread should be independent of $\mathsf{SharedRO}(t)$ items added by other threads concurrently.

We now introduce the rules of Stacked Borrows Separation Logic for reasoning about shared references, which allow independent reasoning about the SharedRO items added by each thread. Whereas Unique items and SharedRW items are represented in essentially the same way in the physical stacks and the ghost stacks (i.e., $\mathsf{GUnique}(t)$ corresponds to $\mathsf{Unique}(t)$ and $\mathsf{GSharedRW}(\bot)$ corresponds to $\mathsf{SharedRW}(\bot)$), we do not use the same representation for SharedRO items. Hence, whereas the physical stack might contain items like $\mathsf{SharedRO}(\{t_1, t_2, t_3\})$ (a group of three consecutive SharedRO items grouped into a set), there are no corresponding $\mathsf{GSharedRO}(\{t_1, t_2, t_3\})$ items. The reason for this is that we are generally not interested in knowing the *whole* set of tags in a SharedRO item (some of which might have been added by different threads concurrently) in the program logic: we are mainly interested in membership of individual SharedRO items, and particularly those SharedRO items that are known to exist in the current thread (e.g., because the current thread has added them).

Set membership assertion To enable local reasoning about the individual tags in a group of SharedRO items, we use the following representation for GSharedRO items:

$$\gamma \in \mathit{SetName}$$
$$g \in \mathit{GItem} ::= \mathsf{GUnique}(t) \mid \mathsf{GSharedRW}(\bot) \mid \mathsf{GSharedRO}(\gamma)$$

This represents the final addition to the syntax of ghost stacks. Here, the GSharedRO item does not contain a set of tags directly. Instead, it contains a *set name* γ. The idea is that this set name refers to a (potentially empty) set of SharedRO items (or rather, the *tags* of those SharedRO items). Then, we add a new assertion $\mathsf{SBSetContains}\,\gamma\,t$, the *set membership assertion*, that can be read as "the set named γ contains the tag t." This assertion is used to keep track of membership of individual $\mathsf{SharedRO}(t)$ items in a group of SharedRO items.

The set membership assertion is defined in terms of standard built-in Iris constructions. Hence, the general idea of representing sets indirectly in this way is not a novel contribution. However, what is interesting is how the set membership assertion and the ghost stack assertion interact in the proof rules for shared references.

Based on the set membership assertion, we can describe the situation where the physical stack has a group of SharedRO items on top that contains the tag t_1 as follows:

$$\ell \mapsto_{\mathbf{g}} [\mathsf{GSharedRO}(\gamma), \mathsf{GUnique}(t_0)] * \mathsf{SBSetContains}\,\gamma\,t_1$$

Here, the ghost stack indicates that there is a group of SharedRO items named γ in the physical stack, and the set membership assertion indicates that the tag t_1 is part of that group. For example, the corresponding physical stack might look like this:

$$[\mathsf{SharedRO}(\{t_1, t_2, t_3\}), \mathsf{Unique}(t_0)]$$

We are only able to conclude that a certain tag appears in the physical stack based on the set membership assertion if the set names in the ghost stack and the set membership assertion match. Hence,

$$\ell \mapsto_{\mathbf{g}} [\mathsf{GSharedRO}(\gamma'), \mathsf{GUnique}(t_0)] * \mathsf{SBSetContains}\,\gamma\,t_1$$

does not tell us anything about whether there is a $\mathsf{SharedRO}(t_1)$ item in the stack, unless $\gamma = \gamma'$.

The set membership assertion $\mathsf{SBSetContains}\,\gamma\,t$ represents the *knowledge* that the set named γ contains the tag t. For this reason, the $\mathsf{SBSetContains}\,\gamma\,t$ proposition is duplicable [2]:

$$\mathsf{SBSetContains}\,\gamma\,t \iff \mathsf{SBSetContains}\,\gamma\,t * \mathsf{SBSetContains}\,\gamma\,t$$

[2] In fact, the set membership assertion is *persistent*, an even stronger notion than duplicability in Iris.

H-SB-Retag-Shared
$\{\ell \overset{q}{\mapsto}_{\mathbf{g}} G * \mathsf{SBGrantsRead}\, G\, t_{\text{old}} * (G = \mathsf{GSharedRO}(\gamma) :: G')\}$

$\texttt{retag}[\&](\langle \ell, t_{\text{old}} \rangle)$

$\{w.\, \exists t_{\text{new}}.\, (w = \langle \ell, t_{\text{new}} \rangle) * \ell \overset{q}{\mapsto}_{\mathbf{g}} G * \mathsf{SBSetContains}\, \gamma\, t_{\text{new}}\}$

SB-Share
$$\frac{G \text{ is not of the form } \mathsf{GSharedRO}(\gamma) :: G'}{\ell \mapsto_{\mathbf{g}} G \Rrightarrow \exists \gamma.\, \ell \mapsto_{\mathbf{g}} (\mathsf{GSharedRO}(\gamma) :: G)}$$

SB-Grants-Read-SharedRO
$\mathsf{SBSetContains}\, \gamma\, t \Rightarrow \mathsf{SBGrantsRead}\, (\mathsf{GSharedRO}(\gamma) :: G)\, t$

SB-Grants-Read-Skip-SharedRO
$\mathsf{SBGrantsRead}\, G\, t \Rightarrow \mathsf{SBGrantsRead}\, (\mathsf{GSharedRO}(\gamma) :: G)\, t$

SBSetContains-Dup
$\mathsf{SBSetContains}\, \gamma\, t \iff \mathsf{SBSetContains}\, \gamma\, t * \mathsf{SBSetContains}\, \gamma\, t$

Figure 4.8: Stacked Borrows Separation Logic rules for shared references.

This means that the knowledge that t belongs to the set named γ can be freely distributed to multiple threads using the H-Par rule. This is unlike the points-to assertion or the ghost stack assertion, which cannot be duplicated freely (although they can be split into fractions for read-only access).

The proposition $\mathsf{SBSetContains}\, \gamma\, t$ should not be confused with a proposition like $t \in \gamma$, because γ is not a set: it is just a *name* that refers to a set, and as we will see, the set that this name refers to can grow to contain more elements during a Hoare triple derivation.

Rules for shared references We can use the set membership assertion to give rules for shared references. The SBSL rules for shared references are shown in Fig. 4.8.

First of all, we have additional rules for $\mathsf{SBGrantsRead}$. The rule SB-Grants-Read-SharedRO states that if we have a $\mathsf{SBSetContains}\, \gamma\, t$ token (indicating that t belongs to the set named γ), and $\mathsf{GSharedRO}(\gamma)$ appears on top of the ghost stack, then we are allowed to read using the tag t. This is because this combination of resources means that a $\mathsf{SharedRO}(t)$ item appears in the physical stack, meaning reading does not cause undefined behavior, and moreover reading using a $\mathsf{SharedRO}(t)$ item does not remove *any* items from the physical stack at all (because $\mathsf{SharedRO}$ items always sit on top of the stack), meaning the ghost stack is preserved as a subsequence.

For example, the assertions

$$\ell \mapsto_{\mathbf{g}} [\mathsf{GSharedRO}(\gamma), \mathsf{GUnique}(t_0)] * \mathsf{SBSetContains}\,\gamma\,t_1$$

imply that we are allowed to read using $\langle \ell, t_1 \rangle$ according to the logic. We are able to derive the following Hoare triple, based on H-Conseq, H-SB-Read, and SB-Grants-Read-SharedRO:

$$\{\ell \mapsto 0 * \ell \mapsto_{\mathbf{g}} [\mathsf{GSharedRO}(\gamma), \mathsf{GUnique}(t_0)] * \mathsf{SBSetContains}\,\gamma\,t_1\}$$
$$\quad {}^*\langle \ell, t_1 \rangle$$
$$\{w.\ (w = 0) * \ell \mapsto 0 * \ell \mapsto_{\mathbf{g}} [\mathsf{GSharedRO}(\gamma), \mathsf{GUnique}(t_0)] * \mathsf{SBSetContains}\,\gamma\,t_1\}$$

Moreover, if we are allowed to read using the tag t according to the ghost stack G, then we are also allowed to do so using $\mathsf{GSharedRO}(\gamma) :: G$, as indicated by SB-Grants-Read-Skip-SharedRO. The idea behind this is the same as for SB-Grants-Read-Skip-SharedRW: reading never removes any SharedRO items from the physical stack, and hence reading using a tag that appears below several $\mathsf{GSharedRO}(\gamma)$ and $\mathsf{GSharedRW}(\bot)$ still preserves the ghost stack as a subsequence of the physical stack.

For example, this means that according to the ghost stack

$$[\mathsf{GSharedRO}(\gamma), \mathsf{GUnique}(t_0)]$$

it is allowed to read using t_0, because doing so does not remove the SharedRO items represented by $\mathsf{GSharedRO}(\gamma)$.

We also have the H-SB-Retag-Shared rule, which is used for reasoning about new shared references derived using retagging, potentially concurrently. Note that this rule only requires a *fraction* of the ghost stack assertion, which is what allows it to be applied in a proof about a concurrent program. This rule requires that t_{old} has read access according to the ghost stack for location ℓ (since retagging for shared references counts as a read access), and it requires the ghost stack to be of the form $\mathsf{GSharedRO}(\gamma) :: G'$. Retagging for shared references produces a new tagged pointer $\langle \ell, t_{\text{new}} \rangle$, and we obtain a *new* set membership assertion $\mathsf{SBSetContains}\,\gamma\,t_{\text{new}}$, which tells us that t_{new} is now in the set called γ. Hence, this rule is where set membership assertions are "created."

The rule SB-Share, called the *ghost sharing rule*, allows us to add a $\mathsf{GSharedRO}(\gamma)$ item on top of a ghost stack that does not already have one. This is a view shift, just like the SB-Forget rule, which means it is not accompanied by a change in the physical state. Hence, there are not necessarily any SharedRO items in the physical stack yet when there is a $\mathsf{GSharedRO}(\gamma)$ in the ghost stack. It is only when we apply the H-SB-Retag-Shared rule that we learn that there are actually some tags in the set referred to by γ.

The SB-Share rule can be applied in a proof to indicate the intent to add SharedRO items to the stack, potentially in a concurrent fashion. After

applying this rule, it is no longer possible to write to the location according to the proof rules (because SBGrantsWrite cannot be shown for a ghost stack with a GSharedRO on top), until the SB-Forget rule is applied to remove the GSharedRO item.

After applying SB-Forget to remove a $\mathsf{GSharedRO}(\gamma)$ item from the ghost stack, all of the $\mathsf{SBSetContains}\,\gamma\,t$ set membership assertions no longer tell us anything about the physical stack: they essentially become worthless pieces of information. The intuitive reason for this is that applying SB-Forget means that the group of SharedRO items currently on top of the physical stack is no longer relevant, and can be safely removed (say, by writing to the location). Hence, after applying SB-Forget, the items might not even be in the physical stack anymore. The next time SB-Share is applied, a new $\mathsf{GSharedRO}(\gamma')$ item is added to the ghost stack. However, the $\mathsf{SBSetContains}\,\gamma\,t$ tokens remain useless, because we cannot conclude that $\gamma = \gamma'$ since γ' is existentially quantified in SB-Share.

Example We now apply the rules for shared references to the $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ program that creates shared references concurrently, which was shown in the beginning of this section.

The following is an outline of the Hoare triple derivation:

$\{\mathsf{True}\}$
$\texttt{let}\ x_0 = \texttt{alloc}()\ \texttt{in}$
$\{\ell_x \mapsto \circledast * \ell_x \mapsto_{\mathbf{g}} [t_0]\}$
$x_0 := 42$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [t_0]\}$
(ghost sharing)
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\gamma, t_0]\}$
$\{\ell_x \overset{1/2}{\longmapsto} 42 * \ell_x \overset{1/2}{\longmapsto} 42 * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [\gamma, t_0] * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [\gamma, t_0]\}$

$\{\ell_x \overset{1/2}{\longmapsto} 42 * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [\gamma, t_0]\}$	$\{\ell_x \overset{1/2}{\longmapsto} 42 * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [\gamma, t_0]\}$
$\texttt{let}\ x_1 = \texttt{retag}[\&](x_0)\ \texttt{in}$	$\texttt{let}\ x_2 = \texttt{retag}[\&](x_0)\ \texttt{in}$
$\{\ell_x \overset{1/2}{\longmapsto} 42 * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [\gamma, t_0] * \mathsf{SC}\,\gamma\,t_1\}$	$\{\ell_x \overset{1/2}{\longmapsto} 42 * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [\gamma, t_0] * \mathsf{SC}\,\gamma\,t_2\}$
$^*x_1$	$\texttt{let}\ x_3 = \texttt{retag}[\&](x_2)\ \texttt{in}$
$\{\ell_x \overset{1/2}{\longmapsto} 42 * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [\gamma, t_0] * \mathsf{SC}\,\gamma\,t_1\}$	$\{\ell_x \overset{1/2}{\longmapsto} 42 * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [\gamma, t_0] * \mathsf{SC}\,\gamma\,t_2 * \mathsf{SC}\,\gamma\,t_3\}$
	$^*x_3$
	$\{\ell_x \overset{1/2}{\longmapsto} 42 * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [\gamma, t_0] * \mathsf{SC}\,\gamma\,t_2 * \mathsf{SC}\,\gamma\,t_3\}$

$\{\ell_x \overset{1/2}{\longmapsto} 42 * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [\gamma, t_0] * \mathsf{SC}\,\gamma\,t_1 * \ell_x \overset{1/2}{\longmapsto} 42 * \ell_x \overset{1/2}{\longmapsto}_{\mathbf{g}} [\gamma, t_0] * \mathsf{SC}\,\gamma\,t_2 * \mathsf{SC}\,\gamma\,t_3\}$
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [\gamma, t_0] * \mathsf{SC}\,\gamma\,t_1 * \mathsf{SC}\,\gamma\,t_2 * \mathsf{SC}\,\gamma\,t_3\}$
(ghost forgetting)
$\{\ell_x \mapsto 42 * \ell_x \mapsto_{\mathbf{g}} [t_0] * \mathsf{SC}\,\gamma\,t_1 * \mathsf{SC}\,\gamma\,t_2 * \mathsf{SC}\,\gamma\,t_3\}$
$\texttt{free}(x_0)$
$\{\mathsf{SC}\,\gamma\,t_1 * \mathsf{SC}\,\gamma\,t_2 * \mathsf{SC}\,\gamma\,t_3\}$
(rule of consequence using $P * Q \Rightarrow P$ on the postcondition)
$\{\mathsf{True}\}$

Here, we have abbreviated $[\mathsf{GSharedRO}(\gamma), \mathsf{GUnique}(t_0)]$ to $[\gamma, t_0]$, and abbreviated $\mathsf{SBSetContains}\,\gamma\,t$ as $\mathsf{SC}\,\gamma\,t$. This proof uses the ghost sharing rule SB-Share to add the $\mathsf{GSharedRO}(\gamma)$ item before splitting the points-to assertion and the ghost stack assertion into fractions for the concurrent part. In each concurrent thread, we apply the H-SB-Read and H-SB-Retag-Shared rules to reason about reading and retagging. Note that by retagging to derive new shared references, we obtain $\mathsf{SBSetContains}\,\gamma\,t$ tokens.

As usual, after the concurrent part, the postconditions of both threads are combined, and hence we receive all of the $\mathsf{SBSetContains}\,\gamma\,t$ tokens created by individual threads. We recombine the fractions before applying ghost forgetting to get rid of the $\mathsf{GSharedRO}(\gamma)$ item, which brings $\mathsf{GUnique}(t_0)$ to the top of the ghost stack and allows us to deallocate.

Note that after deallocation, we still have the $\mathsf{SBSetContains}\,\gamma\,t$ tokens. However, these are not worth anything without a ghost stack that contains the set name γ. Hence, we use the rule of consequence to remove them from the postcondition.

Importantly, while reasoning about each thread, we only have resources corresponding to $\mathsf{SharedRO}$ items that that thread itself has added, because each thread does not "know" about items added by other threads concurrently.

4.7 Relation between SBSL and the Rust type system

We have now introduced all of the rules of SBSL, and applied them to several simple examples to show how the rules can be used to show the absence of undefined behavior related to Stacked Borrows. Moreover, we have shown that the SBSL rules can be used to give more abstract Hoare triples using the ghost forgetting rule. Finally, the rules of SBSL allow reasoning about simple forms of concurrency, where items are added or removed from the stack in a concurrent fashion.

How do we know that the rules of SBSL are in some sense "abstract enough"? We now briefly discuss the correspondence between SBSL and the Rust type system, providing a rough sketch for how the ghost forgetting rule and the ghost sharing rule can be used to model aspects of the Rust type system as formalized in RustBelt (Jung et al. 2018a). This provides an argument for why the rules of SBSL could serve as a foundation for porting the RustBelt safety proof to Stacked Borrows.

As discussed before, in the Rust type system, every reference has a lifetime: the lifetime of a reference starts when it is created, and the lifetime ends once the reference will never be used again. Similarly, in Stacked Borrows, a reference can be used while its tag appears in the stack, and can no longer be used after its tag has been removed from the stack. This similarity is no coincidence, since Stacked Borrows was designed based on the Rust type

system.

However, there is a difference between the Rust type system and Stacked Borrows: in the Rust type system, the lifetime of a reference typically ends *before* the item for that reference has been removed from the stack. This distinction can also be seen in the ghost forgetting rule: the ghost forgetting rule removes an item from the ghost stack *before* that item has been removed from the physical stack. Hence, the ghost forgetting rule in SBSL is closely related to the ending of a lifetime in the Rust type system.

A similar correspondence can be seen for the ghost sharing rule. In the Rust type system (as formalized in RustBelt), there is also a typing rule for converting mutable references into shared references. This conversion can be applied at any point in the typing derivation: there is no explicit program operation to perform the conversion. Similarly, the ghost sharing rule is a logical operation that can be performed at any point in a Hoare triple derivation, regardless of the program under consideration. Hence, ghost sharing in SBSL is closely related to converting a mutable reference to a shared reference in the Rust type system.

Based on these intuitions, we have performed some initial experiments integrating SBSL with the parts of the RustBelt model responsible for modeling references. In these experiments, we have proved some Hoare triples relating reborrowing (deriving a new reference with a shorter lifetime, which is tracked by the RustBelt type system) to retagging (a runtime operation in Stacked Borrows) for mutable references, shared references, and raw pointers. These Hoare triples make use of the *lifetime logic* (Jung et al. 2018a), which is used in RustBelt to model lifetimes and references. We do not describe these experiments in detail here, since understanding the lifetime logic requires explaining more advanced concepts in Iris that we consider out of scope. Further integration into RustBelt is left as future work.

Chapter 5

Model

In Chapter 4, we have presented SBSL and demonstrated how it can be used to reason about the behavior of some example $\lambda^{\mathsf{SB}}_{\mathsf{Rust}}$ programs. In this chapter, we give the Iris model of SBSL, describing how the rules of SBSL are implemented using Iris. Section 5.1 presents the substack relation, which is a slightly modified subsequence relation that is the main component in relating physical stacks and ghost stacks. Section 5.2 sketches how the rules of SBSL are shown to hold with respect to the operational semantics. Section 5.3 presents in more detail how the Iris ghost state mechanism is used to implement the ghost stack assertion and the set membership assertion. Section 5.4 describes the Iris state interpretation for SBSL, which makes the relation between the physical stacks and the ghost stacks fully precise. Finally, in Section 5.5, we state the adequacy theorem for SBSL, which implies that SBSL is sound, guaranteeing that programs verified using the rules of SBSL (e.g., by deriving a Hoare triple $\{\mathsf{True}\}\, e\, \{\mathsf{True}\}$) do not have undefined behavior.

5.1 Substack relation

In this section, we give a more precise account of how a ghost stack relates to its corresponding physical stack. The intuitive definition that we gave for ghost stacks is that they represent a subsequence of the corresponding physical stack. However, this intuitive definition is not entirely precise, because it is possible for the ghost stack to contain a GSharedRO item even if there are no SharedRO items in the physical stack at all.

This discrepancy can be dealt with by considering a physical stack that does not have any SharedRO items on top to have an *empty set* of SharedRO items on top. That is, if the physical stack for a location is

$$[\mathsf{Unique}(1), \mathsf{Unique}(0)]$$

then we can equivalently view this physical stack as

$$[\mathsf{SharedRO}(\varnothing), \mathsf{Unique}(1), \mathsf{Unique}(0)]$$

If we use the alternative representation where the physical stack always has a group of SharedRO items on top, then we can properly view

$$[\mathsf{SharedRO}(\varnothing), \mathsf{Unique}(0)]$$

as a "subsequence" of the physical stack

$$[\mathsf{Unique}(1), \mathsf{Unique}(0)]$$

despite the fact that that physical stack does not strictly speaking have any SharedRO items. This explains why the ghost stack can have a GSharedRO item despite the fact that the physical stack does not appear to have one. We account for this discrepancy in the relation between physical stacks and ghost stacks, although it could also be dealt with by always having an SharedRO item (potentially empty) on top of the stack in the operational semantics. In that case, there would be no need for a special subsequence relation.

Moreover, in the relation that relates physical stacks and ghost stacks we also ensure that physical stacks are well-formed: they should consists of at most one group of SharedRO items, followed by any number of Unique and SharedRW items. It is easy to see that this property always holds for physical stacks in Stacked Borrows: it is an invariant of the semantics. This fact is also mentioned in the original paper on Stacked Borrows (Jung et al. 2020). By building this invariant into the relation between physical stacks and ghost stacks, we get stronger assumptions about what physical stacks look like, which is useful in proving the SBSL rules. This well-formedness constraint also affects the rules of SBSL: the ghost sharing rule SB-Share only allows adding a GSharedRO item when the ghost stack does not already have one on top. This constraint derives from the well-formedness constraint that a physical stack can have at most one group of SharedRO items on top.

The well-formedness constraint on physical stacks and the handling of SharedRO items are captured in the *substack relation* $S_{\text{sub}} \leq S_{\text{phys}}$, which is a slightly modified version of the subsequence relation. It states that S_{sub} is a *substack* of S_{phys}, meaning it is a subsequence of the *normalized* version of the physical stack S_{phys}. Normalizing the stack means adding a $\mathsf{SharedRO}(\varnothing)$ item if the stack does not already have one on top and verifying that the stack is well-formed. This normalization step is expressed by the $\mathsf{StackNormalize}\ S_{\text{phys}}\ S_{\text{norm}}$ relation, which relates physical stacks S_{phys} to their normalized equivalents S_{norm}, adding a $\mathsf{SharedRO}(\varnothing)$ item if necessary. This relation uses StackIsTail to ensure that the tail of the stack consists only of Unique and SharedRW items, thereby enforcing the well-formedness constraint.

$$\textsc{Stack-Sub}\ \frac{\mathsf{StackNormalize}\ S_{\text{phys}}\ S_{\text{norm}} \qquad S_{\text{sub}}\ \text{is a subsequence of}\ S_{\text{norm}}}{S_{\text{sub}} \leq S_{\text{phys}}}$$

$$\textsc{StackNormalize-Shared}\ \frac{\mathsf{StackIsTail}\ \bar{\iota}}{\mathsf{StackNormalize}\ (\mathsf{SharedRO}(T) :: \bar{\iota})\ (\mathsf{SharedRO}(T) :: \bar{\iota})}$$

$$\textsc{StackNormalize-NoShared}\ \frac{\mathsf{StackIsTail}\ \bar{\iota}}{\mathsf{StackNormalize}\ \bar{\iota}\ (\mathsf{SharedRO}(\varnothing) :: \bar{\iota})}$$

$$\textsc{StackIsTail-Empty}\ \mathsf{StackIsTail}\ [] \qquad \textsc{StackIsTail-Cons}\ \frac{\mathsf{StackIsTailItem}\ \iota \qquad \mathsf{StackIsTail}\ \bar{\iota}}{\mathsf{StackIsTail}\ (\iota :: \bar{\iota})}$$

$$\textsc{StackIsTailItem-Unique}\ \mathsf{StackIsTailItem}\ (\mathsf{Unique}(t)) \qquad \textsc{StackIsTailItem-SharedRW}\ \mathsf{StackIsTailItem}\ (\mathsf{SharedRW}(\bot))$$

Figure 5.1: Rules for the substack relation.

This relation is the main component of the relation between physical stacks and ghost stacks, although there is also an additional step to relate $\mathsf{SharedRO}(\{t_1, t_2, \ldots, t_n\})$ items to the indirect representation $\mathsf{GSharedRO}(\gamma)$ using set names. Also note that the substack relation is just an ordinary mathematical relation: there are no Iris notions (such as propositions that express resource ownership) involved.

5.2 Proving the SBSL rules

This section discusses how the rules for deriving Hoare triples such as H-SB-Write are shown to be correct with respect to the operational semantics. We rely on the intuitive definitions for the assertions here, although the model behind the assertions is made more precise in Section 5.3 and Section 5.4. For example, consider the following instance of the H-SB-Write rule:

$$\begin{aligned}
&\{\ell \mapsto 5 * \ell \mapsto_{\mathbf{g}} [\mathsf{GUnique}(1), \mathsf{GUnique}(0)]\} \\
&\quad \langle \ell, 1 \rangle := 10 \\
&\{v.\ (v = \text{☣}) * \ell \mapsto 10 * \ell \mapsto_{\mathbf{g}} [\mathsf{GUnique}(1), \mathsf{GUnique}(0)]\}
\end{aligned}$$

According to the intuitive definitions of Hoare triples and the various assertions, this means that when ℓ holds the value 5 and has a physical stack that

has $[\mathsf{Unique}(1), \mathsf{Unique}(0)]$ as a substack, then $\langle \ell, 1 \rangle := 10$ does not cause undefined behavior and (if it terminates) produces the value $\lightning$, where afterwards ℓ holds the value 10 and the physical stack *still* has $[\mathsf{Unique}(1), \mathsf{Unique}(0)]$ as a substack.

The majority of the reasoning involved in proving such a Hoare triple consists of ordinary mathematical reasoning about lists, subsequences, etc., with relatively little Iris-specific reasoning involved. The Iris-specific reasoning only shows up in the Iris model for the ghost stack assertion and the set membership assertion, for which we use the intuitive definitions for now.

For example, in order to show the above Hoare triple, the main part of the proof (about the parts related to Stacked Borrows) has the following structure: assuming that the precondition holds, i.e., assuming that $[\mathsf{Unique}(1), \mathsf{Unique}(0)]$ is a substack of the current physical stack S_{phys} for location ℓ, we have to show the following two parts:

- No undefined behavior occurs: Since the above Hoare triple is for a write access, this boils down to showing that $\mathsf{WriteSingle}(t_1, S_{\text{phys}}) = S'_{\text{phys}}$ for some S'_{phys} (i.e., $\mathsf{WriteSingle}$ does not produce **fail**).
- Postcondition holds afterwards: We show that for all S'_{phys} such that $\mathsf{WriteSingle}(t_1, S_{\text{phys}}) = S'_{\text{phys}}$, we have that $[\mathsf{Unique}(1), \mathsf{Unique}(0)]$ is *still* a substack of S'_{phys}.

While proving this, we can assume that the current physical stack S_{phys} is well-formed (given by the substack relation), and we have to show that the next physical stack S'_{phys} is still well-formed (by re-establishing the substack relation).

In this case, we get that there is no undefined behavior from the fact that any item that occurs in the substack also occurs in the physical stack (and hence there is a write-granting item for tag 1 in the physical stack). The postcondition follows from the fact that $[\mathsf{Unique}(1), \mathsf{Unique}(0)]$ is preserved as a substack when writing using tag 1.

Hoare triples involving $\mathsf{GSharedRO}(\gamma)$ items are proved in a similar way. For example, if the precondition contains

$$\ell \mapsto_{\mathbf{g}} [\mathsf{GSharedRO}(\gamma), \mathsf{GUnique}(0)] * \mathsf{SBSetContains}\, \gamma\, 1$$

then according to the intuitive definitions of the assertions we know that the location ℓ has a physical stack that has $[\mathsf{SharedRO}(T), \mathsf{Unique}(0)]$ as a substack, where T is some set of tags that contains the tag 1. Then, in order to prove a Hoare triple such as

$$\{\ell \mapsto_{\mathbf{g}} [\mathsf{GSharedRO}(\gamma), \mathsf{GUnique}(0)] * \mathsf{SBSetContains}\, \gamma\, 1\}$$
$$\quad {}^*\langle \ell, 1 \rangle$$
$$\{v.\, (v = \lightning) * \ell \mapsto_{\mathbf{g}} [\mathsf{GSharedRO}(\gamma), \mathsf{GUnique}(0)] * \mathsf{SBSetContains}\, \gamma\, 1\}$$

We show that since the tag 1 occurs in the ghost stack inside a GSharedRO, there is a SharedRO(1) item in the physical stack, meaning read access is allowed with tag 1. Moreover, the substack $[\mathsf{SharedRO}(T), \mathsf{Unique}(0)]$ is preserved, since reading with a SharedRO item never alters the physical stack.

The view shifts for ghost forgetting and ghost sharing are also proved in a way similar to Hoare triples, except there we have that the physical state does not change. For example, for ghost forgetting, we have to show that when S_{sub} is a substack of the physical stack S_{phys}, and S'_{sub} is a subsequence of S_{sub}, then S'_{sub} is still a substack of S_{phys}. This follows directly from transitivity of the subsequence relation.

5.3 Ghost resources

We now explain how the new assertions in SBSL, the ghost stack assertion and the set membership assertion, are implemented in Iris. The techniques described in this section are standard in Iris, and do not represent a novel contribution.

We have seen that the ghost stack assertion $\ell \mapsto_{\mathbf{g}} G$ keeps track of a ghost stack G for each location ℓ, and we have given the intuitive definition that this ghost stack corresponds to a substack of the physical stack for location ℓ. Just like the points-to assertion, the ghost stack assertion behaves like a resource, in the sense that it cannot be duplicated (we do not have $P \Rightarrow P * P$ for the ghost stack assertion), although it can be split into fractional parts. Moreover, ghost stack assertions are "created" in the SBSL rule for allocation, where they appear in the postcondition without appearing in the precondition, and "destroyed" in the SBSL rule for deallocation. Moreover, ghost stacks can also be "updated" or "changed" using the ghost forgetting rule and the various SBSL rules for retagging.

Similarly, the set membership assertion $\mathsf{SBSetContains}\,\gamma\,t$ is another type of resource. Unlike the ghost stack assertion, it can be freely duplicated. Set membership assertions are "created" inside the SBSL retagging rule for shared references, where they are used to indicate that a new element has been added to a SharedRO group. There are no rules for "destroying" set membership assertions, although it is possible to "forget" about them using $P * Q \Rightarrow P$.

Both of these assertions are implemented using Iris' *ghost state* mechanism, which allows defining custom types of logical resources with customizable notions of ownership and sharing. In Iris, such custom types of resources are defined by giving a *resource algebra*, which specifies how resources are owned (e.g., whether they can be freely duplicated or split into fractions) and how the resources can be changed. Iris provides several built-in constructions for constructing resource algebras.

Since we have used only the standard resource algebra constructions of

Iris for this thesis, we will simply give an *axiomatic* description (a "ghost theory") of the ghost resources involved in the ghost stack assertion and the set membership assertion, and not describe in great detail how these "axioms" are derived from the underlying resource algebras.

For the reader who is familiar with Iris, the resource algebra used for the ghost stack assertion is $Auth(Loc \xrightarrow{\mathsf{fin}} Frac \times Ag(GStack))$ (essentially the same resource algebra as used for the fractional points-to assertion). The resource algebra for the set membership connective is $Auth(\wp^{\mathsf{fin}}(Tag))$ where $\wp^{\mathsf{fin}}(Tag)$ is the resource algebra of finite sets of tags with non-disjoint union.

Ghost stack assertions The ghost theory for ghost stack assertions is shown in Fig. 5.2. Most of the rules are view shifts, which are the basic mechanism used in Iris to describe changes (creation, destruction, updating) to ghost resources. Moreover, there is also an additional proposition $\mathsf{GStacksAuth}\ \Gamma$, that we have not introduced before. This is an *authoritative* resource, whose purpose is to keep track of a (finite) mapping Γ that maps each location ℓ to its current ghost stack G. Essentially, this makes it an authority that has a global view of all of the ghost stacks, whereas each individual ghost stack assertion only tracks the ghost stack of a single location. The role of this authoritative resource will become more clear in Section 5.4, where it is used for relating ghost stacks to physical stacks. None of the resources in this section have any inherent relation to physical program states.

We have the rule GStacks-Exclusive, which states that there is only a single authoritative resource: the authoritative resource cannot be duplicated.

Ghost stack assertions are created using GStacks-Alloc, which requires the authoritative resource, and produces a ghost stack assertion $\ell \mapsto_{\mathbf{g}} G$, as well as updating the mapping in the authoritative with the new ghost stack for location ℓ. Conversely, ghost stack assertions are destroyed using GStacks-Free, which consumes the ghost stack assertion and removes the ghost stack for location ℓ from the mapping Γ.

Ghost stacks are updated using GStacks-Set, which changes the ghost stack in the assertion and also reflects the updated ghost stack in the mapping Γ. Finally, we have a rule that says that the authoritative resource really has authoritative knowledge of all of the ghost stack assertions: the rule GStacks-Lookup says that if we have a ghost stack assertion for a location ℓ, then the ghost stack stored in the mapping Γ for that location matches the one in the ghost stack assertion. Also note that looking up the current ghost stack in the authoritative requires only fractional ownership of the ghost stack connective, whereas changing the ghost stack requires exclusive ownership. This is where the restrictions on modifying ghost stacks ultimately come from.

GSTACKS-ALLOC

$$\frac{\ell \notin \mathsf{dom}(\Gamma)}{\mathsf{GStacksAuth}\ \Gamma \Rrightarrow \mathsf{GStacksAuth}\ (\Gamma[\ell \leftarrow G]) * \ell \mapsto_{\mathbf{g}} G}$$

GSTACKS-LOOKUP

$$\mathsf{GStacksAuth}\ \Gamma * \ell \overset{q}{\mapsto}_{\mathbf{g}} G \Rightarrow \Gamma(\ell) = G$$

GSTACKS-SET

$$\mathsf{GStacksAuth}\ \Gamma * \ell \mapsto_{\mathbf{g}} G \Rrightarrow \mathsf{GStacksAuth}\ (\Gamma\left[\ell \leftarrow G'\right]) * \ell \mapsto_{\mathbf{g}} G'$$

GSTACKS-FREE

$$\mathsf{GStacksAuth}\ \Gamma * \ell \mapsto_{\mathbf{g}} G \Rrightarrow \mathsf{GStacksAuth}\ (\Gamma \setminus \ell)$$

GSTACKS-EXCLUSIVE

$$\mathsf{GStacksAuth}\ \Gamma * \mathsf{GStacksAuth}\ \Gamma' \Rightarrow \mathsf{False}$$

Figure 5.2: Ghost theory for ghost stacks assertions.

SET-CREATE

$$\mathsf{True} \Rrightarrow \exists \gamma.\, \mathsf{SetAuth}_\gamma\ T$$

SET-EXTEND

$$\mathsf{SetAuth}_\gamma\ T \Rrightarrow \mathsf{SetAuth}_\gamma\ (T \cup \{t\}) * \mathsf{SBSetContains}\,\gamma\,t$$

SET-ELEM

$$\mathsf{SetAuth}_\gamma\ T * \mathsf{SBSetContains}\,\gamma\,t \Rightarrow t \in T$$

SET-EXCLUSIVE

$$\mathsf{SetAuth}_\gamma\ T * \mathsf{SetAuth}_\gamma\ T \Rightarrow \mathsf{False}$$

SETCONTAINS-DUP

$$\mathsf{SBSetContains}\,\gamma\,t \Rightarrow \mathsf{SBSetContains}\,\gamma\,t * \mathsf{SBSetContains}\,\gamma\,t$$

Figure 5.3: Ghost theory for set membership assertions.

Set membership assertion Similarly, we now present the ghost theory for the set membership assertion in Fig. 5.3. Recall the intuition for the set membership connective $\mathsf{SBSetContains}\,\gamma\,t$ as meaning that t belongs to the set named γ. But where is this underlying set with the name γ actually "stored"? It is stored in another authoritative resource: $\mathsf{SetAuth}_\gamma\,T$ with T a set of tags. The purpose of this resource is to keep track of *all* of the elements contained in the set named γ.

We can create new sets using Set-Create, which produces a new authoritative set (with existentially-quantified name γ) initially holding the chosen set T. We can extend this set using Set-Extend, which extends the authoritative set and produces a $\mathsf{SBSetContains}\,\gamma\,t$ token for the newly-added element t. Given a $\mathsf{SBSetContains}\,\gamma\,t$ assertion, we can conclude that it actually belongs to the authoritative set for γ using Set-Elem. Finally, the authoritative set itself is exclusive (cannot be duplicated) (Set-Exclusive), and the $\mathsf{SBSetContains}\,\gamma\,t$ tokens are duplicable (SetContains-Dup).

Importantly, this set resource describes a set that can only accumulate more elements: it can be extended with additional tags, which provides a $\mathsf{SBSetContains}\,\gamma\,t$ token indicating the presence of the new element. There are no rules for removing elements from a set resource. This corresponds to the intuition that while a $\mathsf{GSharedRO}(\gamma)$ item occurs in the ghost stack, threads should be restricted to merely *adding* more SharedRO items without removing SharedRO items potentially added by other threads.

5.4 Linking physical stacks and ghost stacks

Now that we have described the ghost resources for ghost stack assertions and set membership assertions, we have all the tools needed to describe precisely how the physical stacks and the ghost stacks are related.

Iris provides a generic definition for Hoare triples (more precisely, it defines weakest preconditions, which can be used to implement Hoare triples), that is not specific to any particular programming language. In order to use this generic definition for a particular programming language, one must provide a *state interpretation*, which is an Iris predicate on physical program states that *must hold at every (reachable) program state*. Recall that Iris propositions do not just represent "knowledge" or "facts", but can also describe ownership of resources. Hence, the state interpretation can also own resources. The state interpretation is typically used in Iris to describe how the physical state of the program is related to the ghost state. We use it to relate each ghost stack to its corresponding physical stack.

We only describe the part of the state interpretation that relates to Stacked Borrows, leaving out the parts of the state interpretation used for the points-to assertion. We build up to the state interpretation in several steps.

First, we have the Iris predicate that relates physical items like $\mathsf{Unique}(0)$ to ghost items like $\mathsf{GUnique}(0)$:

$$\mathsf{SBRelItem}(\iota, g) = \begin{cases} \iota = \mathsf{Unique}(t) & \text{if } g = \mathsf{GUnique}(t) \\ \iota = \mathsf{SharedRW}(\bot) & \text{if } g = \mathsf{GSharedRW}(\bot) \\ \exists T. \iota = \mathsf{SharedRO}(T) * \mathsf{SetAuth}_\gamma\ T & \text{if } g = \mathsf{GSharedRO}(\gamma) \end{cases}$$

This makes use of the higher-order nature of Iris, since it is a function that produces an Iris proposition. We have that Unique and $\mathsf{SharedRW}$ have to match exactly (up to different syntax), whereas for $\mathsf{GSharedRO}(\gamma)$ items there should be an authoritative set with name γ holding the tags T that appear in the physical $\mathsf{SharedRO}$ item. This relation is what links the indirect representation using set names to the sets in the physical stacks: it is ultimately what allows us to conclude from a set membership connective $\mathsf{SBSetContains}\,\gamma\,t$ that the tag t actually appears in the physical stack in a $\mathsf{SharedRO}$ item (using the Set-Elem rule from Section 5.3).

This relation on individual items can be lifted pointwise to become a relation on stacks (lists of items):

$$\mathsf{SBRelItems}(S, G) = \bigstar_{1 \le i \le |S| = |G|} \mathsf{SBRelItem}(S^i, G^i)$$

where A^i denotes the i-th element of the list A and $|A|$ denotes the length of the list A.

Based on this, we can now give the relation between a physical stack and its corresponding ghost stack, which makes use of the substack relation defined in Section 5.1:

$$\mathsf{SBRelStack}(S_{\text{phys}}, G) = \exists S_{\text{sub}}.\ (S_{\text{sub}} \le S_{\text{phys}}) * \mathsf{SBRelItems}(S_{\text{sub}}, G)$$

which simply says that S_{phys} has a substack S_{sub}, which is related pointwise using $\mathsf{SBRelItems}$ to G.

Again, we can lift this relation (between physical stacks and ghost stacks) pointwise to become a relation between mappings from locations to physical stacks and mappings from locations to ghost stacks:

$$\mathsf{SBRelStacks}(\xi, \Gamma) = \bigstar_{\ell \in \mathsf{dom}(\xi) = \mathsf{dom}(\Gamma)} \mathsf{SBRelStack}(\xi(\ell), \Gamma(\ell))$$

Finally, we use this to give the state interpretation for the part of the state that relates to Stacked Borrows:

$$\mathsf{SBInterp}(\xi) = \exists \Gamma.\ \mathsf{GStacksAuth}\ \Gamma * \mathsf{SBRelStacks}(\xi, \Gamma)$$

This is an Iris predicate on a part of the physical program state, which we use as part of the state interpretation in the generic definition of weakest preconditions (which can be used to define Hoare triples). This Iris predicate should

hold at every reachable program state, and it contains the GStacksAuth Γ authoritative resource that stores the mapping Γ that maps each location to its current ghost stack. The SBRelStacks predicate ensures that each ghost stack is indeed a substack of its corresponding physical stack.

While proving a Hoare triple, we are assumed to allow that the state interpretation holds in the initial state (and hence, we can also use any resources occurring in the state interpretation, such as the various authoritative resources, to conclude what the physical stacks look like based on ghost stacks), and we have to re-establish that the state interpretation still holds after taking steps (and we are allowed to apply the view shifts in Section 5.3 to manipulate ghost resources to prove this).

There is an additional technical detail that we mention for readers familiar with Iris. For technical reasons, it is normally only possible to access the ghost resources inside the state interpretation while proving a Hoare triple, and not while proving a view shift such as the ghost forgetting rule or the ghost sharing rule (which also involve changing ghost stacks, and therefore requires the authoritative resource that holds the mapping with all of the ghost stacks). Hence, we have wrapped the "state interpretation" in an Iris invariant and linked this invariant to the physical program state in the actual state interpretation using another authoritative resource, a standard pattern in Iris.

5.5 Adequacy

In this section, we state the adequacy theorem for SBSL, which implies that SBSL guarantees the absence of undefined behavior.

Theorem (Adequacy of SBSL)**.** *Suppose φ is a first-order predicate on values (i.e., not an Iris predicate, but an ordinary mathematical statement) and suppose that $\{\mathsf{True}\}\ e\ \{v.\ \varphi(v)\}$ is derivable in SBSL. Then we have the following:*

- *No undefined behavior: for any machine configuration reachable in the execution of e, neither e itself nor any of the threads e_i created by it are stuck (i.e., if they are not values, they can take a step).*
- *Postcondition: in any machine state reachable in the execution of e where e has reduced to a value v, $\varphi(v)$ holds.*

The adequacy theorem follows directly from the Iris adequacy theorem, which is stated in terms of the generic definition of Hoare triples (defined in terms of weakest preconditions) in Iris.

Adequacy implies soundness of the logic: if we derive a Hoare triple $\{\mathsf{True}\}\ e\ \{\mathsf{True}\}$, then e is safe to execute: it does not cause undefined behavior.

Chapter 6

Related work

RustBelt Relaxed The RustBelt Relaxed work (Dang et al. 2020) develops another extension of RustBelt to consider a different operational semantics compared to that used in the original RustBelt work (Jung et al. 2018a). Specifically, RustBelt Relaxed considers an operational semantics with a more *relaxed memory model*, in which certain memory operations provide weaker (relaxed) guarantees about the order in which different threads observe changes to the memory. Such operations are typically cheaper to implement on modern multicore processors compared to *sequentially consistent* operations, which require that changes to the memory are instantaneously visible to all threads in a single, global order.

The general approach of RustBelt Relaxed is similar to that of SBSL, in the sense that RustBelt Relaxed also replaces the program logic on top of which RustBelt is built. RustBelt Relaxed also introduces some important changes to the parts of RustBelt built on top of the program logic. The program logic used by RustBelt Relaxed is an extended version of iGPS (Kaiser et al. 2017). RustBelt Relaxed goes beyond this thesis by re-establishing the RustBelt safety proof with respect to a relaxed memory model.

Logical abstractions on top of physical state The general idea of having ghost state or "logical state" that is more abstract than the underlying physical state is not new. Dinsdale-Young et al. (2010a) extend separation logic with the notion of *concurrent abstract predicates*, which allows reasoning about a shared, concurrent data structure using an abstract specification that provides the illusion of a data structure consisting of disjoint resources, despite the fact that the actual, concrete implementation of the data structure might not be represented in a disjoint way in memory. This illusion of disjoint resources also referred to as *fiction of disjointness* or *fictional separation* (Dinsdale-Young et al. 2010a,b).

In a somewhat similar way, our notion of ghost stacks in SBSL provides the illusion that operations such as reading, writing, and creating shared references

do not change any state (because ghost stacks remains unchanged), despite the fact that those operations might modify the physical state. SBSL achieves this by decomposing the changes to physical stacks into separate, orthogonal operations, implemented by the ghost forgetting and ghost sharing rules. This means that SBSL essentially provides a *fiction of separate operations* for Stacked Borrows, because SBSL decomposes certain physical operations into several logical operations. The main novelty in SBSL is not the concept of logical operations itself, but the particular way in which the operations of Stacked Borrows are decomposed.

The notion of abstract "logical state" (of which SBSL ghost stacks are a particular example) appears in a general form in the *Views Framework* (Dinsdale-Young et al. 2013), which describes a general framework for defining logical resources and relating those to the physical state. The general Iris (Jung et al. 2016, 2018b, 2015; Krebbers et al. 2017a) strategy for defining a program logic for a new language by means of a state interpretation (the strategy followed for SBSL) is heavily inspired by this. Various more expressive forms of logical state have also been introduced in the HOCAP (Svendsen et al. 2013) and iCAP (Svendsen et al. 2014) logics, of which Iris is a modern descendant. SBSL itself does not make use of these more expressive forms of logical state, but RustBelt (Jung et al. 2018a) does rely on them.

Program logics for undefined behavior Various program logics have been developed for programming languages that have more sophisticated forms of undefined behavior. The λ_{Rust} language developed for RustBelt (Jung et al. 2018a) has a form of undefined behavior that occurs when a program contains a *data race*, which describes the situation where two threads attempt to access the same location and at least one threads writes to the location. Data races are only undefined behavior when they occur using *non-atomic accesses*, which we have not included in the λ_{Rust} fragment described in this thesis. The operational semantics for λ_{Rust} includes a "data race detector" that detects when a data race occurs and causes the program to become stuck (i.e., the program has undefined behavior) when this occurs. The program logic for RustBelt (Jung et al. 2018a) accounts for this form of undefined behavior using the rules of its program logic (and SBSL inherits this from the λ_{Rust} program logic).

In the context of undefined behavior related to pointer aliasing, Krebbers (2015) has developed a separation logic that accounts for aliasing-related undefined behavior in the C programming language. In C, it is considered undefined behavior to access, say, a variable of type `int` (integer) through a pointer of type `float` (floating-point number). This type-based restriction on pointer aliasing is called the *strict aliasing rule*. This restriction allows C compilers to make stronger assumptions about pointer aliasing and in turn enables better optimizations. This makes the goal of the strict aliasing rule

similar to that of Stacked Borrows, although Stacked Borrows is based on Rust's borrowing and lifetimes, which do not exist in C. The logic developed by Krebbers accounts for strict aliasing, although it does not have a concept similar to *ghost stacks*, and instead the "logical state" is very similar to the underlying physical state.

Chapter 7

Conclusion

We have presented SBSL, a separation logic for Stacked Borrows. We have presented several examples, showing how this logic can be applied to reason about patterns commonly appearing in Rust code. Moreover, we show that the logic is better able to abstract away implementation details and reason about concurrency compared to a naive approach. We have fully formalized SBSL using Iris inside the Coq proof assistant, and we have provided a machine-checked proof of its soundness with respect to the operational semantics.

7.1 Future work

Scaling up to the full Stacked Borrows semantics The current version of SBSL only considers a simplified version of the full Stacked Borrows semantics presented in (Jung et al. 2020). The full Stacked Borrows semantics also considers locations with *interior mutability*. For such locations, it is allowed to perform writes through a shared references `&T`, which normally provide only immutable access. Interior mutability is important to implement libraries such as `Mutex`, which provides multiple threads with the ability to access the same location for reading and writing by using a lock to ensure that only one thread at a time has access to the location. Stacked Borrows supports interior mutability using an additional type of stack item that we have not considered. The basic principles of adding and removing items on the stack remain the same. We have developed a prototype implementation in Coq of a version of SBSL that supports interior mutability. Supporting the original formulation of the semantics interior mutability turned out to be difficult, and therefore we considered a slightly altered version of the semantics. Due to lack of space, we do not describe this here. As future work, this prototype implementation could be extended and also be used to guide potential changes to the Stacked Borrows semantics to make it easier to reason about.

Moreover, the full Stacked Borrows version also contains *protectors*, which

are special markers that can be added to stack items to indicate that they should not be removed while a certain function call is still ongoing. The motivation for protectors is to provide additional optimization potential. Protectors are a mostly orthogonal extension to the Stacked Borrows semantics in this thesis. An interesting future direction would therefore be to add support for protectors to the program logic.

Integrating SBSL further into RustBelt (Relaxed) The other main direction for future work is to integrate SBSL further into RustBelt (Jung et al. 2018a) (and potentially RustBelt Relaxed (Dang et al. 2020)). Ultimately, this could be used to demonstrate that Rust's safety guarantees still hold in the presence of the additional type of undefined behavior that Stacked Borrows introduces into the language. We have already carried out some initial experiments in updating some of the core parts of RustBelt, namely those related to modeling references and lifetimes. Those experiments, which are part of the Coq development for SBSL, seem to suggest that SBSL might provide a suitable basis for an updated version of RustBelt, although further research is required to perform a full evaluation of its suitability.

Bibliography

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson (2005). "Permission Accounting in Separation Logic". In: *POPL*. URL: https://doi.org/10.1145/1040305.1040327.

John Boyland (2003). "Checking Interference with Fractional Permissions". In: *SAS*. URL: https://doi.org/10.1007/3-540-44898-5%5C_4.

Stephen Brookes (2007). "A semantics for concurrent separation logic". In: *Theor. Comput. Sci.* 375.1-3, pp. 227–270. URL: https://doi.org/10.1016/j.tcs.2006.12.034.

Coq Development Team (2020). *The Coq Proof Assistant*. Version 8.12.0. URL: https://doi.org/10.5281/zenodo.4021912.

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer (2020). "RustBelt Meets Relaxed Memory". In: *POPL*. URL: https://doi.org/10.1145/3371102.

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang (2013). "Views: Compositional Reasoning for Concurrent Programs". In: *POPL*. URL: https://doi.org/10.1145/2429069.2429104.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis (2010a). "Concurrent Abstract Predicates". In: *ECOOP*. URL: https://doi.org/10.1007/978-3-642-14107-2%5C_24.

Thomas Dinsdale-Young, Philippa Gardner, and Mark J. Wheelhouse (2010b). "Abstraction and Refinement for Local Reasoning". In: *VSTTE*. URL: https://doi.org/10.1007/978-3-642-15057-9%5C_14.

C. A. R. Hoare (1969). "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10, pp. 576–580. URL: https://doi.org/10.1145/363235.363259.

Ralf Jung (2020). "Understanding and Evolving the Rust Programming Language". PhD thesis. Saarland University, Saarbrücken, Germany. URL: https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647.

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer (2020). "Stacked Borrows: An Aliasing Model for Rust". In: *POPL*. URL: https://doi.org/10.1145/3371109.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer (2018a). "RustBelt: Securing the Foundations of the Rust Programming Language". In: *POPL*. URL: https://doi.org/10.1145/3158154.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer (2016). "Higher-order Ghost State". In: *ICFP*. URL: https://doi.org/10.1145/2951913.2951943.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer (2018b). "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *J. Funct. Program.* URL: https://doi.org/10.1017/S0956796818000151.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer (2015). "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning". In: *POPL*. URL: https://doi.org/10.1145/2676726.2676980.

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis (2017). "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris". In: *ECOOP*. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2017.17.

Robbert Krebbers (2015). "The C standard formalized in Coq". PhD thesis. Radboud Universiteit Nijmegen.

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer (2018). "MoSeL: a General, Extensible Modal Framework for Interactive Proofs in Separation Logic". In: *ICFP*. URL: https://doi.org/10.1145/3236772.

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal (2017a). "The Essence of Higher-Order Concurrent Separation Logic". In: *ESOP (LNCS)*. URL: https://doi.org/10.1007/978-3-662-54434-1%5C_26.

Robbert Krebbers, Amin Timany, and Lars Birkedal (2017b). "Interactive Proofs in Higher-Order Concurrent Separation Logic". In: *POPL*. URL: http://dl.acm.org/citation.cfm?id=3009855.

Nicholas D. Matsakis and Felix S. Klock (2014). "The Rust Language". In: *Ada Lett.* URL: https://doi.org/10.1145/2692956.2663188.

Peter W. O'Hearn (2007). "Resources, Concurrency, and Local Reasoning". In: *Theor. Comput. Sci.* 375.1-3, pp. 271–307. URL: https://doi.org/10.1016/j.tcs.2006.12.035.

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang (2001). "Local Reasoning about Programs that Alter Data Structures". In: *CSL*. URL: https://doi.org/10.1007/3-540-44802-0%5C_1.

John C. Reynolds (2002). "Separation Logic: A Logic for Shared Mutable Data Structures". In: *LICS*. URL: https://doi.org/10.1109/LICS.2002.1029817.

Rust team (2020). *The Rust Programming Language*. URL: https://rust-lang.org/.

Kasper Svendsen and Lars Birkedal (2014). "Impredicative Concurrent Abstract Predicates". In: *ESOP (LNCS)*. URL: https://doi.org/10.1007/978-3-642-54833-8%5C_9.

Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson (2013). "Modular Reasoning about Separation of Concurrent Data Structures". In: *ESOP (LNCS)*. URL: https://doi.org/10.1007/978-3-642-37036-6%5C_11.

Gavin Thomas (2019). *A proactive approach to more secure code*. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code. Accessed: 2020-12-10.