DISS. ETH NO. 30105

LEVERAGING UNIQUENESS FOR MODULAR VERIFICATION OF HEAP-MANIPULATING PROGRAMS

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES
(Dr. sc. ETH Zurich)

presented by

VYTAUTAS ASTRAUSKAS

MSc ETH CS, ETH Zurich

born on 28 July 1990

accepted on the recommendation of

Prof. Dr. Peter Müller, examiner
Dr. Claude Marché, co-examiner
Prof. Dr. Alexander J. Summers, co-examiner
Prof. Dr. Martin Vechev, co-examiner

2024

Colophon
This thesis was typeset with the help of KOMA-Script and LaTeX using the kaobook class.

Abstract

With software's ever-increasing role in human lives, ensuring its correctness is crucial. Deductive software verification enables formally proving that a program is functionally correct. However, verifying imperative heap-manipulating programming languages is notoriously difficult and requires complex specifications in powerful logics like separation logic. This complexity is a major obstacle to more widespread verification of imperative heap-manipulating programs.

In this thesis, we present a verification approach that, for common cases, is as easy to use as the approaches based on type systems while allowing the use of more powerful reasoning techniques for the parts of the project that require them. Our approach exploits unique properties of Rust, a systems programming language that aims to be a safe replacement for C and C++. For the safe fragment of the language, the Rust compiler ensures memory safety and gives strong disjointness guarantees. We present a novel verification approach that uses the type information from the compiler to synthesize a *core proof* that ensures memory safety and captures disjointness information in a flavour of separation logic suitable for automation. Users can write functional specifications in code annotations using a specification language based on Rust expressions; our technique automatically integrates them into the core proof, enabling modular verification of functional properties. We have implemented our approach for a subset of safe Rust and evaluated it on thousands of examples. Our evaluation shows that we can generate the core proof reliably. As a result, the users are entirely shielded from the underlying complex logic, enabling them to verify safe Rust programs at the programming language's abstraction level with significantly fewer and simpler annotations than other approaches require.

The guarantees provided by the Rust compiler come at a cost: some patterns, such as cyclic data structures, are hard or impossible to implement in safe Rust. As an escape hatch, Rust has an unsafe subset that allows using operations whose safety the compiler cannot guarantee, such as dereferencing C-style raw pointers. If a programmer uses the unsafe subset, they become responsible for ensuring memory safety. The intention is that programmers use unsafe Rust sparingly and hide it behind safe abstractions that enable building safe clients. To understand how and why programmers use unsafe code and how best we can support them, we conducted an empirical study investigating a large corpus of Rust projects. We found that a substantial part of Rust projects (more than one out of five) published in the Rust package registry contain some unsafe code, and most of it is used for calling unsafe functions and manipulating C-style raw pointers.

To support programmers writing unsafe code, we extended our verification approach with support for verifying memory safety and functional correctness of mixed safe and unsafe Rust code where unsafe code may call unsafe functions and manipulate raw pointers. We enable verifying unsafe code by exposing powerful specification primitives based on separation logic. We exploit the non-aliasing guarantees provided by Rust's operational semantics to keep the verification of surrounding safe code lightweight. Our extended version of the approach retains the key property of the original one: verifying safe parts of the code remains lightweight, with the user being shielded from the complex underlying logic with which they only need to interact if they use unsafe code. This property enables users to use simple constructs for most code and switch to more powerful parts of our approach when required.

Zusammenfassung

Da die Rolle von Software im menschlichen Leben immer größer wird, ist die Sicherstellung ihrer Korrektheit von entscheidender Bedeutung. Die deduktive Softwareverifikation ermöglicht den formalen Nachweis, dass ein Programm funktional korrekt ist. Allerdings ist die Überprüfung imperative Programmiersprachen, die den Heap manipulieren, bekanntermaßen schwierig und erfordert komplexe Spezifikationen in leistungsstarken Logiken wie der Separationslogik. Diese Komplexität ist ein großes Hindernis für eine umfassendere Verifikation heapmanipulierender Programme.

In dieser Arbeit stellen wir einen Verifikationsansatz vor, der für übliche Fälle genauso einfach zu verwenden ist wie Ansätze basierend auf Typsystemen und gleichzeitig die Verwendung leistungsfähigerer Argumentationstechniken für die Teile des Projekts ermöglicht, die solche Techniken erfordern. Unser Ansatz nutzt die einzigartigen Eigenschaften von Rust, einer Systemprogrammiersprache, die ein sicherer Ersatz für C und C++ sein soll. Für das sichere Fragment der Sprache sorgt der Rust-Compiler für Speichersicherheit und gibt starke Nicht-Aliasing-Garantien. Wir stellen einen neuartigen Verifikationsansatz vor, der die Typinformationen des Compilers verwendet, um einen *Kernbeweis (Core Proof)* zu synthetisieren, der Speichersicherheit gewährleistet und Disjunktheitsinformationen in einer für Automatisierung geeigneten Variante der Separationslogik erfasst. Benutzer können funktionale Spezifikationen in Annotationen im Code schreiben, indem sie eine Spezifikationssprache verwenden, die auf Rust-Ausdrücken basiert; unsere Technik integriert sie automatisch in den Kernbeweis und ermöglicht so eine modulare Überprüfung funktionaler Eigenschaften. Wir haben unseren Ansatz für eine Teilmenge von sicherem Rust implementiert und den Ansatz anhand tausender Beispiele evaluiert. Unsere Auswertung zeigt, dass wir den Kernbeweis zuverlässig erstellen können. Dadurch sind die Benutzer vollständig von der zugrunde liegenden komplexen Logik abgeschirmt und können sichere Rust-Programme auf der Abstraktionsebene der Programmiersprache mit deutlich weniger und einfacheren Annotationen beweisen, als andere Ansätze erfordern.

Die vom Rust-Compiler bereitgestellten Garantien haben ihren Preis: Einige Muster, wie z. B. zyklische Datenstrukturen, sind in sicherem Rust nur schwierig zu implementieren oder gar nicht implementierbar. Als Ausweichlösung verfügt Rust über ein unsicheres (unsafe) Fragment der Sprache, das die Verwendung von Operationen ermöglicht, deren Sicherheit der Compiler nicht garantieren kann, wie etwa die Dereferenzierung von Rohzeigern im C-Stil. Wenn ein Programmierer das unsichere Fragment verwendet, dann ist er für die Gewährleistung der Speichersicherheit verantwortlich. Die Absicht besteht darin, dass Programmierer unsicheres Rust sparsam verwenden und es hinter sicheren Abstraktionen verstecken, die den Aufbau sicherer Clients ermöglichen. Um zu verstehen, wie und warum Programmierer unsicheren Code verwenden und wie wir sie am besten unterstützen können, haben wir eine empirische Studie durchgeführt, die eine große Anzahl von Rust-Projekten untersucht. Wir haben festgestellt, dass ein erheblicher Teil der Rust-Projekte (mehr als jedes fünfte), die in der Package Registry für Rust veröffentlicht werden, unsicheren Code enthält und der größte Teil davon zum Aufruf unsicherer Funktionen und zur Manipulation von Rohzeigern im C-Stil verwendet wird.

Um Programmierer beim Schreiben von unsicherem Code zu unterstützen, haben wir unseren Verifikationsansatz um Unterstützung für die Überprüfung der Speichersicherheit und der funktionalen Korrektheit von Code, der sowohl aus sicherem als auch unsicherem Rust besteht, erweitert, bei dem unsicherer Code unsichere Funktionen aufrufen und Rohzeiger manipulieren kann. Wir ermöglichen die Überprüfung unsicheren Codes, indem wir leistungsstarke Spezifikationsprimitive basierend auf der Separationslogik zur Verfügung stellen. Wir nutzen die Nicht-Aliasing-Garantien der operationellen Semantik von Rust, um die Überprüfung des umgebenden sicheren Codes simpel zu halten. Unsere erweiterte Version des Ansatzes behält die Schlüsseleigenschaft des ursprünglichen Ansatzes bei: Die Überprüfung sicherer Teile des Codes bleibt einfach, wobei der Benutzer von der komplexen zugrunde liegenden Logik abgeschirmt wird, mit der er nur interagieren muss, wenn er unsicheren Code verwendet. Diese Eigenschaft ermöglicht es Benutzern, einfache Konstrukte für den größten Teil des Codes zu verwenden und bei Bedarf zu leistungsfähigeren Teilen unseres Ansatzes zu wechseln.

Acknowledgements

In my journey through doctoral studies, many people directly and indirectly shaped me and my work. Thank you all! Without *you*, I would not have reached the point where I am now.

I am deeply grateful to Peter Müller for giving me a life-changing chance to be part of his wonderful team. Thank you, Peter, for being a role model and showing me how to be a great researcher, teacher, and supervisor. Thank you also for your patience and for allowing me to experiment and learn through my mistakes.

I am also deeply grateful to Alex Summers for his continuous support and supervision. Thank you, Alex, for guiding me through this journey and explaining how to conduct excellent research. Thank you also for teaching me attention to detail, which is crucial for getting things right.

I would also like to thank my co-examiners, Claude Marché and Martin Vechev, for taking time from their busy schedules to serve on my doctoral examination committee, review my thesis, and give me valuable feedback, which enabled me to improve it significantly.

My internship at Amazon Web Services was an exciting part of my doctoral journey. I would like to thank my internship supervisor, Rajeev Joshi, for guiding and supporting me throughout my entire internship, James Bornholt for insightful technical discussions, Rustan Leino for exciting (and way too short) lunch discussions about Dafny, and the Amazon support team for bringing me safely home in the middle of the pandemic.

Working on a completely new programming language whose compiler has no documentation is hard. I would like to thank Ariel Ben-Yehuda, Oli Scherer, Niko Matsakis, and many other wonderful folks in the Rust community for introducing me to the Rust compiler internals and being extremely open to supporting our needs. I would also like to thank Ralf Jung for our many insightful discussions about the meaning of Rust.

As I mentioned, during my journey, I was part of a wonderful team of people: Alexandra Bugariu, Anqi Li, Arshavir Ter-Gabrielyan, Aurel Bílý, Caterina Urban, Christoph Matheja, Dimitar Asenov, Dionisios Spiliopoulos, Federico Poli, Felix Wolf, Fábio Pakk Selmi-Dei, Gaurav Parthasarathy, Hermann Lehner, Jonás Fiala, João Pereira, Jérôme Dohrau, Lea Brugger, Linard Arquint, Lucas Brutschy, Malte Schwerhoff, Marco Eilers, Maria Christakis, Martin Clochard, Michael Sammler, Milos Novacek, Nicolas Klose, Thibault Dardinier, Uri Juhasz, Valentin Wüstholz, Wytse Oortwijn, and Yushuo Xiao, who were recently joined by Ralf's students Isaac van Bakel, Johannes Hostert, Max Vistrup, and Rudy Peterson. Thank you all for the exciting lunch discussions, feedback on my ideas, the review of paper artefacts and parts of my thesis, the collaborations in research, teaching, and student supervision, and, in general, the exciting time together. I would also like to thank Marlies Weissert and Sandra Schneider for smoothly handling all administrative (and not only) questions.

A large part of my doctoral journey was teaching and student supervision. I would like to thank everyone with whom I had a chance to work together and grow as a teacher. In particular, I would like to thank Malte Schwerhoff, Felix Friedrich, and Ralf Sasse for trusting me with the head teaching assistant position, which gave me a lot of new experience. I would also like to thank the Bachelor's and Master's students whom I had the opportunity to supervise: Ahmed Rayan, Benjamin Schmid, Constantin Müller, David Blaser, Dominic Dietler, Janis Peyer, Jonas Maier, Julian Dunskus, Karuna Grewal, Lukas Friedlos, Markus Limbeck, Matthias Erdin, Maxwell Yang, Mohammad Abdulmoneim, Nicolas Trüssel, Nicolas Winkler, Olivia Furrer, Omar Tarek, Pascal Huber, Thomas Hader, Till Arnold, and William Seddon. Each of you enabled me to understand the field of program verification better and contributed a piece to this thesis. Thank you all for working with me!

Last but not least, I would like to thank my family and friends: without you, I would not have made this journey!

Contents

1 Introduction

Modern human life is hard to imagine without systems controlled by software. Therefore, it is crucial to ensure that this software is correct. However, the methods for ensuring software correctness used by most developers, such as code reviews, linting for problematic patterns, and testing, help improve software correctness but cannot give any guarantees. Therefore, we need to give programmers tools that provide formal guarantees. For such tools to be adopted, they must be *sufficiently expressive* (support the code that programmers write in practice), be as *automatic* as possible, *scale* to realistic systems, have a *low entry bar* (require minimal training for basic use and minimal upfront investment), and allow programmers to *focus* on what they care about (for example, the critical parts of the system). Unfortunately, satisfying all these requirements is challenging.

Deductive software verification can be used to prove that a program is functionally correct. A simple way of specifying basic functional properties is by using runtime assertions. Consider the Java method below that takes two counters and increments one. We[1] want to once and for all prove that the assertion on line 7 checking that the value of the other counter did not change never fails.

```
1 public static void resetAndTest(Counter x, Counter y)
2 {
3     int currentValue = x.getValue();
4     y.resetValue();
5
6     // Will fail if x == y
7     assert(currentValue == x.getValue());
8 }
```

However, proving this seemingly simple assertion is complicated because we need to ensure that the context calling this method guarantees the following four properties:

1. The references `x` and `y` are non-null.
2. The call to `y.resetValue()` does not affect the value returned by `x.getValue()`.
3. No other thread can mutate the value stored in `x` concurrently.
4. The result of `x.getValue()` depends only on its arguments (for example, it internally does not use randomness).

If we tried to prove the entire program at once, we could potentially try to automatically prove some of these properties inside a tool without exposing them to the user. However, realistic software systems are large. The key technique programmers use to keep the development process manageable is splitting them into independent modules that are small enough to be comprehensible and have clearly defined interfaces. Similarly, scaling verification to realistic systems requires using

1: Following the academic tradition, we use the first person plural to refer to the work done by the author, regardless of whether it was done alone or in collaboration with others. For details, please see Section 1.5.

modular techniques. The standard way for achieving modular deductive verification is based on *contracts* [1] that for each method define in a mathematically rigorous way what a method is allowed to assume–the method's *preconditions*–and what it promises to guarantee–the method's *postconditions*. The contracts are usually expressed using assertions in some logic, types, or a combination of both. For example, we can express property 1 by using the following precondition (we use VeriFast [2] syntax):

[1]: Meyer (1997), *Object-Oriented Software Construction, 2nd Edition*

[2]: Jacobs et al. (2011), 'VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java'

```
1 public static void resetAndTest(Counter x, Counter y)
2 //@ requires x != null && y != null
```

Alternatively, type annotations such as `[NotNull] x` proposed in [3] could be used to indicate that the parameters are non-null. Similarly, many verifiers support annotating methods as pure, which captures property 4.

[3]: Fähndrich et al. (2003), 'Declaring and checking non-null types in an object-oriented language'

Properties 2 and 3 are instances of the so-called *frame problem*. The term "frame problem" was first introduced in artificial intelligence research [4] to talk about the problem of expressing information about what remains unchanged by an event. Later, the program verification [5] community adopted the term for expressing what is not modified by a function call. Since concurrent programs became mainstream, we need to think about both changes made by the current and other threads. Therefore, when reasoning about modern programs, the question is what knowledge we can assume to be stable.

[4]: McCarthy et al. (1969), 'Some philosophical problems from the standpoint of artificial intelligence'

[5]: Borgida et al. (1995), 'On the Frame Problem in Procedure Specifications'

In mainstream programming languages such as Java, the frame problem is made challenging by the use of a *global heap* and *unrestricted mutable aliasing*. For instance, in our example above, the assertion would fail if the caller provided the same argument as both `x` and `y`. A naïve way to prevent this case would be adding a precondition that `x != y`. However, this precondition would not be sufficient if the `Counter` objects had an internally shared state. While it is unlikely to be the case for a simple counter, internal sharing is not uncommon for more complex data structures such as trees. Therefore, to prevent `y.resetValue()` affecting `x.getValue()` we need to ensure that the memory region modified by `y.resetValue()` is *disjoint* from the region read by `x.getValue()`. Since we aim for modular verification, we want to respect information hiding and avoid exposing to the clients the concrete memory locations accessed by the implementation. In the last two decades, researchers created many methodologies for tracking disjointness that enabled tackling the frame problem in heap-manipulating programs. However, the developed methodologies require their users to choose between expressive power and goals related to ease-of-use (low entry bar and ability to focus). Importantly, there is no methodology that enables its users to start a verification project with an easy-to-use technique and switch to a more powerful one for the parts of the project that require it. In this thesis, we aim to create such a methodology exploiting the special features of Rust, a new systems programming language. Compared to other mainstream programming languages, Rust makes two important contributions. First, it divides code into safe and unsafe subsets, with the safe subset being more commonly used. Second, for the safe subset, the Rust compiler automatically provides strong disjointness guarantees. In this thesis, we

explore how these Rust features can help us to make verification more approachable.

The remainder of this chapter is structured as follows. In the following section, we discuss the state of the art in deductive verification of concurrent heap-manipulating programs and how well it satisfies the four requirements mentioned in the first paragraph: expressivity, automation, low entry bar, and ability to focus. We discuss only modular techniques because they can be scaled to realistic systems. In Section 1.2, we describe our key observation that enables us to bridge expressive and easy-to-use methodologies. Section 1.3 gives an overview of the thesis and its contributions. Section 1.4 lists the publications in which parts of this thesis were published, and Section 1.5 lists the contributions that were developed collaboratively.

1.1 State of the Art

In 1967, Robert W. Floyd published a method for adding assertions to flowcharts that enabled formally proving that a program satisfies its specification [6]. Based on Floyd's ideas, Tony Hoare developed a *program logic* for proving that a code fragment adheres to its contract [7]. The key principles of the Hoare's approach we still use for verifying software today. Hoare replaced flowcharts with a logical system that we currently call *Hoare triples*. A Hoare triple is an assertion of the form $\{P\}\, C\, \{Q\}$ where P is an assertion expressing the precondition, Q is an assertion expressing the postcondition, and C is code. The Hoare triple expresses that if C is executed in a state that satisfies P and terminates, the end state satisfies Q. Logics that use Hoare triples are nowadays called Hoare logics. The rules of the original logic could be used to prove by hand the correctness of simple examples like the following `max` function:

[6]: Floyd (1967), 'Assigning meanings to programs'

[7]: Hoare (1969), 'An Axiomatic Basis for Computer Programming'

```
1  public static int max(int x, int y)
2  //@ requires true
3  //@ ensures x <= result && y <= result;
4  //@ ensures (x == result || y == result);
5  {
6      return (x <= y ? y : x);
7  }
```

The precondition `requires true` expresses that the method will always succeed. The postcondition written in `ensures` clauses expresses that if the method terminates[2], the result will not be smaller than any of the inputs and be equal to one of the inputs.

2: In this thesis, we focus on partial correctness.

In the last two decades, there have been many advancements in automatic theorem provers such as *SMT (satisfiability modulo theory)* solvers. Modern SMT solvers (for example, Z3 [8] and CVC5 [9]) can automatically prove complex first-order logic formulas. This power enables building deductive verifiers based on (a variant of) the Hoare logic such as VeriFast [2] that can automatically prove the contract of the `max` function without any additional help from the user. However, to scale verification beyond such simple examples, we need techniques that address the challenges mentioned earlier.

[8]: Moura et al. (2008), 'Z3: An Efficient SMT Solver'

[9]: Barbosa et al. (2022), 'CVC5: A Versatile and Industrial-Strength SMT Solver'

[2]: Jacobs et al. (2011), 'VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java'

In the remainder of this section, we will review the techniques for approaching the frame problem in concurrent heap-manipulating programs and the tools that implement these techniques. Presenting all research on deductive verification of concurrent heap-manipulating programs would take a book. Therefore, in this section, we focus only on the most relevant results for this thesis's goal: enabling building tools that programmers could use to verify their code.

1.1.1 Ownership Types

Probably the earliest approach for taming aliasing in verification is Universe Types [10], a verification approach based on an ownership type system [11]. Ownership type systems impose a hierarchical structure on the object graph. For example, in the Universe Type system, all nodes of a linked list would have the linked list class instance as an owner and each other as peers. In Universe Types, the ownership is tracked by the type system. Later, Dynamic Ownership [12] switched to tracking ownership in ghost state[3], which made it slightly more expressive. Ownership combined with an encapsulation discipline such as owner-as-modifier [13, 14] that requires that an object can be modified only via its owner enables efficient reasoning about framing [15] because object graphs owned by different owners are guaranteed to be disjoint. This property enabled building the SMT-based verifier Spec# [16] for a superset of C#. For instance, in a version of our example encoded in Spec#, a precondition `x != y` implies not only that references `x` and `y` are different but also that they point to disjoint object graphs.

[10]: Müller (2002), *Modular Specification and Verification of Object-Oriented Programs*

[11]: Clarke et al. (1998), 'Ownership Types for Flexible Alias Protection'

[12]: Leino et al. (2004), 'Object Invariants in Dynamic Contexts'

3: *Ghost* code is code that is used only for verification and has no effect on real execution.

[13]: Dietl et al. (2005), 'Universes: Lightweight Ownership for JML'

[14]: Dietl (2009), 'Universe Types - Topology, Encapsulation, Genericity, and Tools'

[15]: Müller et al. (2003), 'Modular specification of frame properties in JML'

[16]: Barnett et al. (2011), 'Specification and verification: the Spec# experience'

```
1  public static void ResetAndTest(Counter! x, Counter! y)
2      requires x != y;
3      modifies y.*;
4  {
5      int currentValue = x.GetValue();
6      y.ResetValue();
7
8      // Guaranteed to succeed.
9      assert(currentValue == x.GetValue());
10 }
```

An important advantage of Spec# was that its specification language was boolean expressions with minimal extensions such as quantifiers, lowering the learning curve for new users. However, simpler verification alone was not a strong enough argument for programmers to adopt programming patterns that adhere to the ownership discipline. Primarily because Universe Types and Spec# supported only static ownership. For example, if we wanted to verify a memory allocator, we would need to reason about allocation that transfers ownership of a memory block from the allocator to the client and deallocation that returns the ownership. Pressure to support code that does not adhere to the static ownership discipline led to the development of more flexible techniques at the cost of simplicity. For example, ownership transfer [12] required ensuring a complex condition that the original owner has no references left to the transferred group of objects.

[12]: Leino et al. (2004), 'Object Invariants in Dynamic Contexts'

Another advantage of Spec# and ownership types in general is that they naturally accommodate reasoning based on object invariants. Aliasing makes it hard to soundly reason about object invariants because the object on which an invariant depends could be modified from a context that is not aware of the existence of the invariant. This problem can be addressed by requiring that invariants depend only on owned objects and that all modifications happen through the owner (for example, by using owner-as-modifier discipline). A slightly more flexible version of this approach was implemented in Spec#. A different design was used by VCC [17], a verifier for concurrent C, that used an invariant methodology powerful enough to build ownership reasoning on top. The key contribution of the VCC approach was the admissibility check that ensured that if some invariant I depends on object o with invariant I_o, then any operation preserving I_o must also preserve I. This design, for example, enabled VCC invariants to depend on volatile memory that could be modified by multiple threads. Ownership in VCC was encoded by adding a ghost `owner` field to each object, which means that the ownership could be specified in specifications and changed during execution. These properties enabled VCC to support complex patterns, such as a spin lock that owns some resource (for example, a memory block) while it is locked and transfers that resource to the thread that unlocks it. However, this expressive power came at a cost: the specification language used by VCC is significantly more complicated than the one used by Spec#; it has more constructs that are specific to the VCC methodology and that require expert knowledge to use correctly. From this perspective, VCC is arguably comparable to permission logics discussed in the following section.

[17]: Cohen et al. (2009), 'VCC: A Practical System for Verifying Concurrent C'

1.1.2 Permission Logics

Slightly after ownership types, approaches based on program logics that track ownership in some way started to appear. In this subsection, we cover the three most important ones that were used for building automated verifiers: separation logic [18] and implicit dynamic frames [19].

[18]: O'Hearn et al. (2001), 'Local Reasoning about Programs that Alter Data Structures'

[19]: Smans et al. (2009), 'Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic'

Separation logic [18, 20] is a Hoare logic whose assertions track not only functional properties but also ownership of resources, such as memory. The key property of the separation logic is that if Hoare triple $\{P\}\ C\ \{Q\}$ holds, then code C for its safe execution relies only on resources that are owned by P. As a result, C is guaranteed not to touch any resources not owned by P and, therefore, knowledge about them can be framed. The key features that separation logic introduces are points-to assertion $l \mapsto v$ and separating conjunction $A * B$. A *points-to assertion* $l \mapsto v$ expresses that the current activation record owns the location with address l and that the location has value v. The proof rules of separation logic ensure that only the owner of a location has permission to access it. The *separating conjunction* $A * B$ expresses not only that both A and B hold (like the regular conjunction $\wedge$) but also that the locations owned by A and B are disjoint. The property that knowledge about locations not used for verifying C can be preserved is formally captured by the *frame rule*. The frame rule expresses that if triple $\{P\}\ C\ \{Q\}$ holds, then the triple $\{P * R\}\ C\ \{Q * R\}$ where the state is extended with frame R is also guaranteed to hold. Our example in VeriFast [2], a verifier for

[18]: O'Hearn et al. (2001), 'Local Reasoning about Programs that Alter Data Structures'

[20]: O'Hearn (2007), 'Resources, concurrency, and local reasoning'

[2]: Jacobs et al. (2011), 'VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java'

Java based on separation logic, would look as follows (`&*&` is VeriFast syntax for the separating conjunction, `?variable` is an existentially bound variable, `_` is a wildcard indicating that we do not care about the value, `Counter` is a predicate that abstracts over an assertion).

```
1  public static void resetAndTest(Counter x, Counter y)
2  //@ requires Counter(x, ?value_x) &*& Counter(y, ?value_y)
3  //@ ensures Counter(x, _) &*& Counter(y, _)
4  {
5      int currentValue = x.getValue();
6      y.resetValue();
7      assert(currentValue == x.getValue());
8  }
```

Separation logic provides three key advantages compared to Universe Types and Spec#. First, since a separation logic assertion precisely captures ownership, ownership transfer becomes straightforward: even if the original owner still has a pointer to a memory location, they are not allowed to use it. Second, separation logic supports framing of non-hierarchical structures such as double-linked lists and arrays, which ownership types cannot because the elements of these containers typically have the same owner. Framing in separation logic can be achieved by using, for example, an *iterated separating conjunction* [21] to express that a set of assertions are mutually separated. Third, in addition to the separating conjunction, which is a dual of conjunction, separation logic also introduced a dual of implication called *separating implication* $A \mathrel{-\!\!*} B$, better known as *magic wand*. A magic wand $A \mathrel{-\!\!*} B$ expresses that by giving up both A and the wand, we can obtain B. Magic wands proved to be instrumental for verifying complicated iterative code; for example, [22] showed how magic wands can be used to verify iterators.

[21]: Reynolds (2002), 'Separation Logic: A Logic for Shared Mutable Data Structures'

[22]: Krishnaswami (2006), 'Reasoning about iterators with separation logic'

After the first work on separation logic was published, researchers developed many extensions, especially to support verifying concurrent programs. One key extension was fractional [23] and counting [24] permissions that enable multiple concurrent readers to access the same memory location safely. With *fractional permissions*, the points-to assertion $l \mapsto v$ is generalized to a fractional variant $l \overset{p}{\mapsto} v$ where p is a positive value smaller-equal to 1.0. $p = 1.0$ is full permission and allows mutating the memory location. Any non-zero permission allows reading the memory location. The points-to assertions can be split and recombined, maintaining the invariant that the sum of all permissions to the same location is equal to 1.0. This invariant ensures that each location can be either read by multiple readers or written by a single writer, guaranteeing the absence of data races. Fractional permissions naturally fit into modelling structured parallelism where each thread needs to be given read access. *Counting permissions* also allow multiple readers or a single writer. However, instead of allowing to split permission amounts, they allow taking an unbounded number of read permissions from a source permission. If no permissions are taken away from the source, the source can be used for write access. Counting permissions is a natural fit for modelling data guarded by read-write locks. Later work showed how fractional permissions could be used without specifying concrete fractions to verify a common pattern of providing read permission to a recursive function [25] and how counting permissions could be encoded as fractional ones [26].

[23]: Boyland (2003), 'Checking Interference with Fractional Permissions'

[24]: Bornat et al. (2005), 'Permission accounting in separation logic'

[25]: Heule et al. (2013), 'Abstract Read Permissions: Fractional Permissions without the Fractions'

[26]: Boyland et al. (2014), 'Constraint Semantics for Abstract Read Permissions'

Building automated verifiers for separation logic and its variants is an ongoing effort that has produced many tools such as Smallfoot [27], jStar [28], VeriFast [2], and GRASShopper [29], which provide varying degrees of automation. These tools focus on supporting the core features of separation logic, such as points-to assertions and separating conjunction. However, recent developments show how also the more advanced features such as iterated separated conjunction [30] and magic wand [31–33] and even checking a complex separation logic proof [34] can be automated in an SMT-based verifier.

[27]: Berdine et al. (2005), 'Smallfoot: Modular Automatic Assertion Checking with Separation Logic'

[28]: Distefano et al. (2008), 'jStar: towards practical verification for java'

[2]: Jacobs et al. (2011), 'VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java'

[29]: Piskac et al. (2014), 'GRASShopper - Complete Heap Verification with Mixed Specifications'

[30]: Müller et al. (2016), 'Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution'

[31]: Schwerhoff et al. (2015), 'Lightweight Support for Magic Wands in an Automatic Verifier'

[32]: Blom et al. (2015), 'Witnessing the elimination of magic wands'

[33]: Dardinier et al. (2022), 'Sound Automation of Magic Wands'

[34]: Wolf et al. (2021), 'Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA'

Separation logic and its extensions provide the expressive power to verify concurrent heap-manipulating programs. However, similarly to VCC, this power comes with a cost: while the specification language used by Spec# was based on boolean expressions with first-order logic extensions and could be understood by regular programmers, understanding and writing separation logic specifications requires much deeper expertise. Moreover, before the programmer can focus on proving the properties they care about, they need to specify how the memory ownership is passed through the program, which is a substantial upfront investment (even compared to writing the additional ownership annotations required by advanced type systems). For example, while predicates are a conceptually simple abstraction mechanism (they are just named separation logic assertions), designing them requires expertise. Also, in automatic tools using predicates cause a significant annotation overhead because they have to be declared and manually managed by the user. Some researchers tried to alleviate the upfront investment by designing approaches that effectively infer specifications for common patterns. For example, [35] used type inference to infer internal specifications of tree data structures given top-level specifications of types and methods. However, such approaches still require the user to master separation logic.

[35]: Bakst et al. (2016), 'Predicate Abstraction for Linked Data Structures'

Implicit dynamic frames [19] is a Hoare logic closely related to the separation logic [36]. Instead of the points-to assertion $l \mapsto v$ available in separation logic, implicit dynamic frames provides an accessibility predicate $\mathsf{acc}(l)$ that expresses access permission to location l without telling anything about its value. The values stored at locations are constrained by using heap-dependent expressions. For example, separation logic assertion $r.f \mapsto 0$ corresponds to $\mathsf{acc}(r.f) \wedge r.f = 0$ in implicit dynamic frames. This example shows that implicit dynamic frames provides a natural way of separating ownership (what memory is owned) and functional (what is the value stored in owned memory) concerns. A common strategy in separation logic to achieve similar separation between ownership and functional concerns is to use *snapshots* [21]: mathematical values that fully capture the value stored in a group of heap locations (for example, a linked list). However, to use snapshots, the user needs to define them and relate them to the values stored in the heap, which requires additional effort. Besides cleaner separation of ownership and functional specifications, implicit dynamic frames shares strengths and weaknesses with separation logic: it is expressive (and has constructs that correspond not only to core separation logic such as magic wand and iterated separating conjunction, but also many of its extensions such as fractional and counting permissions) but requires expertise and upfront effort.

[19]: Smans et al. (2009), 'Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic'

[36]: Parkinson et al. (2012), 'The Relationship Between Separation Logic and Implicit Dynamic Frames'

[21]: Reynolds (2002), 'Separation Logic: A Logic for Shared Mutable Data Structures'

The paper that introduced implicit dynamic frames also presented

an automated verifier VeriCool [19]. Later, Chalice [37] was used to demonstrate supporting concurrency features such as channels and locks in a verifier based on implicit dynamic frames. Viper verification infrastructure [38] that is based on implicit dynamic frames was used to build Nagini [39] and Gobra [40], verifiers that aim to support verification of existing Python and Go code, respectively.

In separation logic and implicit dynamic frames, the part of the heap that could be accessed by an operation, its *footprint*, is specified implicitly. In the line of work on *dynamic frames* [41] and *regional logic* [42], the footprint is an explicit set of locations. This set is typically specified by using set-type specification variables. For each operation and method, a user needs to specify a set of locations it may read and a set of locations it may modify. The great flexibility of this approach comes from the fact that the operation may modify the access sets and that the sets manipulated by different operations could overlap if needed. On the flip side, since the sets are not automatically ensured to be disjoint, a lot of specification effort is spent on ensuring this property. As a result, writing specifications requires comparable expertise and effort to the ones in separation logic and implicit dynamic frames. Also, since footprints are tracked by using specification variables, it is unclear how to extend this approach to concurrency because that would require preventing races on the specification variables themselves. SMT-based verifiers that use the dynamic frames methodology are Dafny [43] and the KeY [44] verifier for Java.

[19]: Smans et al. (2009), 'Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic'

[37]: Leino et al. (2009), 'A Basis for Verifying Multi-threaded Programs'

[38]: Müller et al. (2016), 'Viper: A Verification Infrastructure for Permission-Based Reasoning'

[39]: Eilers et al. (2018), 'Nagini: A Static Verifier for Python'

[40]: Wolf et al. (2021), 'Gobra: Modular Specification and Verification of Go Programs'

[41]: Kassios (2006), 'Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions'

[42]: Banerjee et al. (2008), 'Regional Logic for Local Reasoning about Global Invariants'

[43]: Leino (2010), 'Dafny: An Automatic Program Verifier for Functional Correctness'

[44]: Ahrendt et al. (2014), 'The KeY Platform for Verification and Analysis of Java Programs'

1.1.3 Reference Capabilities

All approaches based on permission logics mentioned in the previous subsection suffer from the same two problems: using them requires expert knowledge and a sizeable upfront investment. SYMPLAR [45] attempted to mitigate both of these issues by going back to the verification style that is similar to the one used with Universe Types [10] that had a clear separation between alias tracking and functional specifications. In SYMPLAR, alias control was done entirely in the type system, allowing a simple specification language for specifying functional properties. The alias control mechanism of SYMPLAR was inspired by separation logic: it used *reference capabilities* [46] to represent permissions to access objects. For example, annotating a reference with `@Excl` indicated that the reference has a *uniqueness* [47] capability that expresses that the object could be accessed only through that reference. Similarly, `@Imm` indicated that the reference allows read-only access to the object. The following snippet shows our example with SYMPLAR annotations.

[45]: Bierhoff (2011), 'Automated program verification made SYMPLAR: symbolic permissions for lightweight automated reasoning'

[10]: Müller (2002), *Modular Specification and Verification of Object-Oriented Programs*

[46]: Boyland et al. (2001), 'Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only'

[47]: Hogg (1991), 'Islands: Aliasing Protection in Object-Oriented Languages'

```
1  public static void resetAndTest(
2      @Excl Counter x,
3      @Excl Counter y
4  ) {
5      int currentValue = x.getValue();
6      y.resetValue();
7      assert(currentValue == x.getValue());
8  }
```

Uniqueness is a much stricter property than ownership. For example, in an ownership type system, a linked list node object could be pointed at not only by its owner (typically, an instance of the linked list class) but also by its peer nodes. In a uniqueness type system, a node could be pointed at only by a single reference, typically the field of the linked list object pointing to the head node or a field of a preceding node pointing to the next node. Due to this single reference requirement, ownership transfer is trivial (we only need to ensure that the original owner does not use their reference anymore, for example, by setting it to `null`). However, it becomes impossible to implement cyclic data structures such as doubly-linked lists. These restrictions made some researchers [12] believe that uniqueness is too restrictive to be usable in practice, some even calling them "draconian" [27]. SYMPLAR tried to improve the expressivity of their type system based alias control by providing flexible borrowing constructs that allowed the creation of temporary aliases to unique references within a lexical scope. This flexible borrowing enabled the paper's authors to verify the implementation of an iterator over an array list – an example that none of the prior SMT-based verifiers could handle. However, the SYMPLAR approach still has three significant limitations. First, it does not support important patterns such as doubly-linked lists. Second, since it relies on the type system, it requires the entire code base to be written in the same programming language, which does not always hold for modern systems. Third, while SYMPLAR's capability type system supports complex borrowing patterns, SYMPLAR does not provide a modular technique for verifying functional correctness of code that uses these patterns[4].

[12]: Leino et al. (2004), 'Object Invariants in Dynamic Contexts'

[27]: Berdine et al. (2005), 'Smallfoot: Modular Automatic Assertion Checking with Separation Logic'

4: We provide an in depth comparison with our work in Section 7.2.

1.1.4 Avoiding Aliasing

Yet another approach for avoiding the difficulties of the frame problem caused by the heap and unrestricted aliasing is to forbid aliasing completely. There are two main ways of achieving the absence of aliasing: using a pure functional programming language like Haskell where programs always manipulate immutable values or use a subset of a language without pointers and references. The former was employed by, for example, LiquidHaskell [48], a verifier based on the Liquid Types [49] approach. The latter was employed by, for example, SPARK [50], an industrial strength verifier for Ada. While completely forbidding pointers and the heap is limiting, in some domains, it may be needed for reasons other than simpler verification. For example, most of the verifiers mentioned above assume that the heap is infinite and memory allocation never fails because in desktop and server applications, such events are rare, and a crash caused by an out-of-memory exception is unlikely to cause serious issues. However, crashes must be prevented at all costs in safety-critical domains such as avionics, making it easier to use other memory management methods than a global heap.

[48]: Vazou et al. (2014), 'LiquidHaskell: experience with refinement types in the real world'

[49]: Rondon et al. (2008), 'Liquid types'

[50]: Carré et al. (1990), 'SPARK—an Annotated Ada Subset for Safety-critical Programming'

1.1.5 Automating Verification

Almost all verifiers we mentioned are based on off-the-shelf SMT solvers such as Z3 [8] due to the great automation provided by these automatic provers, which made it much easier to develop verifiers that provide

[8]: Moura et al. (2008), 'Z3: An Efficient SMT Solver'

high degrees of automation. Another essential step that enabled building verifiers for real programming languages was the creation of verification infrastructures based on *intermediate verification languages* (IVLs). Many modern verifiers were built as translators from the source language into some IVL. For example, Gillian-JS and Gillian-C [51] use Gillian [52], which provides a generic separation logic framework that can be instantiated with different memory models; Vercors [53], Gobra, and Nagini use Viper [38], which provides built-in reasoning about heap based on implicit dynamic frames; VCC, Spec#, Chalice, Dafny, and even Viper use Boogie [54], which provides reasoning about imperative, potentially unstructured programs without heap; and SPARK, Cameleer [55], and Frama-C [56] use Why3 [57], which, unlike Viper and Boogie, supports many different solvers, including proof assistants (such as Coq [58] and Isabelle/HOL [59]) that can be used for manually proving examples that SMT solvers cannot handle.

[51]: Maksimovic et al. (2021), 'Gillian, Part II: Real-World Verification for JavaScript and C'

[52]: Santos et al. (2020), 'Gillian, part I: a multi-language platform for symbolic execution'

[53]: Amighi et al. (2012), 'The VerCors project: setting up basecamp'

[38]: Müller et al. (2016), 'Viper: A Verification Infrastructure for Permission-Based Reasoning'

[54]: Barnett et al. (2005), 'Boogie: A Modular Reusable Verifier for Object-Oriented Programs'

[55]: Pereira et al. (2021), 'Cameleer: A Deductive Verification Tool for OCaml'

[56]: Kirchner et al. (2015), 'Frama-C: A software analysis perspective'

[57]: Filliâtre et al. (2013), 'Why3 - Where Programs Meet Provers'

[58]: Barras et al. (1997), *The Coq proof assistant reference manual: Version 6.1*

[59]: Nipkow et al. (2002), *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*

1.1.6 Discussion

If we compare the state of the art with the five requirements we listed at the beginning, the presented approaches are automated using SMT-solvers and modular, which enables to scale them to realistic systems. However, with respect to the over three requirements, we could group the presented approaches into two conflicting camps. In one camp, we have approaches such as Universe Types, Spec#, SYMPLAR, and SPARK. These approaches restrict aliasing, which enables having simpler specifications, lowering entry bar, and allowing users to focus sooner on the properties that are relevant to them. However, these benefits come at the cost of expressivity. For example, some of them do not support at all or poorly support cyclic data structures such as doubly-linked lists. In the other camp, we have approaches such as VCC and permission logics that are expressive but require expert knowledge and a significant upfront investment before a user can focus on the properties they are interested in. To conclude, we still need an approach that would have a low entry bar and allow users to focus on the properties they care about while being sufficiently expressive.

1.2 Rust Opportunity

As we saw in the previous section, with existing approaches, users must choose between ease-of-use and expressive power. Our key insight is that instead of trying to close the gap between these two camps, we can enable users to seamlessly travel between them: let the users start with an easy-to-use technique and allow them to switch to a more powerful one for the parts of the system that require it. Our crucial observation that enables this seamless transition is that the main complexity and overhead of permission logics come from the need to specify and prove a core set of properties. This core set of properties, which we call the *core proof*, specify the memory structure of the program. Importantly, these properties are the same ones as proven by ownership or reference capability type systems (as witnessed by the fact that separation logic is sometimes used to prove soundness of such type systems [60, 61]). Therefore, if we

[60]: Gordon et al. (2012), 'Uniqueness and reference immutability for safe parallelism'

[61]: Balabonski et al. (2016), 'The Design and Formalization of Mezzo, a Permission-Based Programming Language'

managed to construct the core proof from the information available in a capability type system, we would have a lightweight specification and verification approach with a clear path for supporting examples where the type system is not expressive enough.

While ownership and uniqueness types were widely explored by the research community, who created both new programming languages [61–63] and extensions to existing programming languages [60, 64–66], none of the attempts were taken seriously by the wider programmer community. The situation changed in 2015 when Mozilla released Rust: a new safe systems programming language that is designed as a safe alternative to C and C++ and is quickly gaining widespread adoption. Unlike prior memory-safe systems programming languages, Rust for memory safety relies not on the garbage collector but on its type system. Rust's type system was inspired by works on uniqueness and region-based memory management [60, 64, 67–69] and features uniqueness types with borrowing[5]. Compared to prior work such as Cyclone and SYMPLAR, Rust brings two essential innovations that arguably contributed to its take-off. First, for tracking borrows, Rust introduced *lifetimes*, which are an improved version of Cyclone regions. Lifetimes are not only more flexible than prior mechanisms[6], but also more ergonomic because type inference can infer them automatically in most cases. The following code snippet shows our example in Rust (`&mut` indicates that the type is a unique reference to an object).

```
fn reset_and_test(x: &mut Counter, y: &mut Counter)
{
    let current_value = x.get_value();
    y.reset_value();
    assert!(current_value == x.get_value());
}
```

Second, to compensate for restrictions of the type system, Rust provides an escape hatch called *unsafe code*[7]: code marked with an `unsafe` keyword gets access to more powerful language features such as raw pointers and manual memory management [70]. By using unsafe code, programmers can extend the language with new *safe abstractions* that enable new programming patterns [71]; for example, vector `Vec` and unique owning pointer `Box` are two safe abstractions from the Rust standard library implemented by internally using unsafe code. While unsafe code gives additional expressive power, it comes at a significant cost: the responsibility of ensuring the memory safety of the code that uses unsafe is shifted from the compiler to the programmer. The development of safe abstractions is additionally complicated by the fact that Rust code that does not use unsafe must not be able to cause memory safety errors [72], which means that the safe abstraction has to guarantee memory safety even in cases when it is misused.

Rust becoming mainstream gives us a unique opportunity to try achieving our goal: developing a verification approach that enables building a verifier usable by programmers. We pursue this goal by making the verification incremental in two dimensions. First, the developer should be able to focus on proving the properties they care about without large prior investment. Second, the effort required for verifying common code

[61]: Balabonski et al. (2016), 'The Design and Formalization of Mezzo, a Permission-Based Programming Language'
[62]: Clebsch et al. (2015), 'Deny capabilities for safe, fast actors'
[63]: Stork et al. (2014), 'Æminium: a permission based concurrent-by-default programming language approach'
[60]: Gordon et al. (2012), 'Uniqueness and reference immutability for safe parallelism'
[64]: Swamy et al. (2006), 'Safe manual memory management in Cyclone'
[65]: Fähndrich et al. (2006), 'Language support for fast and reliable message-based communication in singularity OS'
[66]: Haller et al. (2010), 'Capabilities for Uniqueness and Borrowing'
[60]: Gordon et al. (2012), 'Uniqueness and reference immutability for safe parallelism'
[64]: Swamy et al. (2006), 'Safe manual memory management in Cyclone'
[67]: Clarke et al. (2003), 'External Uniqueness Is Unique Enough'
[68]: Tofte et al. (1997), 'Region-based Memory Management'
[69]: Grossman et al. (2002), 'Region-Based Memory Management in Cyclone'

5: Rust also supports so-called internally mutable types such as `Mutex` that allow relaxing some of the uniqueness requirements. We leave handling such types to future work.

6: We will compare the borrowing mechanisms in detail in Section 7.5.

7: Many other programming languages also have escape hatches, for example, `sun.misc.Unsafe` in Java. However, since Rust has a more restrictive type system and targets systems code, `unsafe` in Rust is more important.

[70]: Developers (2023), *The Rustonomicon: What Unsafe Rust Can Do*
[71]: Jung et al. (2018), 'RustBelt: securing the foundations of the Rust programming language'
[72]: Developers (2023), *The Rustonomicon: How Safe and Unsafe Interact*

should be as low as for type system based techniques while allowing a smooth transition to more powerful methods when needed.

1.3 Overview and Contributions

In this thesis, we present a novel verification approach for imperative heap-manipulating programs that exploits the special features of Rust to make significant progress towards the five goals we mentioned at the beginning. Our approach is modular, which enables it to scale, and automated with SMT. We target the remaining three goals by making our approach incremental on two dimensions. Our first dimension of enabling the developer to focus on proving the properties they care about without large prior investment corresponds to the goals of ensuring a low entry bar and ability to focus. Unfortunately, as we saw from the state of the art discussion, these two goals are in conflict with expressivity. Therefore, instead of trying to create an approach that is both easy to use and expressive enough to handle all cases (which is impossible), with our second dimension, we enable the user to use more powerful concepts for the parts of the project that require them. The thesis is divided into three parts, which we discuss below.

In Part I, we focus on the first dimension showing how we enable the low entry bar and the ability to focus when verifying safe Rust. In this part, we show how to address three key challenges. First, we demonstrate how information available in the Rust type system can be used to generate a core proof in implicit dynamic frames completely automatically. In particular, we show how to support non-lexical borrowing patterns that were introduced in Rust 2018 edition [73], which are significantly more flexible than the ones used in SYMPLAR and Rust 1.0. Second, we show how specifications written as Rust expressions with first-order extensions can be automatically incorporated into the generated core proof to obtain complete specification of a Rust program. Third, we address an important shortcoming of prior work: SYMPLAR presented a type system that supports returning borrows but did not have a way of describing their effects that respects information hiding. We present *pledges*, a novel construct that fills this gap. We finish the part by presenting Prusti, a Viper-based verifier that we used to evaluate our approach, and comparing our model of Rust borrows with later proposals from other researchers.

[73]: Developers (2018), *Announcing Rust 1.31 and Rust 2018*

In Part II, we present our empirical study on how unsafe code is used in practice, which is a preparation step before verifying unsafe code in Part III. In this study, we investigate an assumption, which we dub Rust hypothesis, that Rust programmers use unsafe code following three key principles: use unsafe code sparingly, make it easy to review, and hide it behind a safe abstraction such that client code can be written in safe Rust. We analyse a large corpus of Rust projects to determine whether the Rust hypothesis holds and to classify the purpose of unsafe code. In particular, in the later parts, we aim to verify the uses of the two most common unsafe features: calls to unsafe functions and dereferences of raw pointers.

In Part III, we focus on the verification of unsafe code and safe abstractions. The two key challenges we address are related to the two dimensions. First, we need to ensure that unsafe code does not violate the type system properties on which safe code relies and which we exploit for the lightweight verification described in Part I. We present a methodology for verifying that safe abstractions guarantee memory safety even when used by unverified safe clients. Second, we show how the verification approach presented in Part I can be extended to the verification of functions containing safe and unsafe code in a way that verification overhead compared to verifying completely safe code is determined by the complexity[8] of unsafe code. The model of borrows presented in Part I is too simplistic to verify unsafe code. Therefore, we started designing a new model suitable for SMT-based verification of safe and unsafe code; we present our partial results in Chapter 21. We evaluate our approach by extending Prusti.

8: The number of unsafe operations is not a good predictor of complexity: for example, a function that uses many unsafe indexing into array operations that omit bounds checks is likely to require no additional specifications while a single pointer dereference may incur large specification overhead.

1.4 Publications

This dissertation contains text and material from the following publications:

- Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 'Leveraging Rust types for modular specification and verification'. In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019) [74]

  This paper was used as the basis for Part I. This part uses the same structure as the paper, but the text is in most cases new with the main exception being Chapter 6 (Implementation and Evaluation) that contains mostly paraphrased text. The figures used in Chapter 6 (Implementation and Evaluation) are taken from the paper.

  The PCS elaboration algorithm described in Subsection 2.2.2 was not described in the main publication. Its description is based on the extended version of the paper [75]. The loan-dependency graph described in Section 3.3 was not presented in the paper; we describe it here for the first time.

[75]: Astrauskas et al. (2019), *Leveraging Rust types for modular specification and verification (extended version)*

- Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 'How do programmers use unsafe Rust?' In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020)

  Text in Part II is taken from this publication with minor adjustments: Chapter 9 (Motivations for Using Unsafe Code) was paraphrased and Chapter 15 (Related Work) was expanded to include related work published after our publication.

1.5 Collaborations

The work presented in Part I and Part II was done in collaboration with fellow doctoral students Federico Poli and Aurel Bílý, postdoctoral researcher Christoph Matheja, and the students we supervised. The work presented in Part III was done entirely by me and the students I[9] supervised. Federico and I contributed equally to most aspects of

9: To clearly distinguish between my contributions and those of others, I use the first person singular to refer to my contributions in this section.

the work presented in Part I. We together designed the specification language, pledges, and the encoding of Rust types in Viper. We also worked together on the implementation. Federico led the work on the PCS elaboration algorithm, while I led the work on modelling mutable and shared references. Federico did the large-scale core proof and overflow freedom studies presented in Subsection 6.2.1 and Subsection 6.2.2, respectively. I evaluated our approach by verifying the properties beyond memory safety as presented in Subsection 6.2.3. The snapshots presented in Section 5.2 were designed by Christoph and me, and implemented over multiple iterations by Christoph, Aurel, and my student Till Arnold as part of his bachelor thesis [77]. The students we supervised typically contributed a specific feature. Therefore, I discuss a student's contribution when discussing the feature they contributed to.

[77]: Arnold (2020), 'Optimization of Viper-Based Verifier'

The work presented in Part II was led by me. The initial version of the Rust compiler plugin for collecting data was written by Nicolas Winkler as part of this bachelor thesis project [78] supervised by me. It also received some fixes and improvements from independent contributor Konstantinos Triantafyllou. Based on the learnings in this initial project, I designed and implemented the Qrates framework, which I used to collect the data. Then, Federico and I analysed the collected data together. Notably, Federico contributed the classification of motivations for using unsafe code discussed in Chapter 9 and the analysis that computes why an unsafe block is needed.

[78]: Winkler (2018), 'Semantic Querying of Rust Code'

Part I

Verifying Functional Correctness of Programs Written in Safe Rust

In the first part of the thesis, based on [74], we focus on the first of the two dimensions mentioned in the introduction. We show how we can leverage the Rust type system to enable incremental verification where users can immediately focus on the properties they care about and verify additional properties only when they see a need. In the introduction, we discussed that one of the key challenges in verifying heap-manipulating programs is the frame problem and claimed that Rust type system helps address it. Before diving into the technical aspects of our approach, we need to get an intuition on how Rust simplifies reasoning about program correctness. Therefore, we revisit the motivating example from the introduction.

[74]: Astrauskas et al. (2019), 'Leveraging Rust types for modular specification and verification'

```
1  pub struct Counter {
2      inner_value: i32,
3  }
4  impl Counter {
5      #[pure]
6      pub fn get_value(&self) -> i32 {
7          self.inner_value
8          // equivalent to: (*self).inner_value
9      }
10     #[ensures(self.get_value() == 0)]
11     pub fn reset_value(&mut self) {
12         self.inner_value = 0;
13     }
14 }
15 #[ensures(x.get_value() == old(x.get_value()))]
16 #[ensures(0 == y.get_value())]
17 fn reset_and_test(x: &mut Counter, y: &mut Counter)
18 {
19     let x_current_value = x.get_value();
20      // equivalent to let ... = Counter::get_value(&*x);
21     y.reset_value();
22     // equivalent to Counter::reset_value(&mut *y);
23     assert!(x_current_value == x.get_value());
24     assert!(0 == y.get_value());
25 }
```

Figure 1.1: An expanded version of the reset counter example from the introduction. For this expanded version, we want to prove not only that `x` did not change (assert on line 23) but also that `y` was reset to 0 (assert on line 24). Note that the compiler often automatically inserts operations for creating and dereferencing a reference. Therefore, for example, the lines 7 and 19 are equivalent to the ones shown in the comments below them.

Example. Figure 1.1 shows an elaborated version of the reset counter example from the introduction with an additional assert. In addition to showing that the value referenced by `x` is preserved (the call to `assert!` macro on line 23), we want to show that the value referenced by `y` is reset to zero (the assert on line 24). The example shows a declaration of `Counter` struct with a single private field and two public methods. Field `inner_value` is a 32-bit signed integer. Method `get_value` is a getter that returns the field's value. Similarly to Python, a reference to the struct itself in Rust is passed as a first argument. `&self` is a syntactic sugar for `self: &Counter`: a shared reference to a `Counter` instance. Shared references in Rust can be aliased but cannot be used for mutation. To make writing code more ergonomic, the Rust compiler often automatically inserts operations for dereferencing a reference. For example, on line 7, the compiler automatically inserts a dereference, as shown by the comment on the line below. Method `reset_value` resets the field's value to zero. `&mut self` is a syntactic sugar for `self: &mut Counter`: a unique (also called mutable)

reference to a `Counter` instance. Unique references in Rust can be used for mutation but cannot be aliased. The Rust type system guarantees that either a memory location can be accessed through multiple aliases or is mutable[10]. The annotations `#[pure]` and `#[ensures(...)]` are instructions for Prusti and we explain them below.

10: There are *internally mutable* types such as `Mutex` that can be mutated via shared references. We leave handling such types for future work.

[79]: Rust community (2023), *Rust: The Reference — Place Expressions and Value Expressions*

Since Rust has a capability type system, each *place* [79], an expression indicating a memory location (called L-value in other languages), potentially carries a capability for the memory location. The parameters `x` and `y` of function `inc_and_test` provide *unique ownership* capabilities to the referenced memory locations. As a result, we have a guarantee that `x` and `y` point to disjoint memory regions. Call `x.get_value()` automatically *reborrows* `x` as a shared reference for the duration of the call (the reborrowing is made explicit in the comment on line 20 that shows a desugared form of the method call). This action makes the reference `x` unusable for mutation while the reborrow is active. Similarly, call `y.reset_value()` automatically reborrows `y` for the duration of the call. Since `reset_value` takes a unique reference, `y` is completely unusable while the reborrow is active. We will cover borrowing and reborrowing in detail in Chapter 3.

Correctness Arguments. In the introduction, we mentioned five properties that need to hold to prove that the assert on line 23 never fails:

1. The references `x` and `y` are non-null.
2. Method `get_value` does not modify any state.
3. The call to `y.reset_value()` does not affect the value returned by `x.get_value()`.
4. No other thread can mutate `x` target concurrently.
5. `x.get_value()` result depends only on its arguments.

The first four properties (1, 2, 3, and 4) are guaranteed by the Rust type system while the last one (5) is specified by the user. Property 1 holds because the Rust compiler forbids accessing invalid references. Property 2 is guaranteed because `get_value` takes a shared reference that provides a capability to read, but does not allow mutating the memory. Property 3, which is the frame property discussed in the introduction, is guaranteed by the fact that `x` and `y` provide unique capabilities to their targets and, therefore, are disjoint. Unique capabilities also imply that no other thread has access to these memory locations, thus implying property 4. Property 5 is stated by the user by annotating function `get_value` with `#[pure]`, which instructs the verifier that the method is a mathematical function (deterministic, terminating, and side-effect free).

To prove that the assert on line 24 never fails, we additionally need to know the value returned by `y.get_value()` after the call to `y.reset_value()`. The user can specify this value by adding a postcondition to `reset_value` method (line 10). `get_value` can be used in the postcondition because it is a pure function. We require the user to write the annotations specifying the purity and the postcondition because we aim for modular verification in which we rely on function contracts and not their implementation (for the same reason, it is a good practice to use pure functions in the specification instead of fields). If we allowed the verifier to inline the called functions, these two annotations would not be necessary, and we could prove the assert statement by relying entirely on the type system. However, modularity is essential for scaling verification, being able to

verify libraries separately from clients, and localizing re-verification efforts when a part of a codebase is changed.

In the introduction, we used the assert macro that performs a runtime check to express the functional property because all languages have such runtime checks. However, even if we prove that the assert cannot fail, it will likely cause runtime overhead. Moreover, the clients of the function `reset_and_test` cannot use the proven properties. Therefore, the lines 15 and 16 show the postconditions that express the same properties as the two assert statements. The first postcondition expressing that the value referenced by `x` did not change uses an `old(...)` expression that evaluates the expression in the state at the beginning of the method. `old(...)` is one of the few (but powerful) additional constructs of our specification language. We could altogether avoid writing this postcondition if we changed the type of `x` to use a shared reference `&Counter` because the postcondition then would be implied by the capabilities.

Verification Approach Overview. Our example shows how the information about capabilities for memory locations available in the Rust type system can be combined with user-provided information about values of these memory locations to achieve a full correctness proof. Compared to prior work such as SYMPLAR, we contribute from both user experience and methodological points of view. Since we base our work on Rust that has a built-in capability type system, we can achieve real incremental verification where the minimal specification that the user has to provide before verifying properties that matter to them is none at all. Our approach enables the user to decide whether to verify the absence of overflows and other possible runtime errors or focus on proving some functional correctness aspect of their program. Importantly, the assertion language in which users express functional properties is at the level of Rust expressions with a few additional constructs, such as the aforementioned `old(...)` expression. One additional construct crucial for verifying Rust programs is a *pledge*, a novel construct that enables specifying functional behaviour of reborrowing; we present pledges in Chapter 5, Section 5.3.

Prior work that aimed at lightweight specification and verification, such as SYMPLAR [45], fully relied on the type checker to control aliasing, side effects, and framing. As discussed in the introduction, such design has two consequences: it is unclear how to support patterns that do not fit into the type system and how to link with verified code written in other programming languages. Therefore, in our work, we explored an alternative path of encoding capabilities and user-provided functional properties into implicit dynamic frames. This program logic is sufficiently expressive to capture both and reason about their interactions. Our technique consists of two steps. First, we encode the capability information and the semantics of Rust statements into a *core proof* that captures the information about aliasing and side effects. Second, we incorporate the specifications the user provides into the core proof. The key challenge is that these two steps and checking the core proof must be completely automatic; otherwise, we would fail to provide the abstraction we aim for. Achieving such a level of automation for program logics powerful enough to model Rust capabilities and non-lexical borrows is a highly complex task. It requires not only choosing a suitable encoding

[45]: Bierhoff (2011), 'Automated program verification made SYMPLAR: symbolic permissions for lightweight automated reasoning'

but also providing auxiliary annotations to automate the proof search, as we show in the chapters of this part. A crucial aspect of our work is that it lays the foundations for supporting unsafe Rust: for Rust programs without shared references, the core proof generated by our technique guarantees memory safety without relying on the checks performed by the compiler.

The verification and specification technique presented in this part can handle a small but technically challenging part of safe Rust. It includes primitive types (`bool`, integers, `char`), safe compound types (tuples, structs, enumerations, and generic type parameters), and references to these types. We provide special support for `Box`, a safe pointer for storing data on the heap, a type defined in the standard library but treated specially by the Rust compiler. The supported fragment also includes functions, methods, and trait methods; we will use *functions* to refer to all three in this thesis. Functions can use generics with trait bounds and lifetime parameters without constraints. Deterministic, side-effect-free functions can be marked as *pure*, which allows using them in specifications. Function bodies may contain branching constructs, loops, boolean and integer operations, function calls, type constructor invocations, casts, assignments that transfer ownership (*move* assignments), and assignments that duplicate ownership (*copy* assignments). Users can specify functional behaviour with pre- and postconditions and loop invariants. Commonly used Rust features that fall outside the supported subset presented in this part include drop handlers, closures, lifetime parameters to struct types, two-phase-borrows, and unsafe code. Note that the language supported by the prototype has additional restrictions; we provide the details in Chapter 6.

[74]: Astrauskas et al. (2019), 'Leveraging Rust types for modular specification and verification'

Contributions. Contributions of the original publication [74] were:

1. A specification language based on Rust expressions that enables modular specification and verification of functional properties.
2. A novel construct for modular specification and verification of Rust functions that return references, which we call *pledges*.
3. A verification approach that uses the Rust type system information to generate a core proof in implicit dynamic frames logic and supports incorporating user-written specifications.
4. Sufficient automation of our approach by using Viper framework that guarantees that the core proof is always accepted by the Viper backend verifier.
5. A prototype implementation of our approach as a Rust compiler plugin.

In addition to the contributions made in our publication, this part of the thesis also compares in depth our approach for modelling Rust borrows with the ones published later and discusses still open challenges.

Verified Properties and Trusted Assumptions. Our verification methodology and its prototype implementation ensure the following property: if a Viper backend verifier accepts the generated proof, every execution of the original Rust function that satisfies the user-written precondition is guaranteed to execute without memory and runtime errors[11] and to

11: Our prototype implementation allows deactivating some of the checks (for example, absence of arithmetic overflows).

satisfy the user-written postcondition. For programs without shared references, this property is ensured under two assumptions: Viper is sound, and our modelling of Rust types and operations in Viper is correct. We could avoid making the latter assumption by proving that our modelling of Rust types and operations in Viper is consistent with Rust operational semantics. However, such a proof would require having semantics for Viper and Rust, both of which are still work in progress at the time of writing. Therefore, we ensured that our modelling of Rust matches the semantics implemented in the Rust compiler by creating a test suite of more than 300 correct and incorrect Rust programs annotated with expected verification errors. For programs with shared references (Chapter 4), we additionally rely on the correctness of the borrow checker to determine when a shared reference expires.

Outline. This part is structured as follows. We start by presenting in Chapter 2 our approach for a Rust fragment without borrows. Then, we show how our approach supports mutable and shared borrows in Chapter 3 and Chapter 4, respectively. In Chapter 5, we discuss three elements necessary for specifying and verifying realistic Rust programs: pledges, pure functions, and snapshots. In Chapter 6, we present and evaluate our implementation. In Chapter 7, we discuss related work that was published before and after our publication [74] and, in Chapter 8, we conclude.

[74]: Astrauskas et al. (2019), 'Leveraging Rust types for modular specification and verification'

Rust's Capabilities for Verification

2

In this chapter, we show how to automatically construct a program in the Viper [38] intermediate verification language in such a way that the correctness of the Viper program implies the correctness of the original Rust program. We start by showing in this chapter our technique for handling Rust programs without references and extend it to handle references in the later chapters. With our methodology, we are aiming to enable building a verifier that works for real Rust programs. Rust has a rich syntax with many syntactic constructs that enable the programmers to express complex ideas concisely. As a result, if we defined our technique on the source level, the critical parts of the technique would likely be obscured by the handling of the complex syntactic constructs. Therefore, we define our technique not for source Rust but for the CFG-based middle intermediate representation (MIR) used in the compiler, where the powerful syntactic constructs are desugared into a few fundamental primitives. However, to make code examples easier to read, we show them using the syntax of source Rust.

We start this chapter by providing background on Rust ownership in Section 2.1. In Section 2.2, we discuss how capabilities differ from ownership and present an algorithm that makes the capability information explicit in the program. Then, Section 2.3 presents how a Rust program with explicit capability information can be automatically encoded into Viper program with a core proof that is guaranteed to be accepted by the verifier. Section 2.4 finishes the chapter by showing how to incorporate user-written specifications in the core proof.

[38]: Müller et al. (2016), 'Viper: A Verification Infrastructure for Permission-Based Reasoning'

2.1 Ownership in Rust

Unlike all other memory-safe systems programming languages that ensure memory safety with a garbage collector, Rust relies on its ownership type system. In Rust, every memory-allocated value is uniquely owned by some variable (function parameters are also variables). Once the owning variable goes out of scope, the value is deallocated. The following example illustrates this behaviour. It shows that the `Box` (a unique pointer for storing values on the heap) containing number `2` is deallocated before the `Box` containing number `1` because its owner, variable `b`, goes out of scope earlier.

```
{
    let a = Box::new(1);
    {
        let b = Box::new(2);
    }   // Box(2) gets deallocated here
} // Box(1) gets deallocated here
```

A value can can be transferred from one owner to another. In this case, the value is not deallocated when the original owner goes out of scope. An assignment that transfers ownership is called *move assignment*[1]. For instance, a modified version of the previous snippet illustrates that if the `Box` ownership is moved from `a` to `b`, it gets deallocated when `b` goes out of scope.

1: Whether an assignment creates a copy or moves depends on whether the type is copyable (implements a `Copy` trait). The most important examples of copyable types are primitive types and shared references.

```
{
    let a = Box::new(1);
    {
        let b = a;  // Box(1) is moved into b
    }   // Box(1) gets deallocated here
}   // since a`s value was moved out, nothing happens
    // when a goes out of scope
```

Ownership in Rust is transitive: the owner of a struct value also owns all of its fields. However, a field could be moved out, making the struct's ownership *partial*. The following snippet illustrates this case by moving out place `pair.first`, which results in the corresponding `Box` getting deallocated earlier.

```
struct Pair {
    first: Box<i32>,
    second: Box<i32>,
}
{
    let pair = Pair {
        first: Box::new(1),
        second: Box::new(2),
    };
    {
        let b = pair.first;
    }   // Box(1) gets deallocated here
} // Box(2) gets deallocated here
```

This example shows that variables are too coarse for tracking ownership, and ownership should be tracked on the granularity of places instead.

Rust guarantees an important safety property that when an owner of a value goes out of scope, and the value gets deallocated, it cannot be reached anymore. To understand this point better, we have to look into how ownership differs from capabilities.

2.2 Capabilities in Rust

If a place `a` owns some value, that does not imply that this value can always be accessed via `a`. For example, as shown in Figure 2.1, if the value is mutably borrowed[2] by some reference `r`, then the capability to access the value is temporarily transferred to `r`. However, the value remains owned by `a`. Knowing what capabilities are held at each program point is crucial for verification because it enables us to address the frame problem: if an activation record has a capability to some value, we can be sure that the value cannot change. Conversely, since `a` temporarily lost its capability, we cannot be sure that the value owned by `a` did not change.

2: In this and the following chapter, we focus on exclusive capabilities. We discuss shared capabilities, as used in the `get_value` method in Figure 1.1, in Chapter 4.

However, since `b` continuously kept capability to the owned value, we can be sure that the value is still the same.

```
1  {
2      let mut a = Box::new(1);
3      let mut b = Box::new(1);
4      {
5          let r = &mut a;
6          // Only r can be used to mutate Box(1).
7          // However, a still owns it.
8          **r = 2;
9      }
10     // r is dead, a can be used for access again
11     assert!(*a == 1);   // Fails.
12     assert!(*b == 1);   // Succeeds.
13 } // a and b go out of scope and boxes get deallocated
```

Figure 2.1: A simple example illustrating how capabilities enable framing knowledge. Since `b` keeps the capability all the time, we know that its value could not change and, therefore, the assert on line 12 is guaranteed to succeed. However, since `a` temporarily lost the capability, we cannot guarantee that its value did not change and, therefore, the assert on line 11 may fail.

While the capability information is useful for verification, unfortunately, it is not available in the Rust source code. The source code does not tell us directly whether we have a capability to a specific place or whether it was borrowed or moved out. The Rust compiler computes some of this information when type-checking a Rust program but not in a form suitable for consumption by external tools. Therefore, we designed the so-called *PCS elaboration algorithm* that takes the information obtainable from the compiler and elaborates a Rust program with capability information suitable for consumption by verifiers. The algorithm presented in this section is optimised for verifying safe Rust code: it does not model implicit deallocation[3] and tracks only the capabilities that correspond to places that can be accessed by safe Rust code. We present a version of the algorithm that supports mixed safe and unsafe code in Part III.

3: Destructors in Rust are implemented by implementing a trait `Drop` and, therefore, have to be behavioural subtypes of this trait. Since the precondition and post-condition of the `Drop::drop` method is `true`, the destructors cannot affect the functional correctness of safe Rust code.

The remaining section is structured as follows. In Subsection 2.2.1, we present place capability sets, a formal way of presenting capability information, and additional operations needed to manipulate them. Subsection 2.2.2 presents our PCS elaboration algorithm for loop-free code and Subsection 2.2.3 extends it to handle loops.

2.2.1 Place Capability Sets

The key information we need for verification is what capabilities we have at each program point and how these capabilities evolve when the program is executed. To capture capabilities, we define *place capability sets* (PCSs) as follows:

Definition 2.2.1 (Place Capability Sets (first version)) Places, *ranged over by p, are expressions defined by the following grammar: $p ::= x \mid p.f \mid (*p)$. For a place p of the form $p'.f$ and $(*p')$, place p' is called a* sub-place *of p; this notion is extended transitively in the natural way. A* place capability set *(PCS) is a finite set of places.*

When a program is executed, capabilities can be changed in two ways. First, executing a Rust statement consumes some capabilities and produces potentially different ones. For example, the following move as-

signment consumes place capability `a` and produces place capability `b`.

```
PCS: { a }
let b = a;
PCS: { b }
```

Second, sometimes, the shape of capabilities needs to be changed. Changing is needed when the shape of capabilities before a statement does not match the shape required by the statement, or capabilities incoming from two different branches need to be unified. For example, the following move assignment requires capability `pair.first` while the available capability is `pair`.

```
let pair = Pair {
    first: Box::new(1),
    second: Box::new(2),
};
PCS: { pair }
???
let b = pair.first;
PCS: { pair.second, b }
```

In such cases, the Rust compiler and our algorithm reshape the capabilities into the required form. Our algorithm makes the reshaping explicit by inserting *PCS operations* before the corresponding statement. In our example, the required capability `pair.first` can be satisfied by *unpacking* the capability `pair` into its constituent capabilities `pair.first` and `pair.second`:

```
PCS: { pair }
unpack pair
PCS: { pair.first, pair.second }
let b = pair.first;
PCS: { pair.second, b }
```

The other two PCS operations are *pack* and *remove*. All three operations are defined as follows:

Definition 2.2.2 (PCS Operations (first version)) *A* PCS operation *is a* remove, unpack, *or* pack *of a capability in a PCS. Remove is defined as the corresponding set operation.*

Let p be a place of struct type, and let $f_1, \ldots, f_n$ be the fields of the struct. For a PCS S such that $p \in S$, the unpacking *of p in S is the PCS $(S \backslash \{p\}) \cup \{p.f_1, \ldots, p.f_n\}$. If p is instead of box type, the unpacking is $(S \backslash \{p\}) \cup \{*p\}$.*

The packing *of p in S is the inverse operation. It is defined only when the $p.f_i$ (or $*p$) are in S.*

In the rest of this section, we present our PCS elaboration algorithm. In Section 2.3, we show how to automatically derive a core proof in an implicit dynamic frames logic for a Rust function elaborated with PCSs and PCS operations. For supporting user-written specifications, we also need a variant of the PCS operation *unpack* that works on expressions. We present it in Section 2.4.

2.2.2 Deriving PCSs and PCS Operations

In the previous subsection, we introduced PCSs and PCS operations. In this subsection, we present an algorithm[4] that computes them for loop-free code. We show how to handle loops in the following subsection.

Our algorithm is a forward pass over a Rust function's CFG that processes a statement only after processing all preceding statements. Since we consider only loop-free code, we can achieve this order by processing statements in reverse topological order. The algorithm computes PCS before and after each statement. The initial PCS at the procedure's entry point contains capabilities of all function parameters. Each statement is processed by collecting its requirements, computing the operations that transform the PCS to satisfy the requirements, and applying the effects of the statement. At join points, the PCSs are unified by trying to pack capabilities; the capabilities not present in at least one of the incoming states are removed because the compiler will conservatively not allow accessing the corresponding places.

4: The algorithm designed by Federico Poli for Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 'Leveraging Rust types for modular specification and verification'. In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019) [74] also computes PCS operations within user-provided assertions, which was needed for the assertion encoding used at that time. In Section 2.4 of this thesis, we present an alternative encoding that does not require PCS operations within assertions. Therefore, in this subsection, we present a simplified algorithm that computes PCS operations only for statements.

```
1  fn modify_first(mut pair: Pair) -> Pair {
2      if random() {
3          let first = pair.first;
4      } else {
5      }
6      pair = Pair {
7          first: Box::new(2),
8          second: Box::new(3),
9      };
10     pair
11 }
```

Figure 2.2: A simple example showing how the first element of a pair is non-deterministically moved out, and then the entire pair is reinitialised.

Figure 2.2 shows a simple Rust function that takes a pair, non-deterministically moves out its first element, and then reinitialises the whole pair. Figure 2.3 shows the same example with computed PCSs and PCS operations. The function's parameter is `pair`; therefore, the initial PCS is `{pair}`. The move assignment on line 7 in Figure 2.3 consumes the capability to its right-hand side (`pair.first`) and produces the capability to its left-hand side (`first`). Since the input PCS has `pair`, but not `pair.first`, the algorithm emits an `unpack pair` operation and applies its effect to the PCS producing `{pair.first, pair.second}`. The algorithm determines what PCS operations it needs to produce by comparing the required capability with the capabilities in the current PCS. If the required place is already in the set, then the requirement is satisfied, and no operations are needed. If the current PCS contains a place that is a prefix of the required place, then the algorithm recursively unpacks the prefix until the PCS contains the required place. Otherwise, the algorithm obtains the required place by packing. Since the algorithm is executed on type-checked code, it is guaranteed to succeed. After unpacking `pair`, the PCS has the required place `pair.first` and, therefore, the algorithm proceeds with applying the effects of the statement: it removes `pair.first` and adds `first` to the PCS.

After processing both branches of the if statement, the algorithm needs to unify their PCSs. The algorithm unifies PCSs by trying to find for each capability in one branch a matching capability in the other branch. If it

```
1  fn modify_first(mut pair: Pair) -> Pair {
2      PCS: { pair }
3      if random() {
4          PCS: { pair }
5          unpack pair
6          // Consume: { pair.first } Produce: { first }
7          let first = pair.first;
8          PCS: { first, pair.second }
9          remove first
10         PCS: { pair.second }
11     } else {
12         PCS: { pair }
13         unpack pair
14         remove pair.first
15         PCS: { pair.second }
16     }
17     PCS: { pair.second }
18     // Soft-drop: { pair } Consume: { } Produce: { pair }
19     pair = Pair {
20         first: Box::new(2),
21         second: Box::new(3),
22     };
23     PCS: { pair }
24     // Consume: { pair } Produce: { }
25     pair
26 }
```

Figure 2.3: The example from Figure 2.2 with elaborated PCSs and PCS operations.

fails to find an exact match, the algorithm tries to obtain the match by first packing, and if packing does not work, then unpacking capabilities. If both fail, the capability is removed. In our example, the then-branch has PCS `{first, pair.second}`, and the else-branch has PCS `{pair}`. `first` does not exist in the else-branch PCS, so it gets removed by adding a `remove first` PCS operation to the then-branch. The next capability checked is `pair.second`. Since the else-branch PCS does not have `pair.second` and `pair` is a prefix of `pair.second`, the algorithm checks whether it can obtain the `pair` capability in the then-branch by packing it. This check fails because the then-branch does not have `pair.first` capability. The algorithm proceeds by unpacking `pair` in the else-branch PCS to obtain `{pair.first, pair.second}`. After this operation, both branches have the `pair.second` capability. Ultimately, the algorithm tries to match `pair.first` from the else-branch, fails, and removes it. The final unified PCS is `{pair.second}`.

The last move assignment on lines 19–22 consumes no capabilities. However, it reinitialises the place `pair`, which means that if we have any capabilities to `pair`, they need to be removed as indicated by "Soft-drop" in the comment on line 18. More precisely, we need to remove any capabilities that have `pair` as a prefix. In our example, `pair.second`. The statement produces capability `pair`; therefore, the final PCS before the end of the statement is `{pair}`.

The PCS elaboration algorithm we presented is completely syntax-driven, which is enabled by two observations. First, in safe Rust programs without references, two syntactically different places are guaranteed to

identify two different locations. This property allows us to treat each place capability independently without worrying that, for example, unpacking a capability through one place would invalidate the view through another place. Second, since we only need to deal with user-written places that are of finite size, we are guaranteed that our algorithm will need only a finite number of steps.

2.2.3 Handling Loops

In the previous subsection, we presented an algorithm for computing PCSs and PCS operations for loop-free code. In this subsection, we extend the algorithm to handle loops.

A standard way of handling loops in deductive verification is by using *loop invariants*, a condition maintained by the loop that allows capturing the effect of unbounded many iterations in a finite way. In our context, a loop invariant must specify all capabilities required to execute the loop body safely. Inferring loop invariants for general programs is an open research problem [80]. Luckily, Rust was designed so that the type checker could efficiently check the capabilities of the code with loops. However, computing the loop invariant PCS is still non-trivial because the invariant should be minimal to frame around as much knowledge as possible. Consider the example shown in Figure 2.4 with two variables `a` and `b`. Only `b` is modified in the loop body; therefore, we would like to prove the assertion on the last line without any user input. However, if we put the capability `a` into the loop invariant, the verifier will conservatively have to assume that `a` could be modified by the loop and thus not frame its value. Therefore, we must ensure that the loop invariant PCS for this example contains only `b`.

[80]: Calcagno et al. (2009), 'Compositional shape analysis by means of bi-abduction'

```
1 let a = 5;
2 let mut b = 1;
3 while random() {
4     b += 1;
5 }
6 assert!(a == 5);
```

Figure 2.4: A simple example showing a loop.

We compute the loop invariant in two steps[5]. First, we execute our regular algorithm up to the loop invariant, which gives us a candidate PCS. For the example in Figure 2.4, the candidate PCS is `{a, b}`. Second, we obtain from the compiler the list of statements that belong to the loop body and filter out from the candidate PCS the capabilities that are not needed for these statements. Since in our example `a` is not used in the loop body, the remaining loop invariant PCS is `{b}`. Since our algorithm is executed on a type-checked Rust program, the loop body is guaranteed to preserve the loop invariant PCS that was computed in this way.

5: We present an algorithm that expects the loop invariant at the loop head. It can be generalised to supporting loop invariants at any point in the loop by duplicating the code between the loop head and the invariant.

This section presented a slightly simplified version of the algorithm. The complete algorithm distinguishes between read and write accesses. For modelling the former, we use *shared* capabilities, which we present in Chapter 4.

2.3 Constructing the Core Proof

In the previous section, we showed how to elaborate Rust functions with explicit capability information. In this section, we show how that information can be used for generating a core proof in the Viper [38] intermediate verification language. Viper assertions are based on implicit dynamic frames [19], a powerful permissions logic. A Viper program consists of methods, fields, and predicates. Our goal is to encode a Rust function into a Viper method in such a way that the method is guaranteed to be successfully verified by a Viper verifier without any additional input. Achieving this automation requires convincing the verifier that each memory access is justified because the activation record holds the required permission. In the following subsections, we show how the place capability information can be used to achieve the desired automation. We start by giving the necessary background on Viper and showing what is needed for verifying a program in Viper. Then, we show how we encode a Rust function with capability information into a Viper method. We finish this section by describing how we handle each kind of Rust type and showing a complete encoding of the Rust function from Figure 2.3.

[38]: Müller et al. (2016), 'Viper: A Verification Infrastructure for Permission-Based Reasoning'

[19]: Smans et al. (2009), 'Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic'

2.3.1 Viper Background

As mentioned in the introduction, in Viper, permissions are specified by using an accessibility predicate `acc(R, p)` that gives `p` permission amount to a resource `R`. In this chapter, `p` is always `write`, indicating that the resource can be both read and mutated. A resource `R` can be either a field or a predicate. Predicates are crucial for modelling recursive data structures. For example, a linked list of mathematical integers in Viper could be modelled by using the following predicate that refers to itself.

```
1  field value: Int
2  field next: Ref
3  predicate LinkedListNode(this: Ref) {
4      acc(this.value, write) &&
5      acc(this.next, write) &&
6      (this.next != null ==>
7          acc(LinkedListNode(this.next), write))
8  }
```

This snippet defines two fields: an integer `value` and a reference `next`. The predicate `LinkedListNode` takes a reference to a node as an argument and gives full permissions to its `value` and `next` fields. Also, if field `next` is not `null`, the predicate gives full permission to the remainder of the linked list. Recursive predicates pose a challenge to automatic verifiers because they may get stuck by infinitely applying a predicate definition. Therefore, automated verifiers such as Viper treat predicates isorecursively [81–83], which means that the user is required to explicitly specify when to apply a predicate definition. Viper provides statements `unfold P` that replaces the predicate instance `P` with its body and `fold P` that does the opposite[6]. Predicates are also important for modelling abstract assertions; for example, an abstract Viper predicate (a predicate without body) can be used for modelling unknown assertions.

[81]: Abadi et al. (1996), 'Syntactic Considerations on Recursive Types'

[82]: Crary et al. (1999), 'What is a Recursive Module?'

[83]: Summers et al. (2013), 'A Formal Semantics for Isorecursive and Equirecursive State Abstractions'

6: In this thesis, we refer to operations such as Viper's `fold` and `unfold` statements that are needed only for convincing the verifier that the code is correct as ghost operations

Predicates enable us to specify how a data structure looks at a specific moment. However, executing a program changes the shape of the memory: memory gets allocated and deallocated, and permissions are transferred from one activation record to another. In Viper, such effects are modelled with `inhale` and `exhale` statements. Statement `inhale A` assumes the pure parts of assertion `A` and adds the permissions mentioned in its accessibility predicates to the current state. The dual of `inhale` is `exhale A` that asserts the pure parts of the assertion `A`, checks that the current state contains enough permission to satisfy the permissions mentioned in the accessibility predicates, and removes these permissions from the state. For example, the following snippet shows how a program that allocates an object `r` with field `value`, modifies the field, and deallocates it could be modelled in Viper.

```
1  // Initially r has some unknown value.
2  var r: Ref
3
4  // Allocate the object.
5  inhale acc(r.value, write)
6
7  // Write to the field.
8  r.value := 42;
9
10 // Deallocate the object.
11 exhale acc(r.value, write)
```

Another vital use of `inhale` and `exhale` is to model permission transfer between contexts such as caller and callee, or loop body and surrounding context. The permissions (and knowledge) are transferred by exhaling them on the sending side and inhaling them on the receiving side. For example, if we have method `callee` with precondition `P` and postcondition `Q`, we could model a call to it as shown in the snippet below.

```
1  method callee()
2  {
3      inhale P    // inhale precondition
4      ...
5      exhale Q    // exhale postcondition
6  }
7  method caller()
8  {
9      ...
10     exhale P    // exhale precondition
11     inhale Q    // inhale postcondition
12     ...
13 }
```

The caller sends assertion `P` by exhaling it, and the callee receives it by inhaling. Similarly, the postcondition `Q` is transferred from the callee to the caller by exhaling it on the callee side and inhaling it on the caller side. Viper's inhale and exhale are powerful primitives that enable succinct modelling of various concepts.

2.3.2 Encoding Rust Functions in Viper

So far, we have seen how to compute explicit capability information for Rust functions and what is needed for verifying a Viper method. When linking the two, the main question is how to model the place capabilities in Viper. The memory locations to which a place capability gives access depend on the place's type, which in Viper can be captured by an assertion. Therefore, we model Rust types with Viper predicates, which are named assertions. For example, the following snippet shows how we model a signed 32-bit integer in Viper (we show how we model the main Rust kinds of types in the following subsection).

```
predicate i32(self: Ref) {
    acc(self.val_i32, write) &&
    MIN_i32 <= self.val_i32 && self.val_i32 <= MAX_i32
}
```

The value of the integer is stored in the field `val_i32`. The type of the field is `Int`, a built-in Viper type for unbounded integers. The predicate gives access to the field storing the value and constrains the value to be in the valid range for 32-bit signed integers (if overflow checking is disabled, the bounds are omitted). Parameter `self` is the identity of the object. We chose to use object identity instead of an address for identifying objects because addresses in safe Rust are of little value (pointer arithmetic requires using unsafe code), enabling us to simplify the model. For example, this design choice allows us to encode Rust move assignment `let b = a;` into Viper as simple assignment `b := a;` followed by setting `a` to `null`, which automatically takes care of the permission transfer.

Assigning to a newly declared variable in Rust creates a capability for that variable, as shown by the following example.

```
PCS: {}
let mut a: i32 = 1;
PCS: { a }
```

We model the effects of creating and destroying place capabilities with `inhale` and `exhale` statements. We encode the given snippet to Viper by inhaling the permissions to the value field, writing a new value, and folding the predicate instance corresponding to the created capability.

```
inhale acc(a.val_i32, write)
a.val_i32 := 1;
fold acc(i32(a), write)
```

We also use `inhale` and `exhale` statements to model capability transfer. The capabilities of function parameters are transferred into the function when it is called, and the capability of the return place is transferred to the caller when the function terminates. Therefore, the capabilities of the parameters belong to the function precondition, and we encode them into Viper by exhaling the corresponding predicate instances on the call side and inhaling the corresponding predicate instances on the callee side. Similarly, the return place belongs to the function postcondition, and we encode it by exhaling the corresponding predicate on the callee side and inhaling it on the caller side.

The last ingredient we need for generating the core proofs is handling PCS operations. The PCS operations `pack` and `unpack` naturally map to Viper statements `fold` and `unfold`. Similarly, the `remove` operation that removes a capability from a state is modelled by using Viper statement `exhale`.

2.3.3 Modelling of Rust Type Kinds

This subsection describes how we model each kind of Rust types.

As already explained, **primitive types** like `bool`, `i32`, `u64`, and `char` are modelled by having a Viper field with potentially additional constraints wrapped in a predicate. For example, the predicate for `u64` is defined similarly to the one for `i32`.

```
predicate u64(self: Ref) {
    acc(self.val_u64, write) &&
    0 <= self.val_u64 && self.val_u64 <= MAX_u64
}
```

Generic type parameters are modelled as abstract Viper predicates.

Safe abstractions are types that are internally implemented by using unsafe code but provide a safe abstraction for the clients. Since the technique presented in this part does not support unsafe code, we model such types as abstract Viper predicates. We show how some of these types could be supported in Part III.

Structs are modelled as a conjunction of their field predicates. For example, the following struct:

```
struct Pair {
    first: Box<i32>,
    second: Box<i32>,
}
```

is encoded with the following predicate:

```
predicate Pair(self: Ref) {
    acc(self.first, write) &&
    acc(Box_i32(self.first), write) &&
    acc(self.second, write) &&
    acc(Box_i32(self.second), write)
}
```

The accessibility predicate `acc(self.first, write)` gives full permission to the field that stores the identity of the Rust field `first` while `acc(Box_i32(self.first), write)` gives permission to the contents of the field. Storing the identity of the field object in field `first` enables us to move in and out values in a simple way. For example, the following Rust move assignment:

```
pair.first = new_value;
```

is encoded as:

```
pair.first := new_value;
```

Enums are modelled as a conjunction of variants guarded by the enums' discriminant. For example, the following enum:

```
struct Option<T> {
    None,
    Some(T),
}
```

is encoded with the following predicate:

```
predicate Option(self: Ref) {
    acc(self.discriminant, write) &&
    (self.discriminant == 0 || self.discriminant == 1) &&
    (self.discriminant == 0 ==>
        acc(Option_Variant_None(self), write)) &&
    (self.discriminant == 1 ==>
        acc(Option_Variant_Some(self), write))
}
```

The discriminant field `self.discriminant` indicates which of the enum's variants is active. The disjunction `(self.discriminant == 0 || self.discriminant == 1)` ensures that the discriminant is always valid[7]. The last two conjuncts give permission to the corresponding variant if the discriminant has the matching value. We encode predicates for variants of an enum (in our example, `Option_Variant_None` and `Option_Variant_Some`) using the encoding of structs. For example, variant `None` is encoded as an empty struct and variant `Some` is encoded as a struct with a single field.

7: A nice consequence of this encoding is that it naturally captures uninhabited types like `Void` [84] that are declared as empty enums because this disjunction becomes `false`. Uninhabited types are used, for example, to inform the compiler that a specific path is unreachable.

Box<T> is a safe abstraction providing a safe pointer to heap-allocated memory. While the type is defined in the standard library, it is also recognized and treated in a special way by the compiler, for example, by allowing special syntax in match statements. Therefore, we also special case boxes by encoding them as follows:

```
predicate Box_T(self: Ref) {
    acc(self.pointer, write) &&
    acc(T(self.pointer), write)
}
```

`pointer` is a field of type `Ref` that points to the heap-allocated part of the box while `T(self.pointer)` represents the heap-allocated value of type `T`.

2.3.4 Encoded Example

With all the details covered, we can finally show a complete encoding of a Rust function. Figure 2.5 shows a slightly simplified for readability encoding of Figure 2.3 into Viper.

```
method modify_first(pair: Ref) returns (_result: Ref)
{
    // Inhale precondition.
    inhale acc(Pair(pair), write)

    var first: Ref

    if random() {
        unfold acc(Pair(pair), write)
        first := pair.first;
        exhale acc(i32(first), write)
    } else {
        unfold acc(Pair(pair), write)
        exhale acc(i32(pair.first), write)
    }

    exhale acc(i32(pair.second), write)

    // Construct pair.first.
    inhale acc(pair.first, write) &&
           acc(pair.first.pointer, write) &&
           acc(pair.first.pointer.val_i32, write)
    pair.first.pointer.val_i32 := 2;
    fold acc(i32(pair.first.pointer), write)
    fold acc(box_i32(pair.first), write)

    // pair.second is constructed in the same way.

    fold acc(Pair(pair), write)

    // Exhale postcondition.
    exhale acc(Pair(pair), write)
}
```

Figure 2.5: Encoding of Figure 2.3 into Viper.

2.4 Functional Specifications

The core proof that we showed how to construct in the previous section reproves the guarantees that are already checked by the type system. However, this core proof contains information on how permissions and values flow through the program. In this section, we show how this information enables verifying the functional correctness of the program.

We start by showing in Subsection 2.4.1 how our approach enables us to check whether user-specified assertions and checks inserted by the compiler could fail at runtime. While proving the absence of runtime errors is valuable, users often want to prove full functional correctness, which is enabled by Prusti's specification language. In Subsection 2.4.2, we show how the core proof can be extended to accommodate user-written specifications. The steps needed for supporting pure functions are very similar, but since pure functions can only read memory, most use cases of pure functions rely on shared references. Therefore, we present how we handle pure functions in Section 5.1 after we presented shared

references.

2.4.1 Checking Absence of Panics and Assertion Failures

As mentioned in the introduction, the simplest way to specify functional behaviour is by using `assert!(...)` macros, as shown in the following example that computes the distance between two points.

```
fn compute_distance(start: i32, end: i32) -> i32 {
    assert!(end > start);
    end - start
}
```

The Rust compiler expands the `assert!(...)` macro into an if statement that starts a panic when the asserted condition does not hold, as shown in the snippet below.

```
fn compute_distance(start: i32, end: i32) -> i32 {
    if !(end > start) {
        // A call to a compiler intrinsic that starts
        // a panic.
    }
    end - start
}
```

Therefore, to prove that the condition in the `assert!(...)` never fails, we only need to prove that the then-branch of the if condition is unreachable. We can make the Viper verifier check this property by encoding the compiler intrinsic that starts the panic into Viper with an `assert false` statement, which checks that the condition `false` holds at that state. With this approach, we can check not only that user-written asserts always succeed but also that other runtime checks performed by Rust such as overflows and division by zero always succeed, too. If a user is not interested in checking that the runtime checks never fail, we can easily turn off these checks by encoding panics as `assume false`, which assumes that the branch is unreachable. By providing the possibility to turn off the checks, we enable the user to focus on the properties they are most interested in.

2.4.2 Specifying and Verifying Functional Properties

Often, the user needs to express properties beyond the ones that are easy to express with assert statements, for example, that a vector is sorted. As shown in the following snippet, such properties can be expressed with our specification language that is based on Rust expressions but provides powerful first-order logic extensions such as quantification. The postcondition in the example expresses that for any two indices `i` and `j`, if `i` is not smaller than `j`, then the `i`th element of the vector will be non-smaller than the `j`th element[8].

8: `lookup` in this snippet is a pure method that returns the value stored at the given index. We will cover pure functions in Section 5.1

```
#[ensures(
    forall(i: usize, j: usize
        (i <= j && j < result.len()) ==>
        result.lookup(i) <= result.lookup(j)
    )
)]
fn create_sorted_vector() -> VecI32 { /* ... */ }
```

Supporting user specifications requires overcoming three challenges. The first challenge is guaranteeing that user-written specifications are sufficiently stable for verifying concurrent and heap-manipulating programs. For example, Dafny [43] does not support verifying concurrent code because its specification language is not resilient against data races. We ensure the stability of our specifications by type-checking them with the Rust type-checker, which ensures that available capabilities frame them. For example, a function precondition can only mention the function parameters because these are the only places the called function has capabilities at its invocation.

[43]: Leino (2010), 'Dafny: An Automatic Program Verifier for Functional Correctness'

The second challenge is computing PCSs and PCS operations for specifications. For example, the state may have a capability to the entire `pair` while the specification mentions only its first field `pair.first`. To justify using `pair.first`, we need to unpack the capability *within* the specification without affecting the outer PCS state because the specifications should have no side effects. Therefore, we introduce a PCS operation `unpacking p in e` that is similar to `unpack p` but works on expressions instead of statements: the expression `e` is evaluated in a state where the capability `p` is unpacked. We also extend the algorithm presented in Subsection 2.2.2 to infer `unpacking` operations in specifications. A simplified version of the extension would be for each place `p` mentioned in the specification, find a place capability that is a prefix of `p` (such place capability is guaranteed to exist for type-checked programs) and insert the necessary amount of `unpacking` operations. Such an extension is sufficiently expressive but suboptimal: it generates significantly more than necessary `unpacking` operations. Federico Poli[85] designed a more complex version of the algorithm that significantly reduces the number of `unpacking` operations.

[85]: Poli (2024), 'Enabling Rich Lightweight Verification of Rust Software'

The third challenge is integrating user specifications into the core proof. This challenge turns out to be surprisingly easy due to our choice to use the implicit dynamic frames logic. As was mentioned in the introduction, implicit dynamic frames supports heap-dependent expressions such as `pair.first == 4`. Therefore, we can conjoin the user-written specifications to the specifications generated from types. We only need to ensure that all heap-dependent expressions are framed, which we achieve by translating the `unpacking` PCS operations into an `unfolding acc(P, write) in e` Viper expression, which evaluates `e` in a state where the predicate instance `P` is unfolded.

```
#[ensures(result.first == old(a))]
fn new_pair(a: i32, b: i32) -> Pair {
    // ...
}
```

Figure 2.6: A simple example showing a constructor of a `Pair` with a postcondition. Note that `old` around `a` is inserted automatically; we show it here for clarity.

Figure 2.7 demonstrates our encoding of the postcondition shown in

```
1  method new_pair(a: Ref, b: Ref) returns (_res: Ref)
2  {
3      // Inhale preconditions.
4      inhale
5          // Preconditions from types.
6          acc(i32(a), write) &&
7          acc(i32(b), write) &&
8          // Functional preconditions.
9          true
10     label pre
11
12     ...
13
14     // Exhale postconditions.
15     exhale
16         // Postconditions from types.
17         acc(Pair(_res), write) &&
18         // Functional postconditions.
19         (unfolding acc(Pair(_res), write) in
20          unfolding acc(Box_i32(_res.first), write) in
21          unfolding acc(i32(_res.first.pointer), write) in
22          _res.first.pointer.val_i32)
23         ==
24         (old[pre](
25          unfolding acc(i32(a), write) in
26          a.val_i32))
27 }
```

Figure 2.7: A simplified encoding of Figure 2.6. The inhaled precondition has `true` as a last conjunct because the precondition defaults to `true` if omitted. `old[l](e)` is a labelled old expression that evaluates the expression `e` in a state marked with label `l`.

Figure 2.6. Figure 2.6 shows a signature of a function that constructs a new pair and whose postcondition guarantees that the value of field `first` is equal to the value that the parameter `a` had at the function's prestate. Figure 2.7 shows an encoding of the specification in Viper. The `inhale` statement on lines 4–9 inhales the function's precondition. The conjuncts on lines 6–7 inhale the permissions that were computed from the type information and the conjunct on line 9 inhales the user-written precondition. Since the user wrote no precondition, it defaults to `true`. Statement `label pre` on line 10 saves the prestate so that later the labelled old expression `old[pre](e)` could be used to evaluate expressions in that state. The postcondition is exhaled on lines 15–26. The conjunct on line 17 exhales the permission to the result value and the conjunct on lines 19–26 exhales the user-written specifications. All `unfolding` expressions in the functional specification are automatically computed by our technique, even though the two sides of the equality use permissions from different states: the left side uses the current state, and the right side uses `pre` state.

3 Mutable Borrows

In the previous chapter, we presented how to handle Rust programs that use move and copy assignments. Moving values around is often both cumbersome and inefficient. Therefore, most Rust programs extensively use references. Since Rust aims to guarantee that a memory location is either mutable or shared, but never both, mutable references in Rust are unique: they temporarily *borrow* the capability from the borrowed place. In Figure 3.1, the owned value `a` is mutably borrowed by `x` on line 4, which transfers the capability for mutating the memory location from place `a` to `*x`. The capability is returned after the last use of `*x` on line 8. As a result, the compiler rejects the assignment to `a` on line 6 but accepts the one on line 14. As shown on line 10, the place capability for reference `x` contains two capabilities inside it: capability `x.pointer`, which is the capability to the memory location containing the address of the target, and capability `x.*`, which is the capability to the target itself. When a reference expires, the capability to the target is transferred back, but the capability to the address remains, allowing reassigning the reference to point to a new location. Similarly to other place capabilities, place capabilities for references can be unpacked and packed using the corresponding PCS operations.

```
1   PCS: { }
2   let mut a = 1;
3   PCS: { a }
4   let x = &mut a;
5   PCS: { x }
6   // a = 2;       // Illegal!
7   PCS: { x }
8   *x = 3;
9   unpack x
10  PCS: { x.*, x.pointer }
11  PCS: { a, x.pointer }
12  assert!(a == 3);
13  PCS: { a, x.pointer }
14  a = 4;
```

Figure 3.1: A simple example of borrowing showing the flow of capabilities.

A reference can not only temporarily borrow a capability from an owning place but also temporarily *reborrow* a capability from another reference as shown in Figure 3.2 where a reference `y` temporarily borrows the target of reference `x`. Reborrowing (borrowing the target of another reference) should not be mixed up with borrowing the entire reference. The latter takes the capability to the entire reference, including its target. The former takes the capability to the target but leaves the capability to the pointer that stores the address. While the reborrowing pattern at first glance looks obscure, in Rust, it is used all the time and supporting it is crucial. For example, accessing an element of a vector gets desugared into reborrow. Also, the Rust compiler often replaces moves of references

with reborrows to allow more code to be accepted (moving a reference moves the capability permanently while reborrowing it also allows to restore it once the reborrowing reference expires).

```
1  PCS: { }
2  let mut a = 1;
3  PCS: { a }
4  let x = &mut a;
5  PCS: { x }
6  let y = &mut *x;
7  PCS: { y, x.pointer }
8  *y = 2;
9  PCS: { x.*, x.pointer }
10 *x = 3;
11 PCS: { a }
12 a = 4;
```

Figure 3.2: A simple example of reborrowing showing the flow of capabilities. Since `y` borrows only the target of `x`, the capability to the actual pointer (`x.pointer` that stores the address of the target remains with `x`.

As already noted earlier, for verification, it is crucial to know how the capabilities flow through the program because it gives us information about the flow of values. The algorithm we presented in Section 2.2 that makes the capability information explicit relies on move assignments explicitly moving capabilities from one place to another. A borrow statement `let x = &mut a;` also explicitly moves capabilities from the lender place `a` to the borrowing place `x`. Therefore, the presented algorithm can also be used for computing PCSs for *forward* capability flow through borrows. However, when a reference expires, the capabilities are implicitly returned to the lender: in Figure 3.1, there is no information that capability from `x.*` has to go to `a`; the Rust compiler only knows that `x.*` loses its capability and `a` regains the capability. Therefore, to be able to prove the assertion on line 12 in Figure 3.1, we have to make the implicit *backward* capability flow explicit to the verifier.

The examples above illustrate a *bounded* forward capability flow. However, a function using a mutable reference to traverse a linked list leads to *unbounded* forward capability flow. This unbounded flow poses the challenge of expressing the backward capability flow in a bounded way. Our key insight is that we can abstract over backward capability flow by introducing an *exchange capability*:

Definition 3.0.1 (Exchange Capability) *An exchange capability (EC) `P1 → P2` is a capability to exchange place capability set `P1` into a place capability set `P2`.*

Intuitively, `P1 → P2` expresses that `P2` is *blocked* by `P1` because forward capability flow derived `P1` from `P2` and, therefore, we can regain `P2` by giving up `P1`. There are two ways to obtain an exchange capability: execute a borrow statement or derive it from other exchange capabilities. For example, executing the borrow statement `let x = &mut a;` produces exchange capability `{ x.* } → { a }`. Exchange capability `{ x.* } → { a }` enables us to regain capability `a` by giving up **both** the exchange capability and capability `x.*`. An exchange capability can be derived using `prove-exchange` operation, which is one of the exchange capability operations[1]:

1: Some exchange capability operations are unnecessary for the Viper encoding because Viper can compute the necessary actions itself. However, we still introduce them to make changes to PCSs explicit.

Definition 3.0.2 (EC Operations) *An* EC operation *is* `exchange P1 → P2` *or* `prove-exchange P1 → P2 G`.

Executing `exchange P1 → P2` *in PCS S checks that* `P1` *is a subset of S, consumes exchange capability* `P1 → P2`*, and updates S to* $(S \backslash \texttt{P1}) \cup \texttt{P2}$.

Executing `prove-exchange P1 → P2 G` *executes the loan-dependency graph* `G` *(defined below) to prove exchange capability* `P1 → P2`.

Exchange capabilities enable us to abstract over arbitrary forward flow, including recursive function calls: a function that takes reference arguments returns an exchange capability that enables the caller to undo the forward capability flow performed by the function. The returned exchange capability could be either used in an `exchange` operation to regain the originally borrowed capabilities or used in `prove-exchange` as a building block to build another exchange capability that is returned to the caller. Similarly to PCSs, reliably computing EC operations requires solving multiple technical challenges, which we approach by introducing a *loan-dependency graph*[2] that captures the backward capability flow in a local context, such as a method or loop body. To simplify the presentation of the loan-dependency graph, we introduce an operation `expire-borrows G` that uses a loan-dependency graph `G` to expire borrows at a given program point. This operation stands for a `prove-exchange` immediately followed by the corresponding `exchange`.

2: We originally called it *reborrowing DAG (directed acyclic graph)*. We still use this name in the Prusti code repository.

We start this chapter by presenting in Section 3.1 different implementations of the borrow checker, the part of the Rust compiler that ensures that capabilities are not duplicated. We explain why the borrow checker does not need to explicitly track backward capability flow and what helpful information it has for us to restore that backward flow. Section 3.2 discusses the challenges that must be addressed when computing the backward flow for Rust programs. In Section 3.3, we present the loan-dependency graph and explain how we construct it. We finish this chapter by showing in Section 3.4 how we extend the core proof to support mutable borrows.

3.1 Overview of Rust Borrow Checkers

A borrow checker is a part of the Rust compiler type checker that, as the name implies, ensures the correct use of borrows. Since the borrow checker is Rust's "secret sauce" [86], there is continuous work on trying to improve it. This work resulted in several versions of the borrow checker, each with different properties. Here, we briefly discuss them from a verification point of view.

[86]: Developers (2023), *MIR borrow check*

Lexical Lifetimes. The first version of Rust shipped with a borrow checker requiring a borrow to last until the end of the scope. This borrow checker would reject all three accesses to `a` in Figure 3.1 as illegal. The motivation for this design was to have a bit more restrictive but easily predictable behaviour. However, the lexical treatment of borrows turned out to be very annoying for programmers. Also, the lexical borrow checker was defined on an AST, and it turned out that correctly implementing a

borrow checker on ASTs for a complex language like Rust is hard [86]. For these two reasons, the Rust compiler developers decided to switch to a borrow checker based on non-lexical lifetimes [87]. When we started our work, it was already clear that the non-lexical borrow checker would replace the lexical one.

[86]: Developers (2023), *MIR borrow check*

[87]: Developers (2017), *The Rust RFC Book: nll*

Non-Lexical Lifetimes. In the Rust 2018 edition[3], a new borrow checker was shipped based on non-lexical lifetimes [73]. This borrow checker is currently the default one in Rust. As we presented in the introduction of this chapter, in this version, a borrow lasts until the last use of the reference. This version of the borrow checker guarantees that capabilities are not duplicated without explicitly tracking the backward flow. It achieves this property using an indirection via lifetimes, which in this version of the borrow checker are sets of program points. For example, the following snippet shows a borrowing statement where a reference `x` borrows `a` with lifetimes made explicit.

3: Rust editions are similar to versions, but allow introducing non-backwards compatible changes.

[73]: Developers (2018), *Announcing Rust 1.31 and Rust 2018*

```
1  let x: &'lb mut i32 = &'ll mut a;
```

In this example, a lifetime `'ll` is the set of all program points for which place `a` is loaned while `'lb` is the set of all program points in which borrow `x` is alive. Since `x` has a capability at program points `'lb`, `a` must not be used at these program points and, therefore, `'lb` $\subseteq$ `'ll`. Having these constraints, the borrow checker can compute minimal sets that satisfy them and check whether, for example, `a` is not used at any program point that is included `'ll`. Unfortunately, these constraints do not contain enough information to reconstruct the explicit backward flow of capabilities, which we need to show the verifier how changes to the borrow affect the loan.

Polonius. We discussed the idea of an algorithm that computes the backward flow of capabilities without relying on the borrow checker with Nicholas D. Matsakis, the leader of the Rust language and compiler teams, and he came up with a new version of the borrow checker called Polonius [88]. Polonius performs a may-borrows analysis, which is much closer to standard may-alias analyses used in other compilers and tools. The advantage of Polonius for Rust programmers is that it accepts more correct programs than the non-lexical lifetimes borrow checker. The advantage of Polonius for verification is that, while it does not compute explicit backward flow, its results are much easier to reuse for this purpose. The key difference between Polonius and previous borrow checkers is the definition of the lifetime: in previous approaches, the lifetime is a set of program points, while, in Polonius, the lifetime is a set of loans. Figure 3.3 shows how Polonius uses lifetimes to track which loans could be borrowed by which references. Reference `x` declared on line 1 has lifetime `'lx`, which initially is empty. On line 4, `x` borrows variable `a`; this loan is given identifier `L1`. After this borrow, the lifetime `'lx` is a singleton set containing `L1`. Similarly, after the borrow of `b` in the other branch, `'lx` is a singleton set containing `L2`. After the if statement, the two sets are merged, so we know that the assignment on line 11 affects either `a` (through loan `L1`) or `b` (through loan `L2`).

[88]: Matsakis (2018), *An alias-based formulation of the borrow checker*

Polonius tells us for each loan which reference may borrow it. We decided to reuse as much as possible information from Polonius because it gives

```
let x: &'lx i32;
// 'lx = { }
if random_choice() {
    x = &mut a; // L1
    // 'lx = { L1 }
} else {
    x = &mut b; // L2
    // 'lx = { L2 }
}
// 'lx = { L1, L2 }
*x = 4;
```

Figure 3.3: An illustration of how Polonius treats lifetimes.

us confidence that our treatment of capabilities matches the Rust compiler. In the following sections, we discuss the challenges we need to address to turn the information available in Polonius into an explicit backward capability flow and our solution to them.

3.2 Challenges in Reconstructing Backward Flow

At the beginning of this chapter, we discussed the exchange capabilities that are our solution to challenge *C0*:

Modularity (*C0*). Since recursive functions can generate unbounded reborrowing, we need a mechanism for capturing unbounded backward capability flow.

In this section, we present three additional challenges that we need to address to be able to reconstruct explicit backward capability flow from the may-borrow information available in Polonius.

Handling effects of PCS operations (*C1*). When a borrow expires and the borrowed place is usable again, we need to know not only that the capability was returned but also the shape of the capability. For example, if the borrowed capability was unpacked like in the following snippet, the backward flow needs to pack it back or bring it to some other well-defined state.

```
PCS: {pair}
let x = &mut pair;
PCS: {x}
unpack x
PCS: {x.pointer, x.*}
unpack x.*
PCS: {x.pointer, x.*.first, x.*.second}
(*x).first = 5;
PCS: {pair}
```

Handling conditional control flow (*C2*). In the example shown in Figure 3.3, reference `x` conditionally borrows either `a` or `b`. As we can see from the lifetime information shown on line 10, Polonius knows that `x` could be borrowing `a` or `b` but does not know which one exactly is borrowed on a specific execution. However, we must know the precise information to restore the capabilities when reference `x` expires.

Handling reference reassignments (*C3*). In Rust, accessing a place that is borrowed is illegal as long the borrow is active. However, reassigning the reference whose **target** was reborrowed is allowed, as shown by the following snippet.

```
1  let mut x = &mut a;
2  let y = &mut *x;
3  x = &mut b;          // Allowed because x itself was not
4                       // borrowed.
5  *y = 5;
```

This example demonstrates that when reference `y` expires, reference `x` has a different meaning than the one it had when `y` was created. In particular, the backward capability flow must not try to use the new place `x` for capability transfer.

3.3 Reconstructing Backward Flow

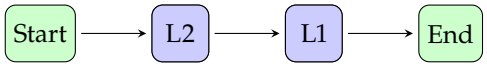
In the introduction of this chapter, we mentioned that we use a *loan-dependency graph* to capture backward capability flow in a local context. The loan-dependency graph is a directed acyclic graph that states how, by using PCS and EC operations, we can transform one PCS into another PCS. The `expire-borrows` operation uses this graph to return capabilities from references to original places, and the `prove-exchange` operation uses it to prove a new exchange capability. In this section, we show how we iteratively define and construct the loan-dependency graph, addressing all the challenges mentioned in the previous section.

When a reference (or, in a more general case, a set of references) expires, Polonius computes which loans get unblocked. At this point, the capabilities are implicitly transferred from the expired references to the borrowed places. We insert `expire-borrows` operations at these points to make the capability transfer explicit. The unblocked set of loans have dependencies between them. Figure 3.4a shows a Rust snippet where variable `a` is borrowed by reference `x` that is reborrowed by reference `y`. Borrowing with `x` creates loan `L1` while reborrowing with `y` creates loan `L2`. Since loan `L2` was created by reborrowing the place created in `L1`, we need to expire `L2` before expiring `L1`. From such dependencies, which we obtain from Polonius, we construct the basic structure of the loan-dependency graph, which we show in Figure 3.4b.

In Figure 3.4b, nodes *Start* and *End* are helper nodes indicating entry and exit points, respectively. Nodes marked with loan identifiers contain the operations needed to bring the capabilities into the state just before the loan was created. The edges between the nodes indicate the dependencies between the loans and are obtained from Polonius. Different paths

```
1 let mut a = 1;
2 let x = &mut a;      // L1
3 let y = &mut *x;     // L2
4 *y = 2;
```

(a) A simple borrowing example.



(b) A loan-dependency graph for expiring reference `y`.

Figure 3.4: A simple borrowing example and the corresponding loan-dependency graph.

through the graph can mean either that some loans are alternatives or that they happened in parallel. For example, Figure 3.5 shows the conditional borrowing example from earlier and the corresponding loan-dependency graph when reference `x` expires. In this case, different paths through the graph mean that loans `L1` and `L2` are alternatives: only one could have happened. Figure 3.6 shows that in code that calls reborrowing functions, different paths can also mean that loans occurred in parallel. In this example, loans `L1` and `L2` are not alternatives; they are alive at the same time. We show how we resolve this ambiguity when we present our solution to challenge *C2*.

```
1 let x;
2 if random_choice() {
3     x = &mut a; // L1
4 } else {
5     x = &mut b; // L2
6 }
7 *x = 42;
```

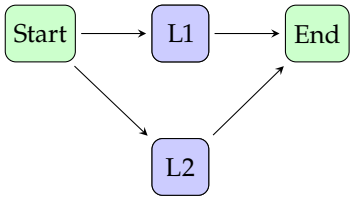


Figure 3.5: A conditional borrowing example and a corresponding loan-dependency graph when reference `x` expires. In this case, different paths through the loan-dependency graph mean alternatives.

```
1 let x = &mut a; // L1
2 let y = &mut b; // L2
3 let z = borrow_both(x, y); // L3
```

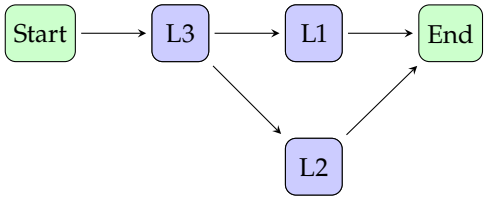
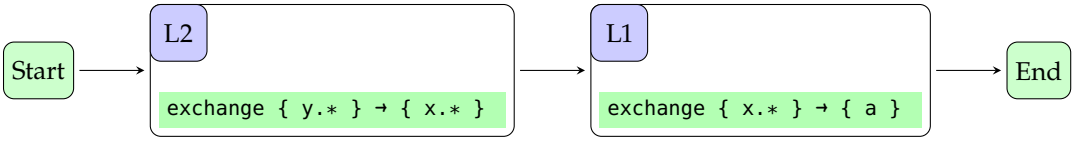


Figure 3.6: An example with a reborrowing function call demonstrating that different paths through the loan-dependency graph can also mean parallelism.

Figure 3.4b shows the basic structure of the loan-dependency graph. As mentioned earlier, after expiring a loan in the graph, the capabilities should be returned to the state they were just before the creation of the loan. In our example, before executing borrow `let y = &mut *x;` that created loan `L2`, the capability was associated with place `x.*`. The borrow statement transferred the capability from `x.*` to `y.*`. Therefore, to undo the borrow statement's effect, we transfer the capability back from `y.*` to `x.*` using EC operation `exchange { y.* } → { x.* }`. We have the necessary exchange capability for executing this exchange operation because executing the borrow statement produced it. We expire loan `L1` created by borrow statement `let x = &mut a;` in a similar way by executing EC operation `exchange { x.* } → { a }`. If we add the two `exchange` operations to our graph, we obtain a graph that completely captures how capabilities get transferred when the reference `y` expires.

The graph is shown below.



As we mentioned in the introduction of this chapter, exchange capabilities are also returned by function calls that take references as arguments[4]. For example, the signature of the following function implies that the result could reborrow either of the two parameters.

```
fn reborrow_one_of(
    x1: &mut i32,
    x2: &mut i32,
) -> &mut i32;
```

This behaviour is captured by an exchange capability `{ result.* } → { x1.*, x2.* }` that is returned by this function to its caller. As we mentioned before, there are two ways to obtain an exchange capability: execute a borrow statement or derive it from other exchange capabilities. Typically, the exchange capabilities returned by functions are non-trivial and have to be derived. We prove the exchange capability `{ result.* } → { x1.*, x2.* }` by adding a `prove-exchange { result.* } → { x1.*, x2.* }  G` operation at the end of the function body. The third argument of this operation is `G`, a loan-dependency graph that justifies the capability transfer. We construct this graph by asking Polonius what happens when reference `result` expires.

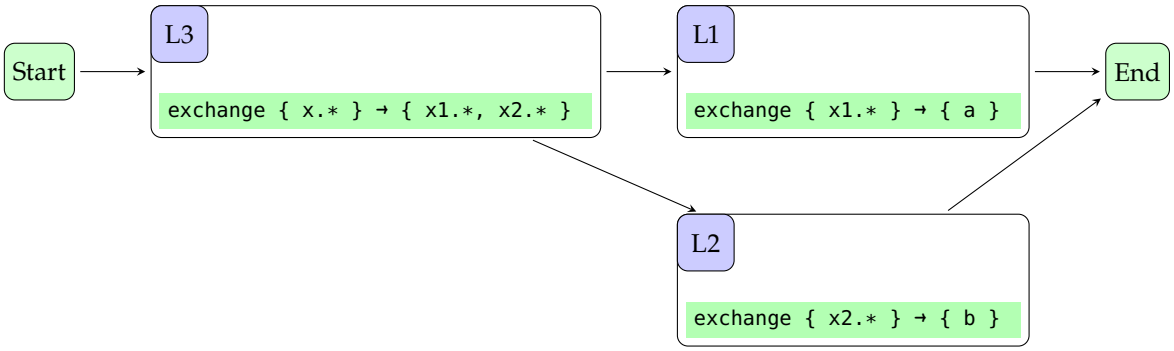
On the caller side, the exchange capability returned by a function is treated the same way as the ones returned by borrow statements. For example, the following snippet shows a possible caller of `reborrow_one_of` with explicit PCS information.

```
PCS: {a, b}
let x1 = &mut a;     // L1
PCS: {x1, b}
let x2 = &mut b;     // L2
PCS: {x1, x2}
let x = reborrow_one_of(x1, x2);     // L3
PCS: {x}
*x = 42;
expire-borrows ...
PCS: {a, b}
```

The loan-dependency graph used in `expire-borrows ...` is shown be-

4: Actually, a single function call can lead to more than a single loan and thus return more than one exchange capability. Our approach treats each loan separately.

low.



As we can see from the graph, the main difference between direct borrows and borrows via functions is that the former always produces an exchange capability between two singleton sets while the latter can be significantly more varied. For example, suppose the called function takes mutable reference arguments but does not return a mutable reference. In that case, the returned exchange capability will have an empty set on its left-hand side. Such capability can be applied for exchange immediately after the function call.

In the following, we iteratively refine the definition of the graph to address each of the challenges discussed in the previous section.

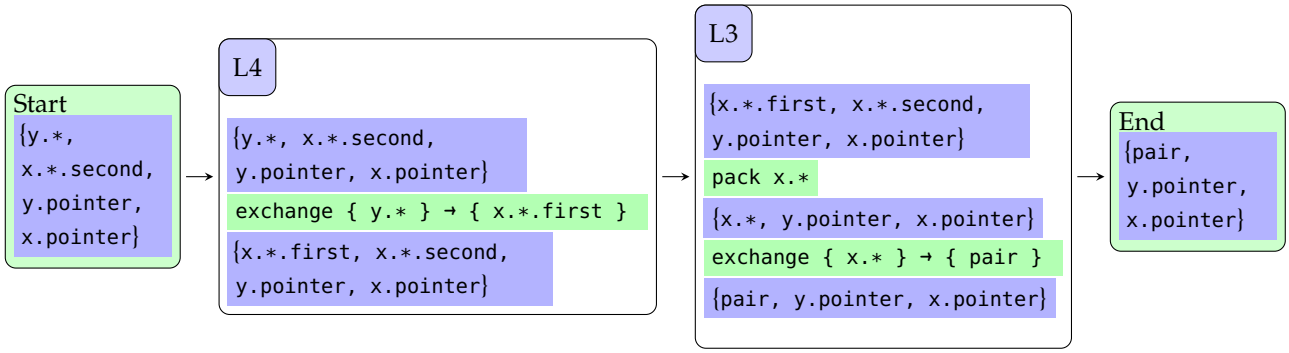
5: The algorithm is *almost* the same. As explained later, we need some adjustments to handle the next challenge.

Handling effects of PCS operations (*C1*). It turns out we can address this challenge using our PCS elaboration algorithm from Chapter 2 without adding any additional features[5]. However, we need to ensure that the assumptions made by the algorithm are upheld. Since the borrow checker is executed before our analyses, we know the program points at which we need to insert `expire-borrows` operations before we run our algorithm that computes PCS operations. As a result, we know the PCS before `expire-borrows`. This property enables us to run our PCS elaboration algorithm on the loan-dependency graph to compute the necessary PCS operations. In Chapter 2, we mentioned that the PCS elaboration algorithm assumes that each place capability can be modified independently without affecting any other capability. This property trivially holds for the Rust fragment without references because there is no aliasing. We can treat each place independently in Rust with mutable references, assuming two properties hold. First, only one of the aliased places has the capability associated with it, allowing the algorithm to use that syntactic place to manipulate the capability. This property is ensured by the Rust type system. Second, when a borrow expires and its capability is transferred from one place to another, the shape of the transferred capability is known to the PCS elaboration algorithm. We ensure this property by guaranteeing that when a borrow expires, the capability is returned in the same shape it was borrowed.

We used the snippet below to motivate the challenge.

```
1  PCS: {pair}
2  let x = &mut pair;       // L3
3  PCS: {x}
4  let y = &mut x.first;    // L4
5  PCS: {y, x.*.second, x.pointer}
6  *y = 5;
7  PCS: {y.*, x.*.second, y.pointer, x.pointer}
8  expire-borrows ...
```

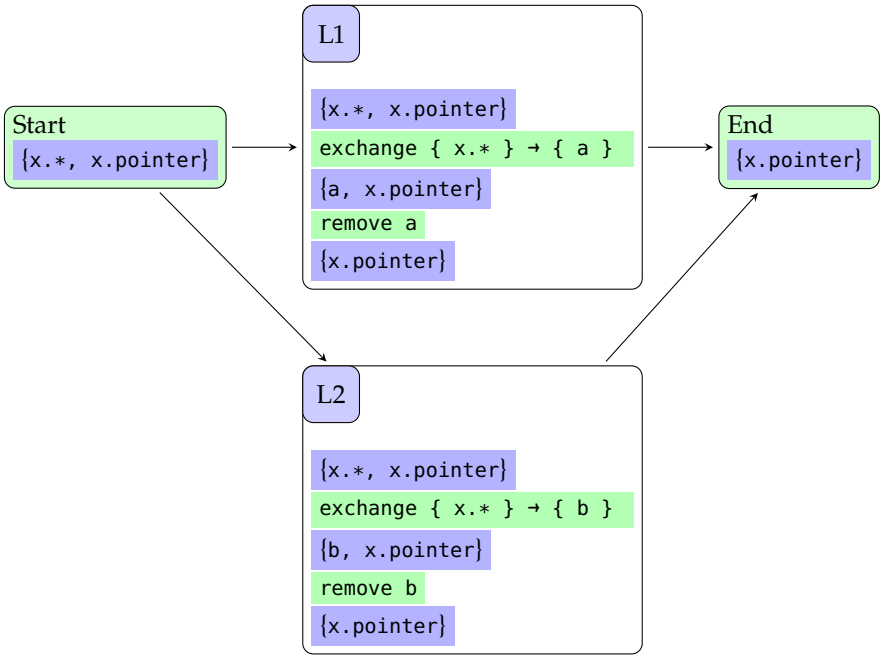
After executing the PCS elaboration algorithm on the graph for this snippet, we would get the result shown in the following diagram. In this diagram, each node contains not only the EC operations but also the PCS operations needed to get the capabilities into the required shape.



Handling conditional control flow (*C2*). The motivating example for this challenge is repeated in the following snippet with PCS information and PCS operations.

```
1  PCS: {a, b}
2  let x: &'lx i32;
3  if random_choice() {
4      PCS: {a, b}
5      x = &mut a; // L1
6      PCS: {x, b}
7      remove b
8      PCS: {x}
9  } else {
10     PCS: {a, b}
11     x = &mut b; // L2
12     PCS: {a, x}
13     remove a
14     PCS: {x}
15 }
16 PCS: {x}
17 *x = 4;
18 PCS: {x.*, x.pointer}
19 expire-borrows ...
20 PCS: {x.pointer, a, b}
```

If we constructed the loan-dependency graph based on how we described so far, it leads to the graph below.



This graph shows two limitations of our approach so far. First, our PCS elaboration algorithm drops the capabilities `a` and `b` when joining the states of nodes `L1` and `L2`. Second, as we mentioned earlier, it is unclear from the graph whether `L1` and `L2` are alternatives or should they happen in parallel.

The first problem of dropping the capabilities happens because the PCS elaboration algorithm we presented in the previous chapter does not distinguish between capabilities that are missing because a place was moved out and capabilities that are missing because a place was borrowed. In the former case, dropping the capability when merging the branches is okay because it cannot be recovered. In the latter case, as our example shows, the capability can be recovered when the borrow expires. We distinguish the two cases by assigning them different capabilities:

Definition 3.3.1 (Capability (first version)) *A capability is either* exclusive *(E) or* borrowed *(B)*[6].

We use the new definition of capability to refine the definition of PCS:

Definition 3.3.2 (Place Capability Sets (final version)) *A* place capability set *(PCS) is a partial map from places to capabilities.*

With the two capabilities defined, we change the semantics of the borrow statement to remember that the borrowed place is borrowed by producing the *borrowed* capability, as shown in the following snippet.

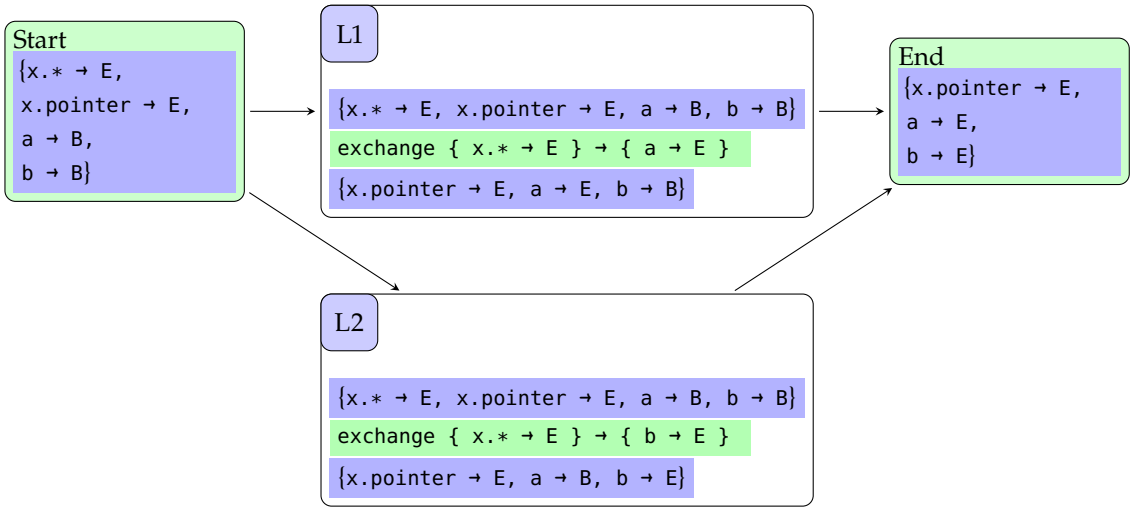
```
1  PCS: {a → E}
2  let x = &mut a;
3  PCS: {b → E, a → B}
```

6: We present here slightly simplified definitions. The actual implementation in Prusti also remembers which loan borrowed the capability to ensure that different borrows are not mixed up.

We update the existing definitions of PCS operations in an obvious way. Unpacking place `p` with capability c produces place capabilities for fields with the same capability c. Packing a place `p` is only allowed if places for all fields have the same capability c, which is also the capability of packed place `p`. The only change to the algorithm that is more interesting is the merging of PCSs, which now works slightly differently when performed on code where the capabilities flow forward and when performed on the loan-dependency graph where the capabilities flow backward. In the forward flow, merging *exclusive* and *borrowed* results in *borrowed*. For example, in the following snippet that shows an updated version of our example, after the if statement, we have capability *borrowed* for both `a` and `b`.

```
1  PCS: {a → E, b → E}
2  let x: &'lx i32;
3  if random_choice() {
4      PCS: {a → E, b → E}
5      x = &mut a; // L1
6      PCS: {x → E, a → B, b → E}
7  } else {
8      PCS: {a → E, b → E}
9      x = &mut b; // L2
10     PCS: {x → E, a → E, b → B}
11 }
12 PCS: {x → E, a → B, b → B}
13 *x = 4;
14 PCS: {x.* → E, x.pointer → E, a → B, b → B}
15 expire-borrows ...
16 PCS: {x.pointer → E, a → E, b → E}
```

Importantly, as can be seen from line 14, `a` and `b` are marked as borrowed places in the PCS before the `expire-borrows` operation. Using this state as the initial one for elaborating PCSs, we get the graph below. Since when elaborating PCSs in the loan dependency graph the capabilities flow backwards (we are restoring the borrowed capabilities), merging *exclusive* and *borrowed* results in *exclusive*.

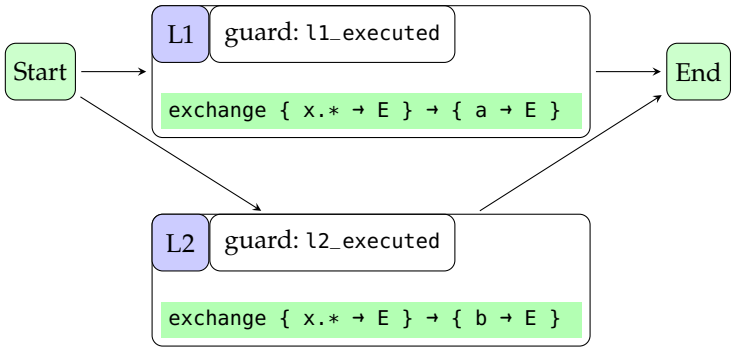


The root cause of the second limitation of the loan-dependency graph not capturing that nodes `L1` and `L2` are mutually exclusive is that the current

version of the loan-dependency graph is control flow independent. We make it dependent on control flow by adding guards to each node: the operations in a node are only executed if the guard condition evaluates to true. Since the verifier knows precise information about which path through the program was taken, we use ghost variables to mark whether a loan happened and needs to be returned. More specifically, for each loan, we create a boolean ghost variable initialised to false at the beginning of the function and set to true when the corresponding loan creation statement is executed. For example, our motivating snippet would be expanded as follows.

```
1  l1_executed = false;
2  l2_executed = false;
3  let x: &'lx i32;
4  if random_choice() {
5      x = &mut a; // L1
6      l1_executed = true;
7  } else {
8      x = &mut b; // L2
9      l2_executed = true;
10 }
11 *x = 4;
```

Ghost variable `l1_executed` tracks whether `L1` was executed. Similarly, ghost variable `l2_executed` tracks whether `L2` was executed. With these ghost variables added, we can make the execution of nodes guarded on whether the corresponding loans were created. As shown in the graph below, we use the ghost variables as the guards of the nodes.



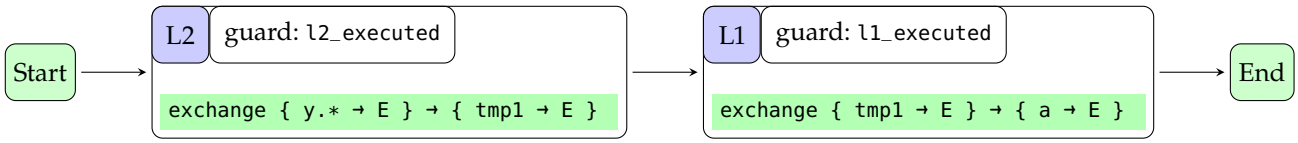
Handling reference reassignments (*C3*). The following snippet repeats our motivating example for this challenge.

```
1  let mut x = &mut a;      // L1
2  l1_executed = true;
3  let y = &mut *x;         // L2
4  l2_executed = true;
5  x = &mut b;              // L3
6  l3_executed = true;
7  *y = 5;
8  *x = 4;
```

The key problem in this challenge is modelling the expiration of `L2` that occurs when reference `y` expires. If we followed the approach we

discussed so far, when expiring `L2`, we should transfer the capability from `y.*` to `x.*`. However, we cannot do that because `x` was reassigned and `x.*` means a different memory location compared to the one when `y` was created. The key observation that leads us to our solution is that `L2` is guaranteed not to be the "final stop". For instance, in our example, when reference `y` expires, both `L2` and `L1` expire: expiring `L2` transfers the capability from `y.*` to `x.*` and expiring `L1` transfers the capability from `x.*` to `a`. `L2` is guaranteed not to be the final stop because when `x` is reassigned, Polonius breaks the relation between `x` and `L1` that is keeping `L1` alive. `L1` is not expired only because it is kept alive by `y`. Therefore, when `y` expires, both `L1` and `L2` expire at the same time.

Since `L2` is guaranteed not to be the final stop, `x.*` is only a temporary place where the capability is stored during the transfer. Therefore, we can avoid the problem of `x.*` having a different meaning by using a freshly created temporary place just for the transfer. The loan-dependency graph constructed at the expiration of reference `y` is shown below with `tmp1` used instead of the old place `x.*`. Note that `L3` expires at a different point and is, therefore, not shown in the graph.



Final loan-dependency graph definition. With all challenges covered, we can give the final definition of the loan-dependency graph:

Definition 3.3.3 (Loan-Dependency Graph) *A loan-dependency graph is a graph $G = (V, E)$ where:*

- *V is a set of nodes. Each node v is the* Start *node, the* End *node, or a loan node. A loan node is a triple (L, G, O) where L is a loan, G is a boolean variable indicating whether the loan was executed and guarding the execution of operations in this node, and O is a sequence of PCS and EC operations.*
- *E is a set of directed edges between nodes in V. An edge (l_1, l_2) between loan nodes exists in E if and only if loan l_2 directly depends on loan l_1 based on Polonius information. An edge* (Start, l) *exists in E iff loan l does not depend on any loans. An edge* (l,End) *exists in E iff no loans depend on l.*

3.4 Extending the Core Proof to Support Mutable Borrows

In this section, we show how we model the introduced new concepts in Viper to extend our core proof with support for mutable references. We first discuss in Subsection 3.4.1 how we model mutable reference capabilities as Viper predicates. Then, in Subsection 3.4.2, we present magic wands, a connective common in permission logics, which is the key primitive used for our model of exchange capabilities. In Subsection 3.4.3,

we present our model of exchange capabilities based on magic wands. We finish by presenting potential optimisations of the encoding in Subsection 3.4.4.

3.4.1 Modelling Capabilities of Mutable Reference

As already discussed, if place `x` is a mutable reference, its capability includes two capabilities inside it: a capability to pointer `x.pointer` and a capability to the referenced target `x.*`. Therefore, we encode mutable reference type `&mut T` as the following predicate where the permission to field `acc(self.pointer, write)` corresponds to the pointer capability and the permission to predicate instance `acc(T(self.pointer), write)` corresponds to the capability to the target.

```
predicate UniqueRef_T(self: Ref) {
    acc(self.pointer, write) &&
    acc(T(self.pointer), write)
}
```

Similarly to the encoding of the `Box<T>` type, the type of the `pointer` field is `Ref`, which we use as an abstract address. With this definition of a reference, we can model borrow and reborrow statements in Viper as simple assignments to the `pointer` field. For example, the following snippet creates a mutable reference `x` and assigns 1 to its target.

```
let x = &mut a;
*x = 1;
```

This snippet is encoded in Viper as follows (we omit modelling of the exchange capability for now; we discuss it later).

```
// let x = &mut a;
inhale acc(x.pointer, write)
x.pointer := a;
fold acc(UniqueRef_i32(x), write)

// *x = 1;
unfold acc(UniqueRef_i32(x), write)
unfold acc(i32(a), write)
x.pointer.val_i32 := 1;
fold acc(i32(a), write)
fold acc(UniqueRef_i32(x), write)
```

3.4.2 Magic Wand Connective

A magic wand $A \mathrel{-\!\!*} B$ expresses that B can be obtained by giving away both A and $A \mathrel{-\!\!*} B$. The wand itself is a resource because it carries the parts of B that are not in A. For example, the magic wand shown in the following snippet must contain `acc(r.second, write)` inside it; otherwise, applying it would forge permissions, which is unsound.

```
acc(r.first, write) --*
    acc(r.first, write) && acc(r.second, write)
```

In Viper, the magic wand can be obtained by using a `package` statement. Packaging the wand requires proving that the right-hand side of the wand can be obtained from its left-hand side. If any permissions are missing, the verifier will pull them into the wand from the surrounding context. For example, packaging the wand from the previous snippet will pull `acc(r.second, write)` into the wand. If the surrounding context does not contain the pulled permission, the packaging of the wand fails.

```
inhale acc(r.second, write)
package acc(r.first, write) --*
    acc(r.first, write) && acc(r.second, write) {
}
```

Packaging a magic wand may involve other ghost operations, such as folding a predicate or applying another magic wand. For example, if the permissions to fields `first` and `second` were grouped into predicate `Pair`, we could prove a variation of the magic wand from the previous snippet that has a `Pair` predicate instance on its right-hand side. As the following snippet shows, packaging such a magic wand requires folding the predicate inside the `package` statement.

```
inhale acc(r.second, write)
package acc(r.first, write) --* acc(Pair(r), write) {
    fold acc(Pair(r), write)
}
```

Once we have a magic wand, we can apply it by using Viper's `apply` statement, as shown in the following snippet.

```
apply acc(r.first, write) --* acc(Pair(r), write)
```

Applying the magic wand consumes the wand and the permissions on its left-hand side (in this case `acc(r.first, write)`) and produces the permissions on its right-hand side `acc(Pair(r), write)`.

It is important to note that Viper preserves precise information about values of the heap locations captured by the wand. Therefore, Viper can prove the two assert statements at the end of the following snippet.

```
inhale acc(x.f) && x.f == 1
inhale acc(x.g) && x.g == 2
package acc(x.f) --* acc(x.f) && acc(x.g)
apply acc(x.f) --* acc(x.f) && acc(x.g)
assert x.f == 1
assert x.g == 2
```

3.4.3 Modelling Exchange Capabilities

We already covered how we model a capability of a reference. We still need to show how we model exchange capabilities, borrow statements, and all newly introduced operations `exchange`, `prove-exchange`, `expire-borrows`.

The exchange capability `P1 ↠ P2` expresses that we can exchange capabilities `P1` together with the exchange capability itself into capabilities

`P2`. The magic wand connective is a natural way of modelling this capability. We model `P1 → P2` as magic wand $\lfloor P1 \rfloor \mathrel{-\!\!*} \lfloor P2 \rfloor$. $\lfloor P1 \rfloor$ and $\lfloor P2 \rfloor$ here are predicate instances corresponding to capabilities in `P1` and `P2` conjoined with a separating conjunction. We model that the function taking mutable references as arguments returns an exchange capability to the caller by adding the corresponding magic wand to the Viper method's postcondition.

```
1  let x: &'lx i32;
2  if random_choice() {
3      x = &mut a; // L1
4  } else {
5      x = &mut b; // L2
6  }
7  *x = 4;
8  // x expires
```

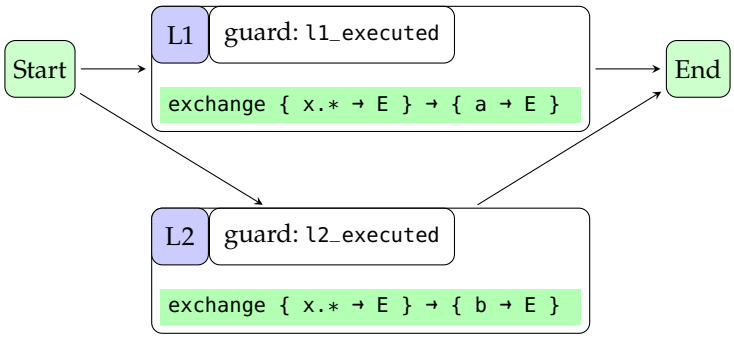


Figure 3.7: Conditional borrowing example repeated from Figure 3.3 and the corresponding loan-dependency graph when reference `x` expires.

The choice to model the exchange capability using a magic wand makes modelling of borrow statements and EC operations straightforward. We model an `exchange` operation using Viper's `apply` statement, which applies the corresponding magic wand. Similarly, we model creating an exchange capability in a borrow statement or `prove-exchange` operation by packaging the corresponding wand. Figure 3.7 repeats the conditional borrowing example from before together with the loan-dependency graph for the program point at which reference `x` expires. Figure 3.8 shows its simplified encoding in Viper[7]. To illustrate the encoding of all EC operations with a single example, we replaced the `expire-borrows` operation with an equivalent sequence of `prove-exchange` immediately followed by `exchange`.

As mentioned above, we encode a borrow statement `let x = &mut a;` as shown on lines 4–8. The borrow statement `let x = &mut a;` produces an exchange capability `{ x.* } → { a }`. We package the corresponding magic wand by adding the `package` statement immediately after the encoding of the borrow as shown on line 8. This `package` statement is guaranteed to succeed because the verifier knows that `x.pointer == a` and, therefore, the left-hand side is equal to the right-hand side.

The encoding of `prove-exchange` is more involved. We encode it as the packaging of the corresponding magic wand. However, in this case, the two sides of the magic wand might be different. Therefore, we need to provide the body of the `package` statement that shows how to derive the right-hand side of the magic wand from its left-hand side. We encode the body by encoding each node of the loan-dependency graph in the order that guarantees that nodes of the dependencies are executed before the dependent nodes. Since the nodes in the loan-dependency graph shown in Figure 3.7 are independent, we can encode them in arbitrary order. Each graph node is a sequence of statements that must be executed if the guard evaluates to true. Therefore, we encode each node as an if statement with the guard in the condition and the encoding of the

7: For example, we use a simplified syntax for magic wands: Viper requires both sides of the magic wand to be self-framing, which means we need to have permission to `x.pointer` on the left-hand side of the magic wand. In the implementation, we avoid this by storing the value of `x.pointer` in a temporary variable and using that variable instead.

```
1  l1_executed = false;
2  l2_executed = false;
3  if random_choice() {
4      // x = &mut a; // L1
5      inhale acc(x.pointer, write)
6      x.pointer := a;
7      fold acc(UniqueRef_i32(x), write)
8      package acc(i32(x.pointer), write) --* acc(i32(a), write) {}
9      l1_executed = true;
10 } else {
11     // x = &mut b; // L2
12     inhale acc(x.pointer, write)
13     x.pointer := b;
14     fold acc(UniqueRef_i32(x), write)
15     package acc(i32(x.pointer), write) --* acc(i32(b), write) {}
16     l2_executed = true;
17 }
18
19 // *x = 4;
20 unfold acc(UniqueRef_i32(x), write)
21 unfold acc(i32(x.pointer), write)
22 x.pointer.val_i32 := 4;
23 fold acc(i32(x.pointer), write)
24
25 // prove-exchange { x.* → E } → { a → E, b → E }
26 package acc(i32(x.pointer), write) --* acc(i32(a), write) && acc(i32(b), write) // R
27 {
28     if (l1_executed) {
29         // exchange { x.* → E } → { a → E }
30         apply acc(i32(x.pointer), write) --* acc(i32(a), write)
31         // recover b
32     }
33     if (l2_executed) {
34         // exchange { x.* → E } → { b → E }
35         apply acc(i32(x.pointer), write) --* acc(i32(b), write)
36         // recover a
37     }
38 }
39
40 // exchange  { x.* → E } → { a → E, b → E }
41 apply acc(i32(x.pointer), write) --* acc(i32(a), write) && acc(i32(b), write) // R
```

Figure 3.8: A simplified encoding of Rust example from Figure 3.7. We replaced `expire-borrows` with `prove-exchange` followed by `exchange` to be able to illustrate all three operations with a single example. The encoding of `expire-borrows` would be exactly the same as of `prove-exchange` (lines 26–38), except without line 26. Also, if `expire-borrows` was used, operation `exchange` would not be needed (line 41).

statements in the then branch. Lines 28–32 show the encoding of the node for loan `L1` while lines 33–37 show the encoding of the node for loan `L2`. Each node has an `exchange`, which, as mentioned above, we encode using an `apply` statement as shown on lines 30 and 35. If we used `expire-borrows` directly instead of replacing it with `prove-exchange` followed by `exchange`, the encoding would be almost the same: it would not have lines marked with comment "R" (lines 26 and 41).

3.4.4 Optimising the Encoding

In the encoding we presented, we consistently use magic wands for modelling all borrows. However, most magic wands are unnecessary and can be removed, leading to a more straightforward encoding that is easier to check for the backend verifier. We present two cases where we can avoid creating magic wands.

When we discussed packing a magic wand for the borrow statement `let x = &mut a;`, we observed that it will always succeed because the left-hand side of the magic wand is equal to its right-hand side. In other words, in our Viper model, the magic wand's footprint is empty: the wand does not transfer any permissions. This difference between Rust, where the capabilities are transferred, and Viper, where permissions are not transferred, comes from the different treatment of locations. In Rust, capabilities are identified by *syntactic places* that cannot be aliased. In Viper, permissions are associated with an *object identity*, which can be syntactically referred to in different ways due to aliasing. We can exploit this additional flexibility of our Viper model by avoiding creating magic wands for loans created with borrow statements. However, we need to distinguish between loans created by borrow statements and function calls. We, therefore, introduce a new exchange capability operation `transfer { p1 } → { p2 }` that we use instead of regular `exchange { p1 } → { p2 }` operation for borrow statements (we continue using `exchange` for function calls). In our optimised Viper encoding, we encode `transfer` as a no-op. Since by encoding `transfer` as a no-op, we omit the applying of a magic wand, we also change the encoding of a borrow statement to skip the packaging of the corresponding magic wand.

The second place where we can avoid creating magic wands is functions that take references as arguments but do not return references. The following snippet illustrates this case with a function that sets the value of the first element of a pair.

```
1  impl Pair {
2      fn set_first(&mut self, new_value: i32) {
3          self.first = new_value;
4      }
5  }
```

Since this function does not return any references, the exchange capability for this function (`{  } → { self.* }`) has an empty left-hand side and, therefore, is encoded to the magic wand shown in the following snippet.

```
1  true --* Pair(old(self.pointer))
```

This magic wand is returned to the caller, and since the caller can trivially satisfy its left-hand side, it would immediately apply it, obtaining the resources on its right-hand side. In this case, we can simplify the encoding by returning the permissions to the caller by putting them directly into the postcondition instead of putting the permissions on the right-hand side of the magic wand. The following snippet shows a simplified encoding of the `set_first` function.

```
method set_first(self: Ref, new_value: Ref)
{
    // Inhale precondition.
    inhale acc(UniqueRef_Pair(self), write)
    inhale acc(i32(new_value), write)
    label pre   // Mark the precondition state.

    unfold acc(UniqueRef_Pair(self), write)
    unfold acc(Pair(self.pointer), write)
    unfold acc(i32(self.pointer.first), write)
    unfold acc(i32(new_value), write)
    self.pointer.first.val_i32 := new_value.val_i32;

    // Expire borrows and reconstruct the permission to
    // the target of the borrow.
    fold acc(i32(self.pointer.first), write)
    fold acc(Pair(self.pointer), write)

    // Exhale postcondition.
    exhale acc(Pair(old[pre](self.pointer)), write)
}
```

4 Shared Borrows

In the previous chapter, we presented how to generate a core proof for Rust programs that use mutable references. Mutable references require exclusive access to their target. In this chapter, we focus on shared references that (as the name implies) allow aliases but do not allow mutating the target[1]. The following snippet illustrates the use of shared references. The owned value `a` is shared borrowed by `x` and, without ending it, borrowed again by `y`. Later, reference `z` is created by assigning `x` to it.

1: There are also so-called *internally mutable* types in Rust (for example, `Mutex`) that allow mutation via a shared reference. We leave supporting such types for future work.

```
1  let x = &a;
2  let y = &a;
3  let z = x;
4  // a = 5;  // Illegal!
5  assert!(a == *x);
6  assert!(*z == *y);
```

The snippet shows three essential aspects of how shared borrows differ from mutable ones. First, the original place `a` can still be used for reading while the borrow `x` is active (lines 2 and 5). Mutable borrows completely block the borrowed place until the borrowing reference expires. Second, on line 3, the capability is *copied* from `x` to `z`: both places can be used for reading as illustrated with the `assert!(...)` statements below. If `x` were a mutable reference, this statement would cause a reborrow that blocks `x` until `z` expires. Third, none of the aliases can be used for writing. Therefore, the memory is guaranteed to be *immutable* as long as at least one of the shared references is still alive.

Standard techniques in permission logics for giving access to a resource to multiple readers are based on splitting the exclusive capability into capabilities that give only read access. For example, we could adapt fractional permissions [23] to PCSs by allowing one to split *exclusive* capabilities into fractions: a full fraction $f = 1.0$ of an *exclusive* capability would allow read and write access while any positive fraction ($0 < f < 1.0$) would allow only read access. The downside of this approach is that regaining the full capability requires collecting all parts of the split capability and combining them back, which would require reconstructing backward capability flow for shared borrows. This reconstruction is more complicated than for mutable borrows (because shared references can be aliased) and unnecessary (shared borrows do not allow mutation, so there are no changes to propagate backwards). The vital information for verification is for how long a borrowed place is guaranteed to be immutable and hence retain the same value. As discussed in Section 3.1, this information is precisely what the borrow checker computes.

[23]: Boyland (2003), 'Checking Interference with Fractional Permissions'

The remainder of the chapter is structured as follows. In Section 4.1, we introduce a new place capability kind *shared* that guarantees that the value stored at that place is immutable and show how we can obtain the flow

of these capabilities from the information available in the borrow checker. Choosing a capability model closer to the borrow checker leaves an open problem of how to map *shared* capabilities to Viper that uses fractional permissions. We show how we address this challenge in Section 4.2.

4.1 Shared Capabilities

To model shared references and shared borrowed places, we introduce a new *shared* capability. *shared* is a temporarily duplicable capability that guarantees that a place is immutable. We start by showing how we update our capability model and then explain how we elaborate PCSs for programs that use shared references.

In the previous chapter, we updated our definition of PCS to be a partial map from places to *exclusive* or *borrowed* capabilities. We extend the definition of capabilities also to include *shared*:

Definition 4.1.1 (Capability (final version)) *A capability is either* exclusive *(E)*, borrowed *(B)*, *or* shared *(S)*.

Existing PCS operations are mostly unaffected by the changed definition since we updated them to be generic over capabilities in the previous chapter. The only exception is `unpack p` and `pack p` operations when `p` is a shared reference. Similarly to mutable references, unpacking a shared reference `p` produces two capabilities `p.pointer` and `p.*`. However, unlike with mutable references where the capabilities of all places always match, with shared references, the target capability `p.*` is *shared* regardless whether the capability for `p` was *exclusive* or *shared*.

The following snippet shows what PCS information the PCS elaboration algorithm computes for the example in the introduction of this chapter (we show all places unpacked and omit `pointer` capabilities to make the code more readable).

```
1  PCS: {a → E}
2  let x = &a;
3  PCS: {a → S, x.* → S}
4  let y = &a;
5  PCS: {a → S, x.* → S, y.* → S}
6  let z = x;
7  PCS: {a → S, x.* → S, y.* → S, z.* → S}
8  // a = 5;  // Illegal!
9  PCS: {a → S, x.* → S, y.* → S, z.* → S}
10 assert!(a == *x);
11 PCS: {a → S, y.* → S, z.* → S}
12 assert!(*z == *y);
13 PCS: {a → E}
```

As can be seen from the example, the first borrow of `a` (line 2) *downgrades* the capability of the borrowed place from *exclusive* to *shared* and *copies* the new capability to the reference. However, the second borrow statement on line 4 only copies the capability from `a` to `y`. The capability is also copied by assigning one shared reference to another on line 6. When a reference expires, it *loses* its capability; for example, reference `x` lost its

capability after line 10. When all references to a expire after line 12, its capability is *upgraded* from *shared* to *exclusive*.

Downgrade and upgrade of a capability and expiration of a shared reference are not explicitly visible in the code. Therefore, we create three new PCS operations: `downgrade`, `upgrade`, and `expire-shared`. `downgrade p` consumes an *exclusive* capability for place `p` and produces a *shared* capability for it. The PCS elaboration algorithm emits this operation in two cases: before the borrow statement that borrows a place that is not yet shared-borrowed and when merging the PCSs of incoming branches if place `p` is *exclusive* in one branch and *shared* in another branch, the *exclusive* capability is downgraded. `upgrade p` consumes a *shared* capability for place `p` and produces *exclusive*. The PCS elaboration algorithm emits this operation after all shared references to place `p` expire. The last operation `expire-shared p.*` consumes a *shared* capability and is emitted when a shared reference `p` expires. We rely on the information from the compiler to deduce when all shared references have expired and all copies of the capability were collected[2]. An important advantage of using duplicable capabilities that can be associated with syntactic places is that we do not make any changes to the parts of the PCS elaboration algorithm that compute `pack` and `unpack` operations. The snippet below shows our example with added PCS operations.

2: The encoding of shared references is the only case where we are not fully re-proving the memory safety

```
1  PCS: {a → E}
2  downgrade a
3  PCS: {a → S}
4  let x = &a;
5  PCS: {a → S, x.* → S}
6  let y = &a;
7  PCS: {a → S, x.* → S, y.* → S}
8  let z = x;
9  PCS: {a → S, x.* → S, y.* → S, z.* → S}
10 // a = 5;  // Illegal!
11 PCS: {a → S, x.* → S, y.* → S, z.* → S}
12 assert!(a == *x);
13 PCS: {a → S, x.* → S, y.* → S, z.* → S}
14 expire-shared *x
15 PCS: {a → S, y.* → S, z.* → S}
16 assert!(*z == *y);
17 PCS: {a → S, y.* → S, z.* → S}
18 expire-shared *y
19 PCS: {a → S, z.* → S}
20 expire-shared *z
21 PCS: {a → S}
22 upgrade a
23 PCS: {a → E}
```

An important case not captured by our example is passing shared references to called functions, as shown in the following snippet.

```
1  fn read_caller(x: &i32) {
2      read(x);
3  }
```

In this case, the function `read` receives a *copy* of the reference `x`. Therefore, after the call, we still have the capability to `x`, which enables us to conclude that `x` was not modified by the call to `read`.

The presented system maintains the following invariant relating *shared* and *exclusive* capabilities: if place `p` is not borrowed, place `p` has *exclusive* capability; if place `p` is shared borrowed, place `p` and each of its shared borrows have a *shared* capability.

4.2 Extending the Core Proof to Support Shared Borrows

Similarly to Rust, Viper guarantees that a single writer can access a memory location with exclusive read-write access or multiple readers can only read that location. Viper achieves this by supporting fractional permissions [23], which enable splitting and recombining accessibility predicates. For example, the following snippet shows a typical pattern of calling a method *callee* and giving it only half of the full permission that allows the callee only to read.

[23]: Boyland (2003), 'Checking Interference with Fractional Permissions'

```
method callee(r: Ref)
    requires acc(r.val_i32, write/2)
    ensures acc(r.val_i32, write/2)

method caller(r: Ref)
    requires acc(r.val_i32, write) && r.val_i32 == 42
    ensures acc(r.val_i32, write) && r.val_i32 == 42
{
    callee(r)
}
```

Since the caller retains the other half of the permission, they know that `r.val_i32` could not have been changed and, therefore, can prove the corresponding postcondition. This pattern can be generalized by using symbolic permission amounts [25] as shown in the following snippet.

[25]: Heule et al. (2013), 'Abstract Read Permissions: Fractional Permissions without the Fractions'

```
method callee(r: Ref, p: Perm)
    requires none < p
    requires acc(r.val_i32, p)
    ensures acc(r.val_i32, p)

method caller(r: Ref, p: Perm)
    requires none < p
    requires acc(r.val_i32, p)
    ensures acc(r.val_i32, p)
{
    callee(r, p/2)
}
```

The parameter `p` is a symbolic permission amount that is required to be positive, which ensures that the methods have at least read access to `r.val_i32`. Since the permission amount is a real number, it can be split an arbitrary number of times (for example, by dividing each time by 2,

as shown in the snippet), enabling a natural way of supporting recursive calls where each gets some non-zero permission providing read access. Fractional and symbolic permissions are a natural fit for code where read permission needs to be temporarily given for well-nested regions of code, such as recursive function calls. However, in Rust, the lifetimes of shared references can overlap, as shown in the following snippet (the comments show the lifetime of each reference) and are, therefore, challenging to model using these models.

```
let x = &pair.first;         // x
pair.second = 1;             // x
let y = &pair;               // x y
assert!(x == &y.first);      // x y
let z = &pair.second;        //   y z
assert!(z == &y.second);     //   y z
pair.first = 2;              //     z
assert!(z == &pair.second); //     z
```

Our solution to this challenge exploits the fact that the code is borrow-checked and relies on PCS operations for shared borrows being correct. Instead of mapping each *shared* capability to a potentially different permission amount as it is typically done when using symbolic permissions, we use the same underspecified permission amount for all *shared* capabilities. We define a global permission-typed variable `read_perm` that is constrained to have some value strictly between no and full permission. We use this variable as a permission amount for encoding places with *shared* capability.

We model the new PCS operations with Viper `inhale` and `exhale` statements. We defined `downgrade p` as consuming an *exclusive* capability for place `p` and producing a *shared* capability for it. In Viper, this operation corresponds to exhaling `write` and inhaling `read_perm` permission amount to the predicate instance that corresponds to place `p`, which can be simplified to exhaling `write - read_perm` permission. Similarly, we model `upgrade p` as inhaling `write - read_perm` permission to the predicate instance corresponding to `p`. The last newly defined operation `expire-shared p` only consumes the capability; therefore, we model it as an exhale of permission `read_perm`. Similarly, when a *shared* capability is duplicated by a borrow, copy assignment, or a function call, we inhale `read_perm` permission to the predicate instance that corresponds to the place that got the capability. Forging permissions when the capability is duplicated may look suspicious, but it is sound because we never constrain `read_perm` from below with any non-zero value. Therefore, no matter how many copies of `read_perm` the verifier has, it will not be able to conclude that the sum reaches the full permission amount.

Specifications for Functional Properties

5

Up to now, we have focused on the core proof and have already shown how we model basic functional specifications. However, the presented features are not enough to verify the functional correctness of a data structure like the generic linked list shown in Figure 5.1. For example, a programmer may want to specify how the following snippet in which an element at index 4 is mutably borrowed and assigned 42 affects the state of the linked list.

```
1 let r = list.index_mut(4);
2 *r = 42;
```

To specify how this snippet affects the state of `list`, we have to solve three challenges. First, the `value` field (line 2) that stores the value of the node and the `next` field (line 3) that is an optional owned pointer to the next node are private. Therefore, if we want a verification approach that respects information hiding and has all the benefits of it, the specifications visible to the client should not mention private fields. Second, since the linked list is a recursive data structure, the statement "the element at index 4" is also defined recursively. The specification approach we presented in Subsection 2.4.2 does not support talking about recursive definitions and needs to be extended. Third, the reference returned by `index_mut` is used to modify the internal values of `list` after the call. As a result, the standard approach of using the postcondition for specifying how the function affects the state is not applicable here because the postcondition is evaluated in the state immediately after the function while `r` can still be used for modification and thus invalidate the postcondition.

In the remainder of this chapter, we present specification features that address the three challenges mentioned above. We start by presenting *pure functions* in Section 5.1, which enable us to specify recursive data structures and to respect information hiding. The presented version of functions has an important limitation: they can return only Viper primitive values. In Section 5.2, we show how we use snapshots [21] to overcome this limitation. We finish by presenting in Section 5.3 *pledges*, which enable specifying functional behaviour of reborrows in a modular way.

[21]: Reynolds (2002), 'Separation Logic: A Logic for Shared Mutable Data Structures'

```
pub struct Node<T> {
    value: T,
    next: Option<Box<Node<T>>>,
}
impl<T> Node<T> {
    #[pure]
    pub fn len(&self) -> usize {
        match self.next {
            Some(box ref node) => {
                1 + node.len()
            }
            None => {
                1
            }
        }
    }
    #[pure]
    #[requires(0 <= index && index < self.len())]
    pub fn index(&self, index: usize) -> &T {
        if index == 0 {
            &self.value
        } else {
            match self.next {
                Some(box ref node) => {
                    node.index(index - 1)
                }
                None => {
                    unreachable!()
                }
            }
        }
    }
    #[requires(0 <= index && index < self.len())]
    #[ensures(result === old(self.index(index)))]
    #[after_expiry(result =>
        self.len() == old(self.len()) &&
        forall(|i: usize|
            0 <= i && i < self.len() && i != index ==>
            self.index(i) === old(self.index(i))
        ) &&
        self.index(index) === before_expiry(result)
    )]
    pub fn index_mut(&mut self, index: usize) -> &mut T {
        if index == 0 {
            &mut self.value
        } else {
            match self.next {
                Some(box ref mut node) => {
                    node.index_mut(index - 1)
                }
                None => {
                    unreachable!()
                }
            }
        }
    }
}
```

Figure 5.1: A linked list with slightly simplified functional specifications, which we discuss in the sections of this chapter. We verify this example with overflow checking disabled.

5.1 Pure Functions

A *pure function* is a Rust function that can be used in specifications. A function is marked as pure by adding an annotation `#[pure]` as shown on lines 6 and 17 in Figure 5.1. Pure functions are instrumental in verifying realistic examples because they enable expressing recursive properties and hiding implementation details. For example, by using the pure functions `len` and `index`, the client could specify that a linked list is sorted without knowing how the list is implemented, as shown in the following snippet.

```
1  #[requires(
2      forall(i: usize, j: usize
3          0 <= i && i <= j && j < list.len() ==>
4          list.index(i) <= list.index(j)
5  ))]
6  fn require_sorted(list: Node) { /* ... */ }
```

The requirements for pure functions are the same as for Rust expressions used in specifications: they have to be deterministic and side-effect-free. We ensure these two properties by checking that pure function parameter types implement the `Copy` trait, that the result is a primitive type (an integer or a bool)[1], and that pure functions call only pure functions. Implementing the `Copy` trait for a type means that the value of that type is entirely captured by the memory bit pattern and, therefore, can be copied. We currently also require that expressions used in specifications and pure functions do not contain loops, but this requirement could be lifted by automatically rewriting the loops into recursion.

1: The limitation to use a primitive type as a function result comes from our encoding to Viper presented in this section. We show how we lift this restriction in Section 5.2.

We model pure functions using Viper's *heap-dependent functions*. The following snippet shows a predicate `Segment` that defines a segment with `start` and `end` points and a heap-dependent function `segment_length` that computes the length of the segment.

```
1  field start: Int
2  field end: Int
3  predicate Segment(this: Ref) {
4      acc(this.start) && acc(this.end) &&
5      this.start <= this.end
6  }
7  function segment_length(this: Ref): Int
8      requires acc(Segment(this, read_perm))
9      ensures result >= 0
10 {
11     unfolding acc(Segment(this, read_perm)) in
12     this.end - this.start
13 }
```

The function computes the length of the segment by subtracting the start point from the end point on line 12. The body of the heap-dependent function is a heap-dependent expression for which all necessary permissions have to be provided by the function precondition (line 8). Since pure functions only read memory, any non-zero permission amount would work. We decided to use the same amount we used for shared references, which is stored in global variable `read_perm`. Postconditions

of heap-dependent functions cannot mention any permissions because heap-dependent functions are side-effect-free. Therefore, even though a heap-dependent function can return a Viper reference, the caller typically cannot use the returned reference because the function cannot return new or reshaped permissions. As a result, Viper functions typically return Viper primitive types such as integers and booleans.

```
function len(self: Ref): Int
    requires acc(Node(self, read_perm))
{
    unfolding acc(Node(self, read_perm)) in
    unfolding acc(Option_Box_Node(self.next, read_perm)) in
    self.next.discriminant == 0 ?
        // Some case
        1 + (
            unfolding acc(Option_Variant_Some_Box_Node(self.next), read_perm) in
            len(self.next.pointer)
        )
    :
        // None case
        1
}
```

Figure 5.2: An encoding of function `len` from Figure 5.1 in Viper. The encoding was slightly simplified for readability.

Encoding pure Rust functions into Viper heap-dependent functions is straightforward. Figure 5.2 shows a (slightly simplified for readability) encoding of the `len` function from Figure 5.1 in Viper. The preconditions and postconditions are encoded in the same way as for regular Rust functions, with the only difference that we use a built-in Viper `Int` as a return type to accommodate the fact that pure functions cannot return permissions. The function body is encoded as any other Rust expression used in the specification.

```
impl Vec {
    #[trusted]
    #[pure]
    pub fn len(&self) -> usize { /* ... */ }
    #[trusted]
    #[pure]
    #[requires(0 <= index && index < self.len())]
    pub fn index(&self, index: usize) -> i32 { /* ... */ }
}
```

Figure 5.3: Pure methods `len` and `index` that enable specifying the functional behaviour of `Vec`.

We presented how we encode pure functions for types that are verified. However, since pure functions enable the creation of an abstraction layer, they can also be used to provide trusted specifications for unverified types. For example, vector `Vec` from the Rust standard library is a safe abstraction implemented by using unsafe code and, therefore, cannot be verified by the techniques presented in this part of the thesis[2]. As mentioned in Subsection 2.3.3, we model safe abstractions as abstract Viper predicates. Without pure functions, we could generate a core proof for examples that use such data structures. However, we could not verify their functional correctness because the verifier knows nothing about instances of such predicates. Pure functions enable us to provide trusted

2: We present a technique capable of verifying `Vec` in Part III.

specifications for such types. For example, the pure functions `len` and `index` shown in Figure 5.3 can be used to specify the contents of the vector in the same way as ones used for linked list in Figure 5.1.

Annotation `#[trusted]` informs the verifier that the function's specification should be trusted without verifying its body. While this annotation should be used with extreme care, it enables users to focus on the fragment of the code base they care about. We encode trusted pure functions as abstract heap-dependent functions in Viper.

5.2 Generalising Functions and Quantifiers with Snapshots

In the approach presented so far, the only way we can have a value of a non-primitive type such as `Pair` is by having an instance of the corresponding predicate, which leads to two important limitations. First, the pure functions presented in the previous section can return only primitive types because they are based on Viper functions that cannot return permissions. Second, quantifiers allow quantifying only over values of primitive types. While Viper supports quantifying over permissions, such quantifiers are unsuitable for expressing user-written quantifiers that quantify over **values** of non-primitive types because they express possession of the quantified-over permission. We address this challenge by using a standard technique called snapshots [21][3]. The snapshots supported by our verifier implementation presented in this part of the thesis are a partial version of the one presented in Part III, which, for example, additionally supports chaining pure functions returning non-primitive types. Therefore, to avoid duplication, we show the general idea in this section and present the more general version in detail in Part III.

[21]: Reynolds (2002), 'Separation Logic: A Logic for Shared Mutable Data Structures'

3: Snapshots were added to Prusti after our publication [74]. Therefore, the work described in the publication could not be used to verify functional properties that require pure functions returning non-primitive types or quantification over non-primitive types like the linked list shown in Figure 5.1.

A *snapshot* is a mathematical value that fully captures the set of values stored in a group of heap locations. For example, the following algebraic data type (ADT) captures the values stored in a linked list of integers. We use the naming convention `Snap<T>` for an ADT representing the snapshot of `T`.

```
1 adt Snap<List> {
2     Nil()
3     Cons(value: Int, tail: Snap<List>)
4 }
```

In implicit dynamic frames, we can obtain a snapshot of a predicate instance using a heap-dependent function[4]. We call such functions `snap` functions (we use `snap<T>` as the name of the function that returns the snapshot of `T`). For example, Figure 5.4 shows a `List` predicate and the heap-dependent function `snap<List>` that returns its snapshot. Once the snapshot is linked to the predicate, we can use it to specify the properties of the list data structure without exposing its private fields.

4: Separation logic does not have heap-dependent functions. Therefore, in separation logic, a snapshot is related to a predicate by making it an additional parameter of the predicate.

While our work does not require snapshots for information hiding preserving specifications (because implicit dynamic frames solves this challenge with heap-dependent functions), we use them to make our pure

```
predicate List(this: Ref) {
    acc(this.value) && acc(this.next) &&
    (this.next != null ==> List(this.next))
}
function snap<List>(this: Ref): Snap<List>
    requires acc(List(this), read_perm)
{
    unfolding acc(List(this), read_perm) in
    this.next != null ?
        Cons(this.value, snap<List>(this.next))
        :
        Cons(this.value, Nil())
}
```

Figure 5.4: A definition of predicate `List` and heap-dependent function `snap<List>` that returns its snapshot.

```
1  adt Snap<Option<Box<T>>> {
2      None()
3      Some(content: Snap<Box<T>>)
4  }
5  adt Snap<Node<T>> {
6      Constructor(value: Snap<T>, next: Snap<Option<Box<T>>>)
7  }
8  function index(
9      self: Snap<&Node>,
10     index: Snap<usize>,
11 ): Snap<&T>
12     requires 0 <= index && index < len(self)
13 {
14     index == 0 ?
15     self.value :
16     (self.next.isSome() ?
17         index(self.next.content, index - 1) :
18         unreachable()
19     )
20 }
```

Figure 5.5: Snapshot definitions for types `Node<T>` and `Option<Box<T>>` and a simplified snapshot-based encoding of method `Node::index` from Figure 5.1.

functions and quantifiers more expressive. Figure 5.5 shows snapshot definitions for types `Node<T>` and `Option<Box<T>>` and a simplified encoding[5] of method `Node::index` from Figure 5.1. As can be seen from lines 8–20, we change the encoding of pure functions to use snapshots for both parameters and returned values, which also requires using snapshot-based operations instead of permission-based ones in the body of a function. Snapshot-based function encoding is very similar to permission-based encoding, which enables us to generate it without any user input[6]. We encode calls to pure functions by converting arguments into snapshots using functions that obtain the snapshot similar to `snap<List>` above. Changing the encoding of pure functions to a snapshot-based one enables us to support pure functions that return shared references, which is necessary to verify such examples as the one shown in Figure 5.1. We also use snapshots to generalise the quantifiers to enable quantification over arbitrary `Copy` types by encoding them as quantifying over snapshots.

The last bit related to snapshots we need to be able to support verifying the example in Figure 5.1 is a way to compare for equality values of arbitrary types. The Rust equality operator `==` is defined by the user by implementing the `PartialEq` trait and is, therefore, not guaranteed

5: We omit obvious conversions to improve readability. For example, we omit conversion from snapshots to Viper integers and conversion from a snapshot of a reference to a snapshot of its target.

6: The implementation of the version of Prusti presented in this part of the thesis requires annotations in some rare cases due to an implementation mistake. The implementation of snapshots for the version of Prusti presented in Part III requires no input from the user.

to adhere to the requirements for equality. We solve this problem by defining a snapshot equality operator === that compares the snapshots of two values. With this operator, we can compare values of type `T` even if this type does not implement `PartialEq` trait.

5.3 Pledges for Mutable Borrows

Method `index_mut` at lines 43–56 in Figure 5.1 mutably borrows the linked list and returns a mutable borrow of the element at the specified index. Since the method takes a mutable reference, it is allowed to modify the linked list in arbitrary ways, and, therefore, we need to specify to the client the state of the data structure after the effects have taken place. If a method did not return a reborrow, like, for example, method `set_head_value` shown in the following snippet, which changes the value at the head, we could specify the effect of the method in the postcondition. The postcondition of this method expresses that the element at index 0 was changed to the new value, and all other values stayed the same.

```
impl Node {
    #[ensures(self.index(0) == new_value)]
    #[ensures(
        forall(i: usize
            0 < i && i < self.len() ==>
            self.index(i) == old(self.index(i))
    ))]
    pub fn set_head_value(&mut self, new_value: i32) {
        self.value = new_value;
    }
}
```

The difference between `set_head_value` and `index_mut` is that the latter returns a reborrow that can be used by the client to modify the originally borrowed linked list node even after the call to `index_mut` terminated. For example, if the caller of `index_mut` assigns `42` to the returned reference, as shown in our motivating snippet, we should be able to prove that the element at index 4 is 42 after the returned reference expires as shown in the following snippet.

```
let r = list.index_mut(4);
*r = 42;
assert!(list.index(4) == 42);
```

One may be tempted to solve this challenge by specifying in the postcondition what changes the function already made and which part of the borrowed data structure the returned reference reborrows so that the client would have enough information to deduce how their changes via reborrow affect the final state of the data structure. However, this approach has multiple drawbacks. First of all, since `r` reborrows `list`, the Rust compiler forbids accessing `list` until `r` expires, which means that the postcondition mentioning `list` would be violating Rust ownership rules and, therefore, potentially be confusing. Second, to specify which part of the data structure is reborrowed, the postcondition would likely need to mention private fields, thus breaking the information hiding. Finally,

reconstructing how modifying a reference that points into the middle of a recursive data structure affects the entire data structure typically requires writing recursive lemmas. Automatically writing such lemmas is beyond state of the art and thus breaks the abstraction our technique aims to achieve.

We propose a new specification construct called *pledges* that addresses all mentioned challenges. A *pledge* is an assertion describing the value of the borrowed place immediately after the reborrow expires, parametric to the value of the reborrow just before it expires. The pledge is written inside a `#[after_expiry(reference => ...)]` annotation as shown in Figure 5.1 on lines 35–42. A typical pledge expresses the state of the borrowed value (in our example, `self`) by comparing it to the state just before the returned reference expires (part of the assertion wrapped in `before_expiry(...)`) and to the function pre-state (wrapped in `old(...)`). The pledge in Figure 5.1 expresses that the new value of the reborrowed element is equal to whatever was the last value of the returned reference before it expired and that all other elements of the linked list were unchanged. Notably, these properties are expressed using public pure functions without exposing any implementation details, which is essential for modular reasoning.

Encoding pledges into Viper is surprisingly straightforward. In magic wand $A \mathrel{-\!\!*} B$, A and B can not only be accessibility predicates but any Viper assertions. Therefore, we can conjoin the pledge translated into Viper to the right-hand side of the magic wand. For example, the following snippet shows a simplified magic wand with encoded line 41 of the pledge from Figure 5.1.

```
1 acc(T(result.pointer), write) --*
2     acc(Node_T(old(self.pointer)), write) &&
3     index(snap<&Node<T>(old(self.pointer)), old(snap<usize>(index))) ==
4         old[lhs](snap<T>(result.pointer))
```

Lines 1–2 show the part of the magic wand generated from the types and lines 3–4 show the encoding of the pledge corresponding to line 41 in Figure 5.1. We translate `before_expiry(...)` into `old[lhs](...)`, which evaluates the argument on the left-hand side of the magic wand. Intuitively, the state of the left-hand side of the magic wand is the state in which the magic wand is applied. When verifying the `package` statement that packages the magic wand with the pledge, Viper will automatically prove that the pledge holds with *any* final value of the reborrow. Similarly, for non-reborrowing functions, we either conjoin the specification describing the value of the borrowed place after the function call to the postcondition or the right-hand side of the corresponding magic wand depending on whether we use the optimisation described in Subsection 3.4.4.

Asserting pledges. In the paper in which pledges were originally published [74], we considered only pledges that are translated into assertions on the right-hand side of the magic wand. Such pledges have a property that the caller of the reborrowing function can assign any value to the returned reborrow, and it is the reborrowing function implementer's responsibility to ensure the correctness of the code. However, sometimes, the implementer would like to rely on the caller assigning only specific

[74]: Astrauskas et al. (2019), 'Leveraging Rust types for modular specification and verification'

values. For example, Figure 5.6 shows a struct `Segment` (lines 2–5) that has two fields representing start and end points and a method `borrow_start` (lines 13–15) that returns a mutable borrow to the start point. Suppose the implementer of this struct wanted to add an invariant (line 1) that the end point is always after the start point, they could not do this without removing the `borrow_start` method. The reason is that the caller of this method can use the borrow of the start point to assign any value to it, including ones that violate the invariant.

```
1  #[invariant(self.start <= self.end)]
2  pub struct Segment {
3      start: i32,
4      end: i32,
5  }
6
7  impl Segment {
8      #[assert_on_expiry(result =>
9          *result <= old(self.end),
10         self.start == before_expiry(*result) &&
11         self.end == old(self.end)
12     )]
13     pub fn borrow_start(&mut self) -> &mut i32 {
14         &mut self.start
15     }
16 }
```

Figure 5.6: A segment with start and end points that has an invariant that the end point is after the start point.

With Matthias Erdin [89], we explored a variation of pledges that enable constraining the value of the returned reference and found that they are a natural fit for programs that use types with invariants. Such *asserting pledges* are defined by using `#[assert_on_expiry(reference => constraint, assertion)]` annotation that in addition to regular pledge `assertion` also have `constraint` that is required to be satisfied when the reference expires. For example, in Figure 5.6, the asserting pledge on lines 8–12 constraints the returned reference when it expires to be smaller than the end point. The core advantage of asserting pledges is that, similarly to regular pledges, they enable the implementer to specify the constraint on the reference in terms of the public interface of the data structure without leaking any details of the internal implementation.

[89]: Erdin (2018), 'Verification of Rust Generics, Typestates, and Traits'

The asserting pledges are encoded into Viper by conjoining the encoded constraint to the left-hand side of the magic wand. Viper automatically considers the additional assumptions when packaging a magic wand with an asserting pledge.

Alternative designs. In this section, we presented a syntax for pledges that uses two annotations `after_expiry(A)` and `before_expiry(B)`. `after_expiry(A)` indicates that assertion `A` should be evaluated in the state immediately *after* the reference expires while `before_expiry(B)` indicates that `B` should be evaluated just *before* the reference expires. We chose this syntax because the reference blocks the place it (re-)borrows and, therefore, mentioning both in the same assertion would be invalid Rust, which we felt would be confusing for the users if we allowed it. However, it is important to note that using different syntax for the two cases is unnecessary because they can always be distinguished. The places that

should be evaluated in the after-expiration state have the borrowed place as a root, and the places that should be evaluated in the before-expiration state have the borrowing reference as a root. Therefore, we could rewrite `index_mut` pledge as a postcondition, as shown in the following snippet.

```
#[ensures(
    self.len() == old(self.len()) &&
    forall(i: usize
        0 <= i && i < self.len() && i != idx ==>
        self.index(i) == old(self.index(i))
    ) &&
    self.index(idx) == *result
)]
```

From the compiler, we know that `result` is a reference blocking `self`. Since `result` does not exist in the pre-state and cannot be used after it expires, the only valid state from the three states used in a pledge in which `*result` can be evaluated is the state just before the reference expires. Similarly, `self` cannot be evaluated before the reference expires, and the expressions that should be evaluated in the prestate are already wrapped in `old(...)`, so we are left with evaluating `self.len()` and `self.index(i)` just after the reference expires.

[90]: Hahn (2015), 'Rust2Viper: Building a static verifier for Rust'

This design was used by Florian Hahn, who did a master thesis [90] to evaluate the feasibility of the approach presented in this part of the thesis.

Implementation and Evaluation 6

We evaluated our technique by implementing a Rust verifier and examined its effectiveness in three ways: two automatic studies and manual verification of examples. Our evaluation shows that our technique can automatically generate core proofs and enables verifying important properties without the need for complicated specifications. We start this chapter by presenting our implementation in Section 6.1. Then, we present our evaluation in Section 6.2.

In this chapter, we present the implementation and evaluation as they were done for the paper [74] on which this chapter is based. Since then, the tool has improved, and some parts had to be rewritten because the components it relied on were deprecated. However, the high-level design of the implementation presented in this chapter is still effectively the same.

[74]: Astrauskas et al. (2019), 'Leveraging Rust types for modular specification and verification'

6.1 Presentation of the Prusti Tool

We implemented the verifier called Prusti as a Rust compiler plugin. This choice has enabled two significant benefits. First, our tool provides the same user interface as the Rust compiler and official linter Clippy [91]: verification errors are reported in the same way as the compiler reports compilation errors, with which Rust developers are already familiar. Second, our tool reuses a significant portion of compiler infrastructure, which gives us confidence that we interpret Rust programs in the same way as the Rust compiler. Our tool performs the central part of its work after the compiler's type-checking pass. We obtain from the compiler a CFG representation of each function called MIR (Middle Intermediate Representation). We use this representation because the borrow checker also uses it and, therefore, we can obtain the borrow information crucial for our technique. We use MIR with the type information to construct a Viper program, which we verify with Viper's symbolic execution verifier. Our implementation does not have an explicit intermediate representation with PCSs; we run the corresponding analyses on the generated Viper code, treating Viper permissions as capabilities. We encode mutable borrows using the optimised encoding presented in Subsection 3.4.4. If the verifier reports any errors, we translate them back and show them to the user by using the error reporting functionality provided by the compiler. To ensure that we correctly model Rust semantics, we developed a test suite of more than 300 correct and incorrect Rust programs annotated with expected verification errors.

[91]: Clippy contributors (2023), *Clippy*

Our tool supports an expressive fragment of Rust that, for example, includes shared and mutable borrows, traits, generics, and common uses of lifetime parameters to functions. As was mentioned in Subsection 2.3.3, we support a safe abstraction type `Box` by special-casing it. While safe

abstractions cannot be verified with this version of Prusti, it still can be used for verifying the code that uses these abstractions by providing trusted specifications for them: if a function is annotated with `#[trusted]`, its body is not verified. However, its specifications are still used when verifying callers. This feature can be used not only to specify types not supported by Prusti but also to focus the verification effort on components vital to the user. In addition to checking the functional specifications provided by the user, Prusti can optionally check the absence of panics and overflows. The implementation presented in [74] does not support reborrowing inside loops, pure functions returning non-primitive types, and abrupt termination of loop bodies. Unless explicitly stated, we present the evaluation of this version in the following section.

[74]: Astrauskas et al. (2019), 'Leveraging Rust types for modular specification and verification'

6.2 Evaluation of the Verification Approach

We evaluated our work in three ways, which we present in the following subsections. In Subsection 6.2.1, we present the construction of core proofs for supported functions from the 500 most popular crates (Rust packages). In Subsection 6.2.2, we discuss whether the absence of overflows can be automatically proven without the user providing any specifications. Finally, Subsection 6.2.3 discusses using user-provided specifications for verifying panic-freedom and richer functional properties. The performance measurements shown in the following subsections were done using a clean Ubuntu 18.04.1 installation on a desktop with a 4-core (8 hyper-threads) Intel i7-2600K 3.40GHz CPU, 32GB of RAM, and an SSD disk.

6.2.1 Automatically Generating Core Proofs

We evaluated the construction of core proofs on 500 crates that, on November 2, 2018, were the most popular on the Rust official package index `https://crates.io/` [92]. From the 500 crates, we obtained the list of functions on which we tested the construction of the core proof by applying three simple filters. First, we filtered out all crates (148) that did not compile within 15 minutes with the standard nightly compiler[1] and Polonius borrow checker[2] (without our tool). The remaining 352 crates left us with 56,257 functions and methods. Second, we applied a simple syntactic check for unsupported language features, leaving us with 11,801 functions. Finally, we manually removed ten unusually large functions, each requiring more than one minute just for the encoding. Five of these functions implement 4×4 matrix operations. The other five contain huge match expressions with up to 2,000 cases. In the end, we had 11,791 functions (21% of the total) on which we ran Prusti with disabled checks for panics and overflows.

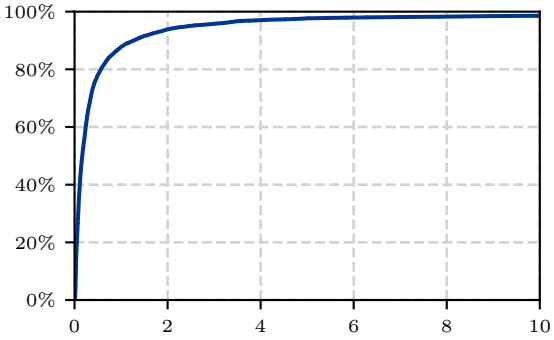
As expected, for each of 11,791 functions, Prusti generated a core proof successfully verified by the Viper symbolic execution verifier without any manual intervention. This successful verification shows that the core proofs generated by our technique are sufficiently complete to be accepted by Viper. However, the proofs were non-trivial. They required a significant number of Viper guiding statements: from 1,140,384 lines of

[92]: Rust community (2023), *The Rust community's crate registry*

1: Version: `nightly-2018-06-27`.

2: We used the following flags: `-Zborrowck=mir -Zpolonius -Znll-facts`. With these flags, the reference Polonius algorithm ("Naïve") was used.

generated Viper code, 138,499 lines were `fold`, `unfold`, `package`, and `apply` statements.



(a) The horizontal axis is the verification time in seconds. The blue line shows a cumulative distribution of how many functions from the supported 11,791 functions required at least that many seconds to verify. 188 functions took more than 10 seconds and, therefore, are not shown on the graph: 177 took between 10 and 120 seconds, while 11 took between 120 and 888 seconds.

Error	Num
assert!(..) might fail	2
unreachable!(..) might be reachable	1
add with overflow	77
remainder with a divisor of zero	50
remainder with overflow	48
divide by zero	42
divide with overflow	45
multiply with overflow	105
negate with overflow	18
subtract with overflow	71
solver incompleteness	8

(b) How often each error message was reported by Prusti for the 519 functions used for overflow freedom evaluation.

Figure 6.1: Results of the large scale evaluation.

Figure 6.1a shows how long it took Viper to verify each function, averaged over three runs. The graph shows that the average verification time per function is 1.2 seconds. However, only 0.16 seconds are enough to verify half of the functions and almost all of the functions (98.6%) are verified in less than 10 seconds each. 1.4% of functions that take more than 10 seconds are slow to verify for the same reasons as the ten unusually large functions we discarded (see above).

6.2.2 Automatically Checking Overflow Freedom

To evaluate whether runtime overflow checks can be proven without user interaction, we automatically collected all 519 supported functions from our evaluation presented in the previous subsection that contain runtime checks for integer overflows or division by zero. We re-ran Prusti on these functions with enabled panic and overflow checks. Out of 519 supported functions, Viper successfully verified 52. Our manual inspection revealed that used expressions could not overflow (for example, `a/2 + a/3`) or were guarded by range checks. Since our tool soundly proved that these checks can never fail, they could be removed to improve performance without compromising safety.

Table 6.1b shows verification errors reported for each of the remaining 467 functions. Manual inspection revealed that, in most cases, the authors of these functions implicitly assumed that the arguments would always be within the valid ranges. Our technique enables the authors to make these assumptions explicit by using preconditions that are verified at each call site. The runtime checks emitted by the Rust compiler must guarantee that a potentially overflowing operation can be executed only with arguments that would not lead to overflow, but Prusti failed to verify this property in eight cases. Manual investigation revealed that the examples contain non-linear arithmetic, which led the SMT solver Z3 used by Viper to struggle. We addressed the problem by increasing

the Viper timeout for Z3 queries from 10 to 60 seconds. After increasing the timeout, Prusti reported the expected "divide by zero" verification errors in all eight cases.

6.2.3 Specifying and Verifying Functional Behaviour

Table 6.1 shows an overview of the examples we collected to evaluate specification and verification of panic-freedom and richer functional properties. We collected the examples from three different sources. From the programming chrestomathy site Rosetta Code [93], we manually chose eleven examples that either fall into the supported subset or can be adapted without significant changes. Instead of taking the linked list example from Rosetta, we took a more extensive version from a Rust tutorial on linked lists [94]. The last two examples we took from Nicholas D. Matsakis' blog posts on Rust language design [95, 96]. These two examples illustrate complex borrowing patterns: "Message" (from [95]) was not supported by lexical lifetimes, and "Borrow First" (from [96]) still works only with Polonius.

[93]: Rosetta Code contributors (2018), *Rosetta Code*

[94]: Rust community (2019), *Learn Rust by writing Entirely Too Many Linked Lists*

[95]: Matsakis (2018), *MIR-based borrowck is almost here*

[96]: Matsakis (2018), *MIR-based borrow check (NLL) status update*

As the column "LOC" of Table 6.1 shows, the unmodified examples had 7 to 89 lines of code, not counting blank lines and comments. We had to modify some examples to make them fit into the supported subset. For these examples, we rewrote `for` loops as `while` loops, changed code to avoid `return` and `break` statements, and wrote trusted wrappers with specifications for standard library types. As shown in column "#Fns", each example had between 1 and 6 functions. The average total verification time (over three runs) is, for most examples, below 30 seconds. The two slowest examples, "Knight's tour" and "Knapsack Problem/0-1", take less than two and a half minutes. Each of these two examples has a large function that takes most of the time. For all examples, the standard deviation of verification time was around 1 second.

We grouped the examples in Table 6.1 into three groups. The first group contains seven examples which contain code that could panic or overflow. For these examples, we verified the absence of these behaviours by adding appropriate specifications[3]. The amount and complexity of specifications varied significantly for different examples, as seen from the column "Spec. LOC". On one end, example "Binary Search" needed only a simple loop invariant showing that the indices are within the required range. These two lines of specifications (for 16 lines of original code) enabled the verifier to prove the absence of both out-of-bounds accesses and integer overflows. On the other end were algorithms that simulate some traversal of a grid. "Knight's tour", where the goal of the algorithm is to find a path that visits every square of the chessboard, required a significant amount of ghost code (71 lines of specifications for 89 lines of original code) to maintain the invariants necessary to show that the algorithm does not try accessing squares outside of the defined chessboard, which would lead to runtime errors. Another example that required large specifications is "Langton's Ant", a cellular automaton implemented by simulating an ant walking on a grid according to predefined rules. While this example required fewer specifications than "Knight's tour" (22 lines of specifications for 58 lines of original code), the specifications were more complex because they had to maintain an invariant of the grid on which the ant walks, which involved using a pledge to specify

3: In the paper [74], "Selection Sort" in the first group is marked as an example that uses a generic type. However, as pointed out by Xavier Denis [97], this marking was an oversight: the original example and the version for which we verified the absence of panics and overflows both use signed integers. We fixed this mistake in our Table 6.1.

how borrowing an element of the grid affects the grid's state. When we tried to verify "Langton's Ant", we found that the example contained a bug: the authors used unsigned integer type `usize` for the ant's position, which could sometimes underflow. We fixed the bug by changing the type to `isize` and fixing the bounds checks.

For the second group of examples, we verified some properties beyond basic safety. This group also contains seven examples, two taken from the previous group. "Selection Sort" is the same as in the previous group, but with strictly more properties verified. "Binary Search" is a monomorphised version of the example from the previous group, where we changed polymorphic types to integers. We monomorphised this example because the evaluated version of Prusti did not support quantifying over non-primitive types, which is needed, for example, for expressing that the ordering function defined on a trait is transitive. Our initial attempt to verify the functional correctness of example "Binary Search" failed. During our investigation, we found an off-by-one bug that sometimes results in an element being skipped during the search. We

Table 6.1: Overview of the examples verified in the third part of the evaluation.

Columns: **LOC** is the number of lines in the unmodified example, **#Fns** is the number of functions, **Spec. LOC** is the number of lines used for specification and ghost code, **All Time** is the total time in seconds that includes the time needed to encode and verify the example, **Viper Time** is the time needed by the Viper symbolic execution backend verifier to verify the encoding, **No Panic** is whether we verified absence of panics, **No Overflow** is whether we verified absence of overflows. Value "–" used in **No Panic** or **No Overflow** columns means that the example contains no operations that could panic or overflow, respectively.

The table shows three groups of examples. We took the first two groups of examples from the Rosetta Code website [93] with the exception of example "Linked List Stack", which we took from [94] because it is more extensive than the one in Rosetta. For the first group of examples, we verified only the absence of panics and overflows, while for the second group, we verified properties that go beyond the basic ones. We had to monomorphise "Binary Search" to prove stronger functional properties because Prusti did not yet support quantification over non-primitive types required to express properties such as transitivity of the less-than operator. In [74], we incorrectly marked "Selection Sort" as generic; we fix the mistake here. For example "Ackermann Func.", we chose preconditions that do not prevent overflow as "×" indicated in the corresponding column. Specifying the correctness of the result and absence of overflows for example "Knapsack Problem/0-1" requires sum comprehensions, an advanced specification feature not yet supported in Prusti. Therefore, we verified only the correctness of all intermediate computations for this example. The last group of examples are from Nicholas D. Matsakis blog posts about non-lexical lifetimes in Rust illustrating complex borrowing patterns [95, 96]. We could not verify the absence of panics for example "Message" because the program does not handle all IO errors.

Example	LOC	#Fns	Spec. LOC	Time (s) All	Time (s) Viper	No Panic	No Overflow	Verified Additional Properties
100 doors	19	2	7	10.9	7.4	✓	✓	
Binary Search (generic)	16	1	2	16.2	12.9	✓	✓	
Heapsort	39	3	18	30.6	26.2	✓	✓	
Knight's tour	89	6	71	127.6	120.2	✓	✓	
Knuth Shuffle	16	2	3	9.5	6.2	✓	✓	
Langton's Ant	58	4	22	16.7	11.8	✓	✓	
Selection Sort (no-panic)*	20	2	8	19.2	15.2	✓	✓	
Ackermann Func.	16	2	17	7.4	4.4	-	×	Correct result
Binary Search (mono.)	16	1	29	25.5	21.4	✓	✓	Correct result
Fibonacci Seq.	46	6	26	9.1	5.7	-	-	Correct result
Knapsack Problem/0-1	27	1	86	139.4	131.6	✓	×	Correct computation
Linked List Stack	59	5	60	21.4	16.9	✓	-	Correct behaviour
Selection Sort (functional)	20	2	34	29.6	24.2	✓	✓	Sorted result
Towers of Hanoi	10	2	5	5.9	3.2	-	✓	Correct param. range
Borrow First	7	1	1	6.6	3.6	✓	✓	
Message	13	1	0	7.2	4.2	×	-	

fixed the bug and verified that if the function terminates, it produces the correct result. However, after the publication, we discovered that in some cases, the function does not terminate; we discuss the fixed version later, together with examples we used to evaluate snapshots. Other examples for which we verified functional correctness were "Fibonacci Seq." and "Ackermann Func."; for them, we showed that different implementations compute the same correct result. For "Knapsack Problem/0-1", a classical dynamic programming problem, we verified that the values stored in the table match the recursive formula. For data structure example "Linked List Stack", we specified and verified the functional behaviour of each method. For "Selection Sort", we verified that the result is sorted; for "Towers of Hanoi", we verified that the parameter values are valid.

4: Prusti git commit: `cee016e86b6-32cb368a8609c4bf450fc564282d5. Z3 version: 4.8.7. Viper version: v-2023-08-26-2125`.

Table 6.2 shows an overview of examples verified in a later version of Prusti[4], which supports snapshots. The performance was measured on a ThinkPad T470p laptop with a 4-core (8 hyper-threads) Intel i7-7700HQ 2.80GHz CPU, 16 GB of RAM, an SSD disk, and Ubuntu 20.04.6. To enable comparison with timings shown in Table 6.1, the first two examples in the table, "Binary Search (generic)" and "Binary Search (mono.)", are identical to the ones in Table 6.1, just with updated specification syntax. The other three examples are generic versions of the examples from Table 6.1 for which snapshots enabled us to verify some functional properties. The code of "Binary Search" is almost the same as that of "Binary Search (generic)", with the only difference that we added a break statement to fix the non-termination. We verified that the fixed version terminates by manually asserting at the end of the loop body that the size of the currently explored range is non-negative and strictly smaller than at the beginning of the loop body. The functional correctness properties we proved for this example are the same as the ones we proved for the monomorphised version "Binary Search (mono.)" (the non-termination did not affect the correctness of the specifications, so we could directly translate the existing specifications to generic versions). Similarly, "Selection Sort" is a generic version of the corresponding example from Table 6.1 for which we proved the same functional correctness properties. The final example, "Linked List", is a generic linked list taken from the same tutorial as "Linked List Stack", for which we additionally verified `index_mut` method similar to the one shown in Figure 5.1 in Chapter 5.

Table 6.2: Examples verified in an updated version of Prusti. The meaning of columns is the same as in Table 6.1, except "Time (s)" refers to the overall time needed to encode and verify the example. The first two examples ("Binary Search (generic)" and "Binary Search (mono.)") are identical to the ones in Table 6.1 just with updated specification syntax. "Binary Search" and "Selection Sort" are generic versions of the examples from Table 6.1 with verified functional properties. The last example, "Linked List", is based on [98], which is a generic linked list taken from the same tutorial as "Linked List Stack" in Table 6.1. For this example, we verified `List::new`, `List::push`, `List::pop`, and `test::basics` methods taken from [98], and a version of `List::index_mut` method from Figure 5.1 in Chapter 5 adapted to the example.

Example	LOC	#Fns	Spec. LOC	Time (s)	No Panic	No Overflow	Verified Additional Properties
Binary Search (generic)	16	1	2	3.1	✓	✓	
Binary Search (mono.)	16	1	29	5.5	✓	✓	Correct result
Binary Search	16	1	43	6.0	✓	✓	Correct result, terminates
Selection Sort	20	2	38	373.6	✓	✓	Sorted result
Linked List	39	5	81	11.9	✓	-	Correct behaviour

The examples in Table 6.2 demonstrate that features enabled by snapshots are crucial for verifying the functional correctness of generic code. All three examples required quantifying over non-primitive types and using functions that return non-primitive types. However, the evaluation also revealed that there is still work to be done to make the experience of verifying generic code smooth: when verifying monomorphic versions of our examples, we relied heavily on SMT solver's built-in support for integers and its knowledge of transitivity property of operators like less-equals. However, when verifying generic code, we had to manually express these properties using quantifiers, which is tedious and requires care to achieve good performance (as seen from the verification time of "Selection Sort" example). A possible solution to this problem would be implementing a standard library that provides well-designed primitives for reasoning about generic code, similar to the libraries provided by Why3 and Dafny.

Analysis of both groups of examples confirms three key advantages of our approach compared to the state-of-the-art verifiers for heap-manipulating programs that existed at the time of the publication. First, specifying the functional properties listed in Table 6.1 took, on average, 1.3 lines of specifications for each line of code. While this overhead is non-negligible, it is still significantly smaller than other tools. For generic versions of the examples listed in Table 6.2, the average increases to 2.6 lines of specifications for each line of code; as discussed before, the additional overhead could be reduced by implementing a library of verification primitives. Second, our annotations are effectively Rust expressions with standard first-order logic connectives that are much simpler to understand and use than the powerful logics required for specifying heap-manipulating programs. Third, our approach supports incremental verification. For example, proving the absence of panics and overflows for "Binary Search" example requires only two lines of specifications. A user can additionally verify that if the function reports that it found an element (by returning `Some`), the returned element is correct by adding another two assertions. Verifying the last bit that if the function reported that it did not find an element (by returning `None`), it does not exist in the input is slightly more involved: it requires using a quantifier to specify that the input is sorted. However, none of these specifications exposes the complexity of the underlying permission logic.

7 Related Work

This chapter discusses the work related to what we presented in this part of the thesis. Our work, presented in this part of the thesis, focuses on using Rust's capability type system to lower the barrier for verification significantly. As we mentioned in Section 1.2, Rust is not the first programming language with a capability-based type system. We discuss other languages with capability type systems in Section 7.1. We already discussed the related work on using type systems for verification in Section 1.1. In Section 7.2, we take another look at SYMPLAR [45] and compare it in more detail with Rust and our work. Then, in the following three sections, we discuss related work on Rust. In Section 7.3, we discuss verification tools for Rust. In Section 7.4, we present formalisations of Rust, except RustBelt [71], which we discuss in Section 7.5. Since borrowing is the key novelty of Rust, there are multiple proposals for modelling borrows. We compare these models in detail in Section 7.5.

[45]: Bierhoff (2011), 'Automated program verification made SYMPLAR: symbolic permissions for lightweight automated reasoning'

[71]: Jung et al. (2018), 'RustBelt: securing the foundations of the Rust programming language'

7.1 Capability-Based Type Systems

Rust uses capabilities to enable safe memory management and guarantee the absence of data races. Many type systems associate capabilities [46] with references to achieve stronger properties than the ones guaranteed by typical type systems. Most of the systems that were developed either as extensions to existing programming languages ([64] to C, [65] and [60] to C#, and [66] to Scala) or as built-in to new programming languages (Pony [62], Mezzo [61], Æminium [63]) focused on guaranteeing absence of data races, which became pressing with the raise of multi-core computers. However, compilers of new programming languages also sometimes exploited the additional information available in the type system for other means: for example, Pony showed that capabilities could be used for enabling distributed garbage collection. The main advantage of reference capability type systems is that they provide stronger guarantees than the regular type system at a comparatively low annotation cost. Our work shows how the strengths of the reference capability type systems could be exploited to enable lightweight verification.

[46]: Boyland et al. (2001), 'Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only'

[64]: Swamy et al. (2006), 'Safe manual memory management in Cyclone'

[65]: Fähndrich et al. (2006), 'Language support for fast and reliable message-based communication in singularity OS'

[60]: Gordon et al. (2012), 'Uniqueness and reference immutability for safe parallelism'

[66]: Haller et al. (2010), 'Capabilities for Uniqueness and Borrowing'

[62]: Clebsch et al. (2015), 'Deny capabilities for safe, fast actors'

[61]: Balabonski et al. (2016), 'The Design and Formalization of Mezzo, a Permission-Based Programming Language'

[63]: Stork et al. (2014), 'Æminium: a permission based concurrent-by-default programming language approach'

7.2 Ownership and Uniqueness for Verification

In Subsection 1.1.3, we discussed SYMPLAR [45] that exploited reference capabilities for verification. Our work presented in this part of the thesis improves over SYMPLAR at least in two ways. First, the fragment of Rust supported by our work accepts more patterns than are supported by SYMPLAR. SYMPLAR's type system does not have a concept similar to Rust's lifetimes and determines the length of a borrow based on the scope in which the borrow was created. The core advantage of scope-based borrows is that they are much easier to automate because the borrowing patterns have a tree-like structure, which leads to more explicit permission flow compared to the flows possible in an arbitrary control flow graph on

[45]: Bierhoff (2011), 'Automated program verification made SYMPLAR: symbolic permissions for lightweight automated reasoning'

which Rust borrow checker is defined. For example, tree-like borrowing patterns fit well the patterns supported by abstract read permissions [25], which, therefore, could be used to encode read capabilities. As witnessed by Rust switching from lexical to non-lexical lifetimes, programmers prefer a more expressive borrow management based on the uses of references than the easier-to-understand one based on lexical scopes. Our work showed that supporting Rust non-lexical borrows is possible and restricting programmers is unnecessary. The second crucial contribution of our work compared to SYMPLAR is pledges. While the SYMPLAR capability type system supports borrowing, SYMPLAR had no construct allowing modular specification of borrows.

[25]: Heule et al. (2013), 'Abstract Read Permissions: Fractional Permissions without the Fractions'

7.3 Rust Verification Tools

Before the publication of our work, there has been work on both *bounded* and *unbounded* verification of Rust programs. The initial tools for verifying Rust were bounded C verifiers adapted to Rust. CRUST [99] is based on CBMC [100], a bounded model checker for C. The key contribution of CRUST was the automatic generation of test drivers that exercise API calls in a way that is likely to trigger aliasing invariant violations if the tested safe abstraction did not uphold them. [101] added support for Rust to SMACK [102], a verifier that works by translating LLVM bytecode into Boogie. Similarly, [103] adapted KLEE [104], a symbolic execution engine for LLVM, to support bytecode generated by the Rust compiler. Both SMACK and KLEE supported adding checks as Rust expressions. The key downside of integrating at the LLVM level is that the Rust type information is already gone and cannot be used. However, it also enables these tools to support not only safe but also unsafe code. Since we integrate at a higher level, we can exploit the available type information for unbounded modular verification of significantly richer properties.

[99]: Toman et al. (2015), 'Crust: A Bounded Verifier for Rust (N)'

[100]: Clarke et al. (2004), 'A Tool for Checking ANSI-C Programs'

[101]: Baranowski et al. (2018), 'Verifying Rust Programs with SMACK'

[102]: Rakamaric et al. (2014), 'SMACK: Decoupling Source Language Details from Verifier Implementations'

[103]: Lindner et al. (2018), 'No Panic! Verification of Rust Programs by Symbolic Execution'

[104]: Cadar et al. (2008), 'KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs'

The first work on unbounded verification of Rust was a feasibility study of the approach presented in this part of the thesis by Florian Hahn [90]. He developed Rust2Viper, a small prototype that supported certain patterns of owned types and shared and mutable (re-)borrows. Even though Rust2Viper targeted an older version of Rust that still used the lexical borrow checker and had no MIR, it clearly showed that this approach of verifying Rust programs is worth exploring. Sebastian Ullrich, in his master thesis on Electrolysis [105], took a different path. The key observation he exploited is that safe Rust programs can be *purified*: encoded into equivalent pure functional programs. This observation enabled encoding Rust programs into the Lean theorem prover [106] and interactively verifying them. Electrolysis supported a fixed set of (re-)borrowing patterns via lenses [107]. The same observation was exploited by recent work at Galois that encodes Rust programs into SAW [108]. In contrast to these works based on purification, our technique does not require the user to construct proofs manually. Also, purification-based approaches have the same drawbacks as SYMPLAR: no clear path exists to support unsafe code.

[90]: Hahn (2015), 'Rust2Viper: Building a static verifier for Rust'

[105]: Ullrich (2016), 'Simple Verification of Rust Programs via Functional Purification'

[106]: Moura et al. (2015), 'The Lean Theorem Prover (System Description)'

[107]: Foster et al. (2005), 'Combinators for bi-directional tree transformations: a linguistic approach to the view update problem'

[108]: Dockins et al. (2016), 'Constructing Semantic Models of Programs with the Software Analysis Workbench'

Verus [109, 110] is an SMT-based verifier that, similarly to other purification-based verifiers, relies on the Rust type checker to be able to generate an encoding of the Rust program that contains no capability information.

[109]: Lattuada et al. (2023), 'Verus: Verifying Rust Programs using Linear Ghost Types'

[110]: Lattuada et al. (2023), 'Verus: Verifying Rust Programs using Linear Ghost Types (extended version)'

Unlike other verifiers for Rust, Verus focuses on developers who build verified systems from scratch instead of focusing on verifying existing Rust code. This different focus leads to different priorities: instead of prioritising supporting as many Rust constructs as possible, Verus prioritises high verification performance and an expressive specification language. For example, Verus has only basic support for Rust's mutable borrows, which are allowed only as function parameters. However, it gives the developer more control over how a part of the code is checked by an SMT-solver, enabling the user to achieve better and more stable verification performance. The key innovation of Verus is linear ghost code. While other purification-based verifiers rely on the Rust type checker only for executable Rust code, Verus uses it also to enable resource-based specifications. For example, the Verus standard library contains pointer type `PPtr` that, unlike Rust references, can be used for shared-mutable accesses. Verus ensures that the accesses are safe by requiring the user to provide an instance of `PointsTo` type that matches the pointer: reading from `PPtr` requires a shared reference to `PointsTo` and writing to `PPtr` requires a mutable reference to `PointsTo`. An instance of this `PointsTo` type acts as a points-to predicate in separation logic, effectively turning Rust type system into a lightweight checker for separation logic. Flexible linear ghost code enables specifying many interesting patterns but also has a cost: code written with the types from the Verus standard library is sound only if verified with Verus, which means that Verus is effectively a dialect of Rust [111].

[111]: Developers (2023), *Verus Tutorial and Reference: Memory safety is conditional on verification*

Flux [112] is a verifier for Rust that is based not on some program logic like the verifiers mentioned earlier but on refinement types. Flux enables the user to refine a Rust type by specifying a logical predicate restricting the set of values the type could hold. Compared to our work, the key advantages of Flux are its better performance and low annotation overhead. However, Flux is limited to properties that can be expressed with unary predicates. For example, it cannot be used to verify that the result of a sorting algorithm is sorted.

[112]: Lehmann et al. (2022), 'Flux: Liquid Types for Rust'

Initial versions of many verification tools focused on modelling borrows, which is the key novelty of Rust. We discuss these approaches in Section 7.5. Later, work on these verification tools switched its focus to verifying features from other programming languages, to which ownership and borrowing give a unique twist. [113] enabled verification of Rust closures in Prusti. [114] enabled verification of Rust iterators in Prusti while [115] enabled their verification in Creusot. Both Prusti extensions heavily rely on snapshots, demonstrating the importance of the ability to separate functional specification from the management of resources when specifying complex language features.

[113]: Wolff et al. (2021), 'Modular specification and verification of closures in Rust'

[114]: Bílý et al. (2022), 'Compositional Reasoning for Side-effectful Iterators and Iterator Adapters'

[115]: Denis et al. (2023), 'Specifying and Verifying Higher-order Rust Iterators'

7.4 Rust Semantics and Formalisations

Patina [116] was the first attempt to formalise a subset of Rust that included unique places and lexical borrowing. This work targeted the version of Rust before the first stable release, which led to Rust not matching the model anymore. [117] presented a formal semantics for a Rust-like language with ownership and borrowing modelled by using indirect capabilities (based on [118]). In their model, borrows stored in

[116]: Reed (2015), 'Patina: A formalization of the Rust programming language'

[117]: Benitez (2016), 'Short Paper: Rusty Types for Solid Safety'

[118]: Crary et al. (1999), 'Typed Memory Management in a Calculus of Capabilities'

reference variables are kept alive until the end of the function body. Oxide [119] formalised a type system that captures a subset of safe Rust that includes non-lexical lifetimes. Importantly, Oxide uses the same notion of lifetimes as Polonius. However, unlike us, the Oxide authors are not interested in reconstructing the backward capability flow. A later version of Oxide [120] introduced an Oxide type checker and a compiler from Rust to Oxide, which the authors used to ensure that Oxide matches Rust. The Oxide model was also used to implement a program slicer for safe Rust called Flowistry [121]. Flowistry uses the input of Polonius to implement may-alias analysis, whose results it uses to determine which statements could affect the value the user is interested in. The analysis looks similar to parts of Polonius, so it would be interesting to explore the differences and similarities between the two in more depth. [122] and [123] formalised subsets of safe Rust in the K Framework [124], which can be used to derive a verifier from the given semantics automatically. However, the derived verifier exposes the logic used to define the semantics to the user, while our goal was to enable the user to reason on the Rust level. Featherweight Rust [125] presented a formalisation of a type system for a greatly simplified version of Rust that, for example, omits control flow and still uses lexical lifetimes. Jung et al. [126] proposed operational semantics for Rust called Stacked Borrows that defines what non-aliasing assumptions the compiler can make when optimising code. The proposed semantics turned out to forbid important use cases. Therefore, Villani [127, 128] proposed a different version called Tree Borrows that aims to support the same key optimisations while allowing the important use cases. It would be interesting to try connecting these works to our PCSs because PCSs have a direct connection to how the Rust compiler reasons about code.

[119]: Weiss et al. (2019), 'Oxide: The Essence of Rust'

[120]: Weiss et al. (2021), 'Oxide: The Essence of Rust'

[121]: Crichton et al. (2022), 'Modular information flow through ownership'

[122]: Kan et al. (2018), 'K-Rust: An Executable Formal Semantics for Rust'

[123]: Wang et al. (2018), 'KRust: A Formal Executable Semantics of Rust'

[124]: Rosu et al. (2010), 'An overview of the K semantic framework'

[125]: Pearce (2021), 'A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust'

[126]: Jung et al. (2020), 'Stacked borrows: an aliasing model for Rust'

[127]: Villani (2023), 'Tree Borrows'

[128]: Villani (2023), *Tree Borrows: A new aliasing model for Rust*

7.5 Modelling Borrows in Rust-Like Languages

Since borrows are one of the core novelties of Rust, most work on Rust focuses on modelling them. The proposed approaches for modelling borrows target various use cases such as proving the soundness of Rust type system, manually verifying functional correctness of unsafe code in a proof assistant, and (like our own work in this part of the thesis) SMT-based verification of the functional correctness of safe code. In this section, we analyse the proposed models to answer two questions. First, are the models fundamentally different, or is some underlying idea common to all? For example, in Chapter 3, we discussed that the model of mutable borrows has to address four challenges to reconstruct the backward flow. In this section, we compare how different approaches address each challenge and their benefits and weaknesses. Second, is there a model suitable for SMT-based verification of mixed safe and unsafe Rust code? This question is important because we focus on verifying mixed safe and unsafe code in the rest of the thesis. By reviewing the related work, we could distinguish three main approaches for capturing backward flow: hiding it behind a library abstraction, using prophecies, and reconstructing explicit backward flow as in our work. We compare the technical differences of these approaches and their strengths and weaknesses in Subsection 7.5.1, Subsection 7.5.2 and Subsection 7.5.3, respectively. Finally, in Subsection 7.5.4, we discuss the findings concerning

our two questions and potential future work.

7.5.1 Borrowing via Library in RustBelt

RustBelt [71, 129] is a formalisation of Rust that, differently from prior work, focuses on unsafe code. As we mentioned in the introduction, in unsafe code, programmers can use unsafe language features such as raw pointers at the cost of becoming responsible for memory safety. Unsafe code is crucial for Rust: it enables extending the language with new type system primitives that are implemented as safe abstractions that safely hide the unsafe code inside them so that client code written in safe Rust cannot tell whether the implementation is unsafe or not. The goal of RustBelt was to formalise this extendable type system approach and state the precise rules under which it is sound. Instead of using a more standard progress-and-preservation approach used by most prior work, RustBelt chose a path based on *semantic typing*. The authors formalised semantic typing for Rust using a separation logic, which is powerful enough to prove the type-correctness of both safe and unsafe Rust code. Then, they showed that a syntactically type-correct (safe) Rust program is also semantically type-correct. This approach enabled them to prove the soundness of several complicated safe abstractions, which implies that these abstractions can be safely used by arbitrary safe code.

[71]: Jung et al. (2018), 'RustBelt: securing the foundations of the Rust programming language'
[129]: Jung (2020), 'Understanding and evolving the Rust programming language'

Since RustBelt had different goals from ours, the authors of RustBelt made different design choices and contributions. High-level differences are that RustBelt is based on the powerful higher-order separation logic framework Iris and is fully mechanised in the Coq proof assistant, which enables modelling the most complex types allowed by the Rust type system (for example, a type of a function returning itself) and gives a small trusted base. Our work is based on the first-order implicit dynamic frames logic and the SMT-based verification infrastructure Viper, which enables us to achieve a high degree of automation. RustBelt uses λ_{Rust} as a language. λ_{Rust} is similar to the Rust compiler's middle intermediate representation (MIR) but is based on continuation passing, which makes it more convenient to use in Coq. Before a verification expert can start verifying the type soundness of a function, they have to manually translate it into λ_{Rust}. Since RustBelt focuses on type soundness, it does not provide a clear way of verifying the functional correctness of Rust functions. While our implementation, Prusti, uses MIR for most of its work, the user interacts with Prusti at the Rust source level, which includes writing specifications in a specification language based on Rust expressions. Verifying a function in RustBelt requires the verification expert to directly use a complex separation logic while our work completely hides the complex logic from the user and is, therefore, suitable for regular programmers.

In the approach we presented in Chapter 3, we use the information available in Polonius to reconstruct the backward flow of capabilities when a borrow expires. Instead of reconstructing the backward capability flow, RustBelt introduced a lifetime logic that, similarly to the Rust compiler, enables implicit capability transfer when a lifetime expires. Intuitively, the lifetime logic can be understood as a library written in a functional programming language that implements a bank-like entity. Creating a borrow `let x = &'l1 mut a;` transfers the permission to the borrowed place `a` into the *bank*, and the bank issues two tokens: a mutable

borrow token and an inheritance token. The mutable borrow token grants exclusive permission to the borrower to access the borrowed place as long as lifetime `'l1` is alive. The inheritance token grants exclusive permission to the lender to regain access to the borrowed place as soon as lifetime `'l1` ends. That a lifetime is still alive is tracked via a lifetime token: any non-zero permission to a lifetime token indicates that the lifetime is still alive. The borrower can access the borrowed place by opening the borrow. Opening the borrow consumes the borrow token and a fraction of the lifetime token, giving permission to the borrowed place. Closing the borrow consumes the permission to the borrowed place and returns the borrow token and a fraction of the lifetime token. It is important that opening the borrow consumes some fraction of the lifetime token because it guarantees that the lifetime cannot be ended as long as there are any open borrows because ending the lifetime requires full permission to the lifetime token. Ending the lifetime consumes the lifetime token and produces a dead lifetime token. The lender can use the dead lifetime token and the inheritance token to regain permissions to the borrowed place, thus returning to the initial state of capabilities. Shared borrows are supported similarly with the difference that the shared borrow token is duplicable and opening it gives only read access.

The first challenge (*C1*) we had to address in our approach was handling the effects of PCS operations in backward capability flow, which required us to run the PCS elaboration algorithm on the loan-dependency graph. In contrast, the lifetime logic hides this complexity inside the bank. It provides an interface that enables the user to transfer the borrowed capability back to the lender in two simple steps. First, the user ends the lifetime to get the dead lifetime token. Second, the user regains the loaned capability using the dead lifetime token. To provide such a simple interface to the user requires the bank to do complex and precise accounting. Intuitively, the lifetime logic bank keeps two *accounts* (called boxes in [129]) for each lifetime: one for borrowers and one for lenders (there can be multiple borrowers and lenders because lifetime logic allows creating multiple borrows for the same lifetime). When a place `a` is borrowed for the lifetime, it is put into the borrower's account to indicate that the borrower can access it at any point. When the lifetime expires, all resources from the borrower's account are transferred into the lender's account to indicate that the resources from now on belong to the lender. Maintaining the accounts requires solving three technical challenges. First, the borrower's account needs to be maintained in sync with the borrower's tokens. For example, if a borrower splits a mutable borrow token for a struct into mutable borrow tokens for its fields, the corresponding accounts must be split, too. Second, the lifetime logic has to guarantee that through all operations performed on borrowers' accounts, these accounts together always contain sufficient resources to justify all lenders' requirements for that lifetime. Third, reborrowing allows borrowing away the contents of a borrower's account, which requires ensuring that all reborrows return their borrows before the original borrow expires. Solving these challenges required complex technical solutions, which we do not discuss here. However, RustBelt's authors successfully solved these technical challenges, which enabled them to completely hide *C1* within the library. Since the complexity of applying PCS operations in the backward flow is completely hidden inside the library, the encoding that uses such a library-based approach

[129]: Jung (2020), 'Understanding and evolving the Rust programming language'

could be potentially simpler than ours. Therefore, exploring whether the library-based approach could be applied to SMT-based functional verification would be interesting.

Hiding PCS operations inside the library is not the only aspect of the lifetime logic approach that could lead to potentially simpler to generate encodings, which makes it an interesting approach for automated verification. In the lifetime logic, the state transitions are tied to changes in lifetimes and not to hitting verification boundaries like in our approach (*C0*). In our approach, when a verification boundary, such as the end of a loop or method body, is reached, all unneeded capabilities must be collected into the corresponding exchange capabilities to ensure that they are transferred back. In RustBelt, borrow tokens can be just leaked because they are not needed for recovering the borrowed place and leaking them does not affect the internal state of the bank. However, this flexibility comes at the cost of weaker specifications: RustBelt only ensures that the borrow is guaranteed to maintain its type invariant, precisely the property needed for verifying type system soundness. Our encoding enables the implementation of asserting pledges that assert a user-specified functional property at the point when the reference expires.

The second challenge (*C2*) of handling conditional control flow we approached by introducing ghost variables `l*_executed` that track whether a loan was created or not. The lifetime logic tracks what loans were created in the internal state of the bank. An important advantage of the RustBelt's approach is that it avoids using ghost variables and branching on them, which is important because branches can significantly impact verification time. However, tracking loans in the internal state also has one crucial drawback: it does not allow attaching functional specifications.

The third challenge (*C3*) of handling reassignments of mutable references we approached by using temporary places in the loan-dependency graph instead of the overwritten ones. In the lifetime logic, this challenge is handled implicitly by the fact that the bank sees only the targets of the references and not the references themselves.

To summarise, RustBelt's lifetime logic provides a library-like interface for managing borrows that, similarly to the Rust compiler, enables implicit backward capability flow. The essential advantage of lifetime logic is that it enables verification of the soundness of safe abstractions, which was the project's goal. While RustBelt targets manual verification of soundness in a proof assistant, it seems likely that the library-based approach could also benefit SMT-based verification. In particular, pushing complexity into the library that is proven once and for all may lead to a simpler and more performant encoding of the core proof for Rust programs. Therefore, an interesting future research direction would be to build an SMT-based verifier that uses the library-based approach and evaluate its performance, simplicity of the encoding, and completeness with respect to the Rust compiler (similarly to how we evaluated Prusti in Subsection 6.2.1). Since RustBelt focused on type soundness, it left an unanswered question of how the library-based approach could be used to verify functional correctness. We discuss one possible solution in the following subsection.

7.5.1.1 RustBelt Extensions

The original RustBelt paper [71] made a simplifying assumption that the language is sequentially consistent. However, many safe abstractions designed for concurrent code try to achieve maximum performance and, as a result, rely on relaxed memory operations. Therefore, [130] adapted RustBelt to lift the assumption that the language is sequentially consistent.

[71]: Jung et al. (2018), 'RustBelt: securing the foundations of the Rust programming language'

[130]: Dang et al. (2020), 'RustBelt meets relaxed memory'

Another change to RustBelt was explored by the authors of `GhostCell` [131]. `GhostCell` is a safe abstraction that enables creating cyclic data structures without the performance overhead caused by other solutions such as reference counted smart pointers `Arc<T>`. For soundness, `GhostCell` relies on the client being unable to use lifetime `'l1` used in the type of one `GhostCell<T, 'l1>` to call a method of another `GhostCell<T, 'l2>` if the two lifetimes `'l1` and `'l2` are not syntactically the same lifetime. However, original RustBelt used a semantic notion of lifetimes that, for example, allowed proving that all lifetimes are equal to `'static` if the code provably does not terminate[1]. Therefore, the authors of [131] added a syntactic notion of lifetimes to RustBelt, which allowed them to prove `GhostCell` sound. Interestingly, the algorithm used in Polonius would also conclude that all lifetimes are equal to `'static` if it managed to prove that the code does not terminate. However, Polonius cannot prove non-termination in practice because it has to consider that any called Rust function might panic, which is arguably a very fragile protection.

[131]: Yanovski et al. (2021), 'GhostCell: separating permissions from data in Rust'

1: For example, because it calls `std::process::abort`.

[131]: Yanovski et al. (2021), 'GhostCell: separating permissions from data in Rust'

7.5.2 Prophecies

In our model, the value of a memory location is coupled with the capability to access that memory location: whoever can access the memory location also knows its value. In Section 5.2, we showed that for specifying functional correctness, it is often helpful to introduce snapshots, which are mathematical values that fully capture the meaning of a group of memory locations. These snapshots are still linked to the capabilities that justify them. As a result, by restoring the explicit backward capability flow, we also reconstruct the knowledge of how the actions performed on references affect the values of borrowed places. As was shown by SYMPLAR [45], in some cases, it is possible to completely separate reasoning about capabilities and values: if we have the flow of capabilities checked by, for example, a type system, we can exploit this knowledge and construct a *purified* encoding that tracks only values. When applying this approach for Rust, the main challenge is finding a way to support mutable references, which requires answering the following three questions:

[45]: Bierhoff (2011), 'Automated program verification made SYMPLAR: symbolic permissions for lightweight automated reasoning'

1. How do we capture the backward value flow when a reference expires?
2. How do we justify that the backward value flow is correct?
3. How do we enable the user to specify the flow?

In our encoding into Viper, magic wands capture point 1 and point 2 while pledges enable point 3. In this subsection, we discuss an alternative way of capturing the backward flow based on a concept called *prophecies*, and, in the next section, we discuss approaches that explicitly capture the backward value flow. We focus on mutable borrows and generally

omit discussing shared borrows because supporting them in purification approaches is straightforward.

Up to now, prophecies have been explored in three papers, each focusing on a different question from our list. RustHorn [132] introduced prophecies and focused on using them to capture the backward value flow. We present prophecies in Subsubsection 7.5.2.1. While RustHorn [132] did contain a soundness proof that showed the approach is sound for a core fragment of safe Rust, the justification of the prophetic approach was the core focus of the RustHornBelt paper [133]. As indicated by the name, RustHornBelt combined RustBelt with prophecies to enable functional verification of unsafe code, which required finding a way to justify the prophetic approach even in the presence of unsafe code. We discuss RustHornBelt in Subsubsection 7.5.2.2. Lastly, Creusot [134] focused on prophecies as a specification mechanism for reasoning about mutable borrows in the context of modular unbounded SMT-based verification, which we discuss in Subsubsection 7.5.2.3. With prophecies introduced, we compare them with magic wands and pledges used in our work in Subsubsection 7.5.2.4 and Subsubsection 7.5.2.5, respectively.

[132]: Matsushita et al. (2020), 'RustHorn: CHC-Based Verification for Rust Programs'

[132]: Matsushita et al. (2020), 'RustHorn: CHC-Based Verification for Rust Programs'

[133]: Matsushita et al. (2022), 'RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code'

[134]: Denis et al. (2022), 'Creusot: A Foundry for the Deductive Verification of Rust Programs'

7.5.2.1 RustHorn: Capturing Backward Value Flow With Prophecies

The goal of RustHorn [132] was to enable functional verification of Rust programs using an approach based on constrained Horn clauses (CHCs) [135, 136]. Similarly to unbounded modular verification discussed in this thesis, one of the critical challenges in CHC-based verification is controlling aliasing, which makes Rust an appealing verification target. Similarly to Electrolysis [105], RustHorn exploits Rust's ownership type system to treat functions written in safe Rust as if they were written in a pure functional programming language. The key innovation of RustHorn was a novel way of modelling references as a pair of a current value and a *prophecy* of what the *final* value the reference will have when it expires.

[132]: Matsushita et al. (2020), 'RustHorn: CHC-Based Verification for Rust Programs'

[135]: Grebenshchikov et al. (2012), 'Synthesizing software verifiers from proof rules'

[136]: Bjørner et al. (2015), 'Horn Clause Solvers for Program Verification'

[105]: Ullrich (2016), 'Simple Verification of Rust Programs via Functional Purification'

```
let x = &mut a;
*x = 5;
assert!(a == 5);
```

(a) A simple borrow.

```
var a: Int
var x_current: Int
var x_final: Int

assume x_current == a
havoc a
assume x_final == a

havoc x_current
assume x_current == 5

assume x_current == x_final

assert a == 5
```

(b) An encoding of the simple borrow using the prophetic model.

Figure 7.1: A simple borrow example and its encoding using the prophetic model.

The prophetic encoding has some interesting relation to our work, understanding which requires to have an intuition of how it works. Figure 7.1 shows a simple borrowing example (Figure 7.1a) and its encoding in Viper based on the prophecies approach (Figure 7.1b). In Figure 7.1a,

place a is borrowed by x and then its value is set to 5 via this reference. Since the RustHorn approach is based on purification, in the encoding shown in Figure 7.1a, there are no resources on the Viper heap, and the variable a is encoded as a Viper integer `Int`. Similarly, `x_current`, the current value of reference x, and `x_final`, the final value of reference x, are also just integers. Borrowing is done in two steps. First, the current value of the reference becomes equal to the value of the borrowed place as shown on line 5 in Figure 7.1b to model the fact that when reading via x we will observe the same values as a had at the time of borrowing. Second, the borrowed place and the final value of the reference are set to the same fresh prophecy as shown on lines 6–7 to model the fact that when the reference x expires, a will have the same value as x at that time. Lines 9–10 show how the assignment to the reference is encoded; for consistency with the rest of the example, we encode it by havocking the target and then assuming the new value. Finally, when a reference expires, its current value becomes its final value, which means that we can learn the value of the prophecy by *resolving* the prophecy that assumes both values to be equal as shown on line 12. As a result, we can prove that the new value of a is 5.

In our model presented in Chapter 3, we reconstruct the backward value flow by explicitly telling the verifier how to transfer the capabilities back to the lenders once the borrows expire. This explicit capability flow enables the verifier to reconstruct the backward value flow. The prophetic approach teaches the verifier how to reconstruct the backward value flow by linking the prophetic values of references via assumptions. For example, if we have a reborrow statement `let y = &mut *x;` after which x is not used anymore, the effect of the prophetic encoding will be equivalent to the two statements shown in the following snippet.

```
1  y_current := x_current;
2  assume y_final == x_final;
```

This snippet shows that the final value of x was linked to the final value of y. An important property of prophetic encoding is that these links depend on the control flow. The following snippet resembles our motivating example from challenge *C2*.

```
1  if random_choice() {
2      z = &mut *x;
3  } else {
4      z = &mut *y;
5  }
```

Depending on which branch is executed, the reference z will be linked either to x or y. This way, the prophetic encoding constructs the precise backward value flow without introducing ghost variables, which we needed for solving *C2*. For the same reason that the link is established when the borrow is created, the prophetic encoding handles reassignments of references (*C3*) without any additional effort. Handling of PCS operations (*C1*), which in the value context means how deeply definitions should be unrolled, RustHorn leaves to the heuristics of the underlying verifier. One may wonder whether it is possible to link the capabilities in the same way as the prophetic encoding links values to avoid using ghost variables required by our approach. Such linking would be possible,

but only if we had a verifier that can completely automatically compute all necessary PCS operations (in other words, if we pushed the PCS elaboration step into the verifier).

While the magic wands used in our approach capture the backward flow of values and justify it, the prophetic approach requires an external argument that shows that the same prophecy cannot be resolved to different values. The RustHorn paper [132] proved that the prophetic encoding is sound for a core fragment of safe Rust. In the following subsection, we present RustHornBelt that showed how prophetic reasoning can be supported even in Rust code that contains unsafe.

[132]: Matsushita et al. (2020), 'RustHorn: CHC-Based Verification for Rust Programs'

7.5.2.2 RustHornBelt: Justifying Prophetic Backward Value Flow

Justifying prophetic backward value flow requires solving two key challenges. The first challenge is relating the mathematical values used in the encoding to capabilities that the Rust program manipulates. The second challenge is ensuring that a prophecy cannot be resolved to two different values because that would be unsound. RustHorn proved these two properties against the type system of safe Rust, which left an unanswered question of how to support specifying and verifying unsafe code. RustHornBelt [133], as the name implies, followed the path of RustBelt: updated the definition of semantic typing to include reasoning about mathematical values and prophecies and proved that syntactic typing implies semantic typing. RustHornBelt addressed the first challenge of relating RustHorn mathematical values to RustBelt resources by using the snapshot approach for separation logics, which we discussed in Section 5.2: they changed the definition of a predicate that models an owned Rust value to include also the snapshot (called "representation value" in the paper [133]). To address the second challenge of guaranteeing that prophecies are used soundly, RustHornBelt introduced a new separation logic construct called *parametric prophecies*, which enables ensuring the desired property that the same prophecy cannot be resolved to different values. RustHornBelt changed how the RustBelt predicates are defined and how the lifetime logic bank is used. However, it kept the implementation of the lifetime logic bank unchanged: the values of prophecies are effectively tracked outside of the lifetime logic. As a result, in RustHornBelt, mutable borrows are modelled on three levels: RustBelt lifetime logic tracks capabilities, snapshots are used for tracking values, and parametric prophecies with some other Iris constructs are used for soundly linking the two together. Similarly to RustBelt, RustHornBelt had to solve many technical challenges we are not discussing here.

[133]: Matsushita et al. (2022), 'RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code'

[133]: Matsushita et al. (2022), 'RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code'

7.5.2.3 Creusot: Prophecies as a Specification Tool

In our work, we used magic wands to capture the precise backward value flow and pledges to enable the user to specify the backward value flow when the precise flow cannot be restored, for example, when crossing a verification boundary. Since prophecies link values and not resources, they can be used as an alternative way to provide specifications on the level of Rust expressions, enabling lightweight modular unbounded verification. Creusot [97, 134] is a Why3 [57] front-end that uses a prophecy-based purification approach to enable unbounded verification

[97]: Denis et al. (2021), *The CREUSOT Environment for the Deductive Verification of Rust Programs*

[134]: Denis et al. (2022), 'Creusot: A Foundry for the Deductive Verification of Rust Programs'

[57]: Filliâtre et al. (2013), 'Why3 - Where Programs Meet Provers'

of safe Rust programs. Similarly to our work, Creusot uses a specification syntax based on Rust expressions with the key difference that instead of pledges, they use prophecies: Creusot introduces new dereference operator ^ that allows accessing the final value of the reference while regular dereference operator * allows accessing the current value of the reference. The following snippet shows a simple reborrowing function and its specification in Creusot that expresses that the final value of field `first` will be equal to the final value of the result and that field `second` will be unchanged (Creusot does not have `old(...)` expressions and always evaluates arguments in the pre-state of the function).

```
#[ensures(
    (^pair).first == ^result &&
    (^pair).second == (*pair).second
)]
fn reborrow_first(pair: &mut Pair) -> &i32 {
    &mut pair.first
}
```

The authors of Creusot also showed that the specification approach based on prophecies can be used for modular specification of types containing references inside them. For example, the following snippet shows a function that takes a generic `Box<T>` and drops it.

```
#[ensures(b.resolve())]
fn destroy<T: Resolve>(b: Box<T>) {}
```

Since the parameter `b` is dropped, any reference contained in `T` will expire. However, since `T` is generic, the implementation of `destroy` does not know whether `T` contains any references, and if it does, it has no way of referring to them. Creusot solves this challenge by introducing the `Resolve` trait whose `resolve` method specifies what it means for all references within a specific type to be expired. For example, this trait is implemented for mutable references of integers, as shown in the following snippet (we slightly simplified the definition to make it more readable).

```
impl Resolve for &mut i32 {
    #[predicate]
    fn resolve(self) -> bool { ^self == *self }
}
```

The `resolve` method is annotated with `#[predicate]`, which means that it contains an assertion and not an executable code. Unsurprisingly, the `Resolve` implementation expresses that the current and final values of expired references are the same. With this implementation, we can verify the following snippet that calls the `destroy` function with a mutable reference.

```
let x = &mut a;
let y = Box::new(x);
**y = 5;    // Double dereference because we need to also
            // dereference the Box.
destroy(y);
assert!(a == 5);
```

An important difference between Creusot and Prusti is that Creusot relies on the Rust compiler for tracking capabilities while Prusti reconstructs the core proof that is checked by Viper. This different design leads to two important consequences. First, Creusot requires an intricate soundness proof showing that relying on the Rust compiler can ensure that each prophecy is resolved at most once, and it is impossible to create prophetic cycles. Second, pushing the complexity into the soundness proof makes it much easier for Creusot to support complex control flow. For example, supporting reborrowing in loops in Prusti is challenging: even though the loop invariant computed by Polonius expresses the same constraints as a function signature, recovering exchange capabilities is much more complicated because of the many shapes loops can take. Creusot side steps this problem by relying on the Rust compiler to check the capabilities.

In the following subsubsections, we compare how the prophetic approach differs from ours in detail. We start by comparing prophecies to magic wands in Subsubsection 7.5.2.4.

7.5.2.4 Prophecies and Magic Wands

On a surface level, ours and the prophetic approach look very different, so the question arises whether they are related in any way. Magic wands relate both permissions and values, while prophecies relate only values. Therefore, to compare the two, we look "under the hood" and see how magic wands are implemented in Viper. Viper's symbolic execution backend used by Prusti uses snapshots to track the values of the resources [137], which enables it to support heap-dependent expressions and functions. The following equation shows the definition of the magic wand. This definition expresses that magic wand $A \mathrel{-\!\!*} B$ holds in state σ iff combining σ with any state σ' for which A holds gives us a state in which B holds.

[137]: Schwerhoff (2016), 'Advancing Automated, Permission-Based Program Verification Using Symbolic Execution'

$$\sigma \models A \mathrel{-\!\!*} B \;\Leftrightarrow\; \forall \sigma' \perp \sigma \cdot\; (\sigma' \models A \;\Rightarrow\; \sigma \uplus \sigma' \models B) \tag{7.1}$$

Let s_A be the snapshot of A and s_B be the snapshot of B. Then, from the definition of a magic wand, we can see that for each magic wand, there must be a function f_{wand} such that applying it to any valid snapshot s_A, we get a valid snapshot such that $s_B = f_{\text{wand}}(s_A)$ [137][2]. Packaging a magic wand must define this function f_{wand}, and applying the magic wand obtains the snapshot of B by applying f_{wand} on the snapshot of A in that state.

[137]: Schwerhoff (2016), 'Advancing Automated, Permission-Based Program Verification Using Symbolic Execution'

2: This definition of a magic wand snapshot should not surprise readers familiar with Curry-Howard correspondence, which expresses that propositions can be viewed as types. If we considered propositions in separation logic, magic wand $A \mathrel{-\!\!*} B$ (also called separating implication) would correspond to a function that takes resources A and produces resources B.

Malte Schwerhoff [137] observed that the magic wand snapshot can be simplified from a function to a constant under certain conditions (magic wand is applied only once; applying expressions and fractional magic wands are not allowed). More specifically, instead of defining function f_{wand} that takes the snapshot of A as an argument, we can represent it as a constant that is constrained against a fresh snapshot s'_A representing the snapshot of A at the hypothetical state in which the magic wand is applied [137]. To see how this observation enables us to relate prophecies and magic wands, consider Figure 7.2 showing an encoding of Figure 7.1a in Viper. In addition to Viper statements and available resources, the example shows the snapshots for resources and

[137]: Schwerhoff (2016), 'Advancing Automated, Permission-Based Program Verification Using Symbolic Execution'

[137]: Schwerhoff (2016), 'Advancing Automated, Permission-Based Program Verification Using Symbolic Execution'

```
1  acc(i32(a), write): s1
2
3  x.pointer := a;
4  package acc(i32(x.pointer), write) --* acc(i32(a), write)
5  acc(i32(x.pointer), write) --* acc(i32(a), write): swand
6  acc(i32(x.pointer), write): s1
7  slhs = swand
8
9  unfold acc(i32(x.pointer), write)
10 x.pointer.val_i32 := 5;
11 acc(i32(x.pointer), write) --* acc(i32(a), write): swand
12 acc(i32(x.pointer), write): s2
13 s2 = 5
14
15 fold acc(i32(x.pointer), write)
16 apply acc(i32(x.pointer), write) --* acc(i32(a), write)
17 acc(i32(a), write): swand
18 s2 = slhs
19
20 unfold acc(i32(a), write)
21 assert a.val_i32 == 5
```

Figure 7.2: A Viper encoding of the example from Figure 7.1a with shown snapshots for resources (in blue) and the equalities learned between snapshots (in green). Since snapshots are immutable, we show only newly learned equalities. We omit the snapshot changes done by `unfold` and `fold` statements for readability.

learned equalities between them. Creating a borrow on lines 3–7 transfers the predicate instance `i32` from place `a` to `x.pointer` and packaging a magic wand that represents a capability to undo this action. The transfer of the predicate instance in our example is shown by removing the corresponding resource on `a` and creating a resource on `x.pointer` with the same snapshot s_1 (as discussed in Subsection 3.4.4, Viper remembers that the two values are aliased instead of transferring the resource, but to simplify the explanation we pretend that it does transfer). Packaging produces a magic wand with snapshot s_{wand}. This snapshot depends on the snapshot s_{lhs} that represents the snapshot of the left-hand side of the magic wand when it is applied. Since the magic wand is trivial, the two snapshots are simply equated to each other. Assigning the target of the reference on lines 10–13 creates a fresh snapshot for the corresponding resource and assumes it to be equal to the snapshot of the assigned value. When the reference expires, the magic wand is applied (lines 15–18 |), which consumes the magic wand and reference target resources, produces the resource that corresponds to the right-hand side of the magic wand with the wand's snapshot s_{wand}, and assumes that the snapshot of the hypothetical left-hand side is equal to the actual left-hand side. If we compare the prophetic encoding shown in Figure 7.1b with how the snapshot values are updated in Figure 7.2, we can see that the two are effectively the same with hypothetical snapshot s_{lhs} acting as a prophecy. In other words, hypothetical snapshot s_{lhs} is effectively a prophecy.

We build on this observation to show that prophecies and magic wands are highly related, and we could also extend our approach to support prophetic specifications. To make the explanation of how we could justify prophetic specifications with magic wands more explicit, we need a way to refer to the snapshot of a magic wand[3]. In Section 5.2, we presented `snap` functions that can be used to obtain predicate snapshots. By using

3: In Viper, we could avoid using the snapshots directly by relying on heap-dependent expressions.

these functions, we can rewrite a magic wand `A(x)` $-\!\!\circledast$ `B(y)` that expresses that reference `x` reborrows from reference `y` to an equivalent magic wand shown in the following assertion.

$$\texttt{A(x)} \mathbin{*} \texttt{snap\_A}(x) = \underbrace{s_A}_{\texttt{\^{}x}} \mathrel{-\!\!\circledast} \texttt{B(y)} \mathbin{*} \texttt{snap\_B}(y) = \underbrace{f_{\text{wand}}(s_A)}_{\texttt{\^{}y}} \tag{7.2}$$

In this assertion, s_A is a snapshot of predicate `A(x)` and corresponds to the final (prophetic) value of `x` while $f_{\text{wand}}(s_A)$ is a snapshot of predicate `B(y)` and corresponds to the final value of `y`. One of the key contributions of Viper's support for magic wands [31, 33] is the package algorithm that infers the resources that need to be taken from the context to prove the magic wand and that as part of this process defines f_{wand}. In our work, we always know precisely what permissions (capabilities) should be consumed by the magic wand and, therefore, can manually define f_{wand}. Therefore, using f_{wand} explicitly in a wand would require more implementation effort but would not cause additional challenges. However, if we tried to use this form of a magic wand to model mutable borrows, we would run into the exact problem that prophecies and magic wands are supposed to solve: we would have to provide the final value of s_A when packing the magic wand while the magic wand should work for *any* value of s_A. We can avoid this problem by quantifying over s_A, as shown in the following assertion[4].

[31]: Schwerhoff et al. (2015), 'Lightweight Support for Magic Wands in an Automatic Verifier'
[33]: Dardinier et al. (2022), 'Sound Automation of Magic Wands'

$$\forall s_A \cdot (\texttt{A(x)} \mathbin{*} \texttt{snap\_A(x)} = s_A \mathrel{-\!\!\circledast} \texttt{B(y)} \mathbin{*} \texttt{snap\_B(y)} = f_{\text{wand}}(s_A)) \tag{7.3}$$

4: Alternatively, we could use Viper's `old[lhs](...)` expression that allows referring to the state on the left-hand side of a magic wand. In that case, the wand would be A(x) $-\!\!\circledast$ B(y) $*$ `snap_B(y)` = f_{wand}(`old[lhs](snap_A(x))`).

It is important to note that $\forall s_A$ in this assertion is a *logical* quantification, an unbounded generalisation of regular conjunction (as opposed to separating conjunction). s_A in this assertion ranges over all possible *valid* snapshots of `A(x)`[5]. This assertion cannot be written in the current version of Viper because Viper's `forall` has a requirement that the quantified resources are injective with respect to bound variables [30], which does not hold for our assertion. However, we could still write it in an implicit dynamic frames logic formalised in a proof assistant such as Coq or Isabelle/HOL.

5: The snapshots we presented in Section 5.2 include only valid values. The snapshots we present in Part III also include invalid values and, therefore, would require adding a guard that the snapshot is valid.

As presented in Section 5.3, we encode pledges by conjoining the encoded assertion to the right-hand side of a corresponding magic wand. The encoded assertion relates the left and right sides of the magic wand. Since s_A captures the value of the left-hand side and $f_{\text{wand}}(s_A)$ captures the value of the right-hand side, we can represent the encoded pledge as a boolean function $f_{\text{pledge}}(s_A, f_{\text{wand}}(s_A))$. We would get the following assertion by conjoining this function to the right-hand side of the magic wand.

$$\forall s_A \cdot (\texttt{A(x)} \mathbin{*} \texttt{snap\_A}(x) = s_A \mathrel{-\!\!\circledast} \texttt{B(y)} \mathbin{*} \texttt{snap\_B}(y) = f_{\text{wand}}(s_A) \mathbin{*} f_{\text{pledge}}(\underbrace{s_A}_{\texttt{\^{}x}}, \underbrace{f_{\text{wand}}(s_A)}_{\texttt{\^{}y}})) \tag{7.4}$$

Since s_A is the final value of `x` and $f_{\text{wand}}(s_A)$ is a final value of `y`, we can see that the pledge is an assertion that relates the prophecies of

the reborrowing and the reborrowed references. As a result, we can see that pledge-based specifications (excluding asserting pledges) can always be rewritten as prophetic specifications (assuming other parts of Prusti's specification language are unchanged). While the asserted part of the asserting pledge is also a function that relates the prophecies of the reborrowing and the reborrowed references, it belongs to the left-hand side of the magic wand and, therefore, is asserted and assumed in different locations than regular prophetic specifications. Therefore, modelling of asserting pledges in the prophetic approach would require adding an assert statement that performs the check before it is assumed that the current value of the reference is equal to its final value. However, the soundness proof of the prophetic encoding would need to be extended to show why inserting assert and assume statements at the corresponding locations is justified.

This shape of magic wands enables us to show how a pledge-based specification can be rewritten prophetically and how magic wands can be used to justify prophetic specifications. For example, when discussing Creusot, we showed the following snippet with the prophetic specification.

```
1  #[ensures(
2      (^pair).first == ^result &&
3      (^pair).second == (*pair).second
4  )]
5  fn reborrow_first(pair: &mut Pair) -> &i32 {
6      &mut pair.first
7  }
```

Figure 7.3 shows the encoding of the postcondition of the function from this snippet based on Assertion 7.3. Lines 5–12 contain the permission specification generated from the type information: line 5 contains the permissions of the returned reference, and lines 6–12 contain the magic wand for recovering the permission of the argument. The quantifier that binds `s_res`, the snapshot of the left-hand side of the magic wand, wraps the entire postcondition, including both permission and functional specifications. We use the symbol ∀ for the quantifier instead of Viper's `forall` to indicate that it is a logical quantifier. Putting the entire postcondition under a quantifier enables us to translate the prophetic specification straightforwardly as shown on lines 14–15: `s_res` corresponds to `^result`, $f_{\texttt{wand}}$`(s_res)` corresponds to `^pair`, and `old(snap<Pair>(pair.pointer)` corresponds to `*pair`. `Pair_first` and `Pair_second` are projection functions that given the snapshot of `Pair`, return the snapshot of `first` or `second` fields respectively.

The shown rewriting enables us to justify prophecies using magic wands. However, the approach is still limited to the fragment of Rust for which we can automatically generate the core proof, which is currently smaller than the one supported by Creusot. However, there is also a significant gap between Creusot and RustHornBelt, which may allow unsoundnesses to creep in when advanced Creusot features such as ghost code are combined with prophecies. One advantage of linking prophecies to magic wands via quantifiers is that these are well-understood concepts that provide the tool developer with an intuition of what uses of prophecies could be unsound.

```
method reborrow_first(pair: Ref) returns (res: Ref)
    requires ...
    ensures ∀ s_res :: (
        // Specification from types.
        acc(res.pointer, write) && i32(res.pointer) &&
        (
            i32(res.pointer) &&
            snap<i32>(res.pointer) == s_res
            --*
            Pair(old(pair.pointer)) &&
            snap<Pair>(old(pair.pointer)) == f_wand(s_res)
        )
        // User provided specification.
        Pair_first(f_wand(s_res)) == s_res &&
        Pair_second(f_wand(s_res)) == Pair_second(old(snap<Pair>(pair.pointer)))
    )
```

Figure 7.3: A prophetic encoding of `reborrow_first` postcondition with justification based on magic wands.

7.5.2.5 Prophecies and Pledges

In the previous subsubsection, we saw that pledges can be seen as assertions that relate prophecies, from which follows that all specifications using non-asserting pledges can be translated into prophetic ones. For the other direction, we know how to translate prophetic specifications into pledge-based ones for patterns that are supported by our approach. However, there are patterns supported by Creusot, which we do not know yet how to support in pledge-based specifications. One example is function `destroy` we showed above. We repeat the function in the following snippet.

```
#[ensures(b.resolve())]
fn destroy<T: Resolve>(b: Box<T>) {}
```

The challenge for specifying this function is that its specification needs to be generic over type `T`. Creusot's solution to this challenge relies on predicates that can contain prophetic assertions. When designing pledges, we did not consider them being used inside predicates. However, there also does not seem to be any fundamental reason why a solution similar to Dafny's two-state predicates [138] could not be adopted in this case. So far, we have not found an example that could be specified with prophecies but could not be specified with pledges extended with features already present in other mature verifiers such as Dafny. Therefore, it remains an open question whether specifications based on prophecies can always be translated into ones based on pledges. It is also an open question what prophetic syntax would be suitable for expressing asserting pledges.

[138]: The dafny-lang community (2023), *Dafny Reference Manual: Two-state lemmas and functions*

7.5.3 Explicit Backward Flow

This subsection presents work that, similarly to ours, attempted to reconstruct the backward flow for mutable borrows.

7.5.3.1 Aeneas

The prophetic purification approach we discussed in the previous subsection generates a non-executable logical encoding of a Rust program. Aeneas [139] set a goal to purify a Rust program into an executable functional program. Similarly to Electrolysis, this functional program then could be manually verified in a proof assistant; Aeneas's authors chose to target F*. Another important difference from prior work is that Aeneas does not rely on Rust's borrow checker; instead, the authors defined a borrow calculus that enables them to check whether capabilities and borrows are used correctly in a function. Similarly to Oxide [119] and our work, the borrow calculus is based on the notion of lifetimes as loans introduced by Polonius. Aeneas supports mutable, shared, and two-phase borrows and reborrows of mutable references. To translate mutable references into a pure functional program, Aeneas has to reconstruct the backward flow of values. Similarly to an optimised version of our encoding, they use different approaches for function-local (re-)borrows and reborrowing functions.

[139]: Ho et al. (2022), 'Aeneas: Rust verification by functional translation'

[119]: Weiss et al. (2019), 'Oxide: The Essence of Rust'

For function-local borrows, Aeneas precisely tracks what is borrowed by what and uses this information to propagate the new value to the borrowed place when the borrow ends. Aeneas achieves the required high precision by, instead of joining control flow branches (except at the end of the function body), duplicating the code after them. This simple solution works because all traces in a type-correct Rust function are guaranteed to end with the same capability state: they return the capabilities to result and borrowed parameters. The Aeneas approach is a much simpler way of addressing the challenges *C1* (handling the effects of PCS operations in backward capability flow)[6] and *C2* (conditional control flow). A complex part of solving *C1* is unifying incoming PCSs when joining the control flow. By not joining the branches, Aeneas completely avoids this complexity. *C2* is a direct consequence of joining the control flow: by joining the branches, our approach "compresses" the loan-dependency graph from a tree into a directed acyclic graph and, in this way, loses the precision, which we recover by introducing ghost variables. While Aeneas's solution to these challenges is more straightforward, it likely leads to a larger and potentially less efficient encoding. When restoring backward capability flow, Prusti branches only when the borrow checker does not have precise information which loan the expiring reference was borrowing. In contrast, Aeneas duplicates branches always. As we discussed in Subsection 7.5.1, the third alternative to whether to join the control flow is to hide the state behind an abstraction, which enables the use of a significantly simpler unification procedure. Similarly to our approach, Aeneas handles reassigned references (*C3*) by introducing new ghost variables that store the old value of a reference.

6: Aeneas manipulates the capabilities only when checking the program. During the verification, it is up to the user to apply the required definitions.

Aeneas encodes reborrowing functions by generating a forward and a backward function for each of them. A forward function corresponds to the original reborrowing function in which the values flow forward from the borrowed places into the reborrowing ones. In contrast, in the backward functions, the flow is reversed, enabling the caller to deduce how the changes via a reborrow affect the original borrow. When a borrow expires, Aeneas applies the backward function to obtain the new value of the borrow passed to the reborrowing function. A backward function is

effectively function f_{wand} we discussed above that enables obtaining the snapshot of the right-hand side of the magic wand given the snapshot of its left-hand side. A backward function is created from a Rust reborrowing function by adding parameters corresponding to returned reborrows, inverting the direction of all assignments, and changing the function to return the new values of the borrowed parameters. When a borrow expires, the backward function is called with the original arguments with which the reborrowing function was called and the final values of the reborrowing references. An important difference between backward functions and f_{wand} is that a backward function is generated for a specific reborrowing function while f_{wand} is defined when a magic wand is packaged. As a result, f_{wand}, similarly to closures, implicitly captures the state, such as the original arguments and what non-deterministic choices were made, which enables us to support non-deterministic reborrowing functions with no additional effort. Similarly, to function-local borrows, constructing reborrowing functions is much simpler than the Viper package statements needed to package a magic wand but is likely to lead to larger and potentially slower encoding because the reborrowing function contains the entire original function while the package statement generated by Prusti contains only the encoded PCS operations.

Understanding Aeneas also gives an interesting perspective on the prophetic encoding we discussed earlier. As we mentioned, Aeneas for each reborrowing function creates a backward function by inverting all assignments in the original function. The prophetic encoding could be understood as an optimisation of this encoding that, instead of duplicating the function, duplicates the value of each reference: the current value of the reference participates in the forward flow, and the final value of the reference participates in the backward flow. Since verifiers do not support inverted assignments, the prophetic approach uses `assume` statements instead, which enable achieving the same effect. Alternatively, Aeneas could be viewed as a way to justify the prophetic approach. For verification, the prophetic approach has three key advantages compared to the more explicit Aeneas approach. First, it generates only one function for each reborrowing function, which may lead to significant performance gains. Second, as we mentioned in Subsubsection 7.5.2.1, the backward flow generated by the prophetic encoding depends on the forward flow, which enables it to avoid the challenges of joining the branches without duplicating the control flow. Third, the prophetic encoding (and purely magic wand based encoding) with the same technique supports both function-local borrows and reborrowing functions. The key advantages of Aeneas are that it does not rely on the borrow checker for correctness and generates an executable pure functional program.

7.5.3.2 The Move Prover

Rust's success inspired many language designers to adopt ownership and borrowing. One such language is Move, a programming language for the Diem blockchain. Similarly to Rust, Move supports shared and mutable references, including functions returning references, but without reference-typed fields. Move has a strong focus on safety and includes a built-in specification language based on first-order logic. The critical limitation of the specification language is that it does not support

specifying functions that return borrows [140], which would require a construct similar to our pledges. However, it does support specifying functions that take reference-typed parameters. In addition to its native specification language, Move also has an official verifier called The Move Prover (MVP) [141, 142]. The second and significantly improved version [142] uses the strong guarantees provided by the Move type system to eliminate references when encoding a Move program with specifications into Boogie intermediate verification language [54]. MVP eliminates references by using a static analysis to construct a borrow graph that tracks relationships between references and enables it to deduce how the borrowed places should be updated when the references expire. This borrow graph looks pretty similar to the loan-dependency graph used by our work, with the key difference being that the borrow graph cannot represent reborrowing functions. MVP encodes the effects of borrows in one of two ways depending on whether the borrowing relationship is static or dynamic (our challenge C2). For the static case, MVP determines the originally borrowed variable and writes a new value into it when the borrow ends. It supports functions with mutable reference parameters by treating them as input-output parameters. The rewriting of the function signature is shown in the following snippet (we use Rust syntax for readability).

[140]: Association (2023), *Move Specification Language (version 1.4): Expressiveness*

[141]: Zhong et al. (2020), 'The Move Prover'

[142]: Dill et al. (2022), 'Fast and Reliable Formal Verification of Smart Contracts with the Move Prover'

[142]: Dill et al. (2022), 'Fast and Reliable Formal Verification of Smart Contracts with the Move Prover'

[54]: Barnett et al. (2005), 'Boogie: A Modular Reusable Verifier for Object-Oriented Programs'

```
// Function signature:
fn set_value(x: &mut i32) { /* ... */ }
// Becomes:
fn set_value(x: i32) -> i32 { /* ... */ }
```

For the dynamic case, MVP puts the borrowed value in a wrapper that stores the loan identifier, which is then used to determine where the new value needs to be written to propagate the effect of a borrow.

7.5.3.3 SPARK

Rust inspired not only the Move, but also SPARK developers. SPARK is a subset of Ada for mission-critical software. Originally, SPARK avoided the problems caused by aliasing by simply forbidding pointers. However, inspired by Rust's success, the SPARK developers added support for Ada pointers by adopting Rust-like ownership and borrowing [143, 144]. Similarly to SYMPLAR and the Move prover, SPARK relies on the type information to purify the program and avoid modelling the heap completely. Since the SPARK developers had to add support for pointers without changing the language to avoid breaking backward compatibility, the borrowing patterns supported by SPARK are fundamentally more limited than the ones supported by Rust. However, SPARK does support complex patterns like reborrowing in a loop. SPARK encodes borrows by using a *borrow relation* predicate (which is similar to manually encoded magic wands [32]) that maintains the relationship between the borrowing reference and the borrowed place. For specifying the functional behaviour of reborrows, SPARK authors initially adopted our pledges that allow the user to specify the overapproximate behaviour of the borrow relation on verification boundaries such as loop invariants. However, later they switched to using prophetic specifications [145].

[143]: Jaloyan et al. (2020), 'Verification of Programs with Pointers in SPARK'

[144]: Dross et al. (2020), 'Recursive Data Structures in SPARK'

[32]: Blom et al. (2015), 'Witnessing the elimination of magic wands'

[145]: SPARK Developers (2023), *SPARK's User's Guide: Annotation for Referring to a Value at the End of a Local Borrow*

7.5.4 Discussion and Future Work

In this section, we discussed multiple ways of modelling borrows and found that they all share a similar structure. Almost all models that support reborrowing functions model them using a construct similar to a closure: a magic wand, encoding of a closure with prophetic variables, or a function that takes the otherwise captured arguments as explicit parameters. The only exception is RustBelt, which uses a custom data structure inside its library to track borrows. Even though this data structure uses a magic wand, the meaning of this wand is different: instead of tracking a single borrow, this magic wand tracks *all* borrows created with the same lifetime. For modelling local borrows, the discussed approaches either used the same technique as for functions or explicitly reconstructed the backward flow with assignments. While the approaches have similar structures, they still have different strengths and weaknesses in addressing the challenges from Chapter 3. While RustBelt was created for manual verification in a proof assistant, we observed that its approach could also be useful for SMT-based verification: pushing the complexity of tracking borrowing behind an abstraction enables potentially simpler encoding of the core proof and avoids some branching in the verifier checking the core proof. When comparing Aeneas with the prophetic encoding, we observed that the prophetic encoding exploits the fact that it operates on purified values that the SMT solver can manipulate to push the complexity of reconstructing the backward flow into the solver. We also observed that prophetic encoding could be justified by using magic wands.

This thesis aims to enable incremental verification on two dimensions: enable programmers to focus on proving properties they care about without significant prior investments for safe code and allow a smooth transition to more powerful techniques for unsafe code. From our analysis of the related work, we can see that multiple models of borrows successfully target the first dimension where the key challenge of the model is to capture backward value flow. When moving to the second dimension, we need a model of borrows that addresses the following challenges:

1. Soundness with respect to operational semantics: does the model ensure that the aliasing rules imposed by the operational semantics (for example, Stacked Borrows or Tree Borrows) are respected?
2. Soundness with respect to the borrow checker: does the model assume only the properties guaranteed by the borrow checker and guarantee all the properties assumed by the borrow checker?
3. Completeness with respect to the borrow checker: can we automatically generate encoding for all borrowing patterns accepted by the borrow checker?
4. Functional correctness: does the model enable verifying functional correctness?
5. Automation: can the model be automated in an SMT-based verifier?

Existing models address only some of the challenges and often only partially. For example, our model targets challenges 3, 4, and 5 while RustBelt focused on 2 and RustHornBelt on 4. Currently, no single model would address all five challenges, and it is important future work to create such a model.

8 Conclusions of Part I

The goal of the thesis is to make verification incremental on two dimensions. In this part of the thesis, we focused on the first dimension: enabling incremental specification and verification of heap-manipulating code. We presented a specification and verification approach that leverages the type system of safe Rust. Unlike prior work on verifying functional properties of heap-manipulating programs, our approach enables incremental verification where the user can immediately focus on the properties that are important to them. We achieve this by using Rust type information to generate the core proof that captures the information about aliasing and side effects. Our evaluation shows that our technique can completely automatically generate core proofs for realistic Rust code, enabling users to focus on verifying interesting functional properties without learning complex logics. Our analysis of alternative approaches of modelling references revealed that while all models have a similar structure, some models enable pushing the complexity away from the Rust verifier (either into a once-and-for-all verified library or an SMT solver), which could be beneficial for SMT-based verification. In the analysis, we also found that multiple models successfully target our first dimension, but no model is suitable for SMT-based verification of unsafe code yet.

In the rest of the thesis, we will focus on the second dimension: enabling a smooth transition to more powerful verification methods, which are needed for verifying unsafe Rust. Since supporting all unsafe code is unfeasible, we start by analysing the most important patterns to support in Part II. Then, in Part III, we present our approach for verifying the safety and functional correctness of safe abstractions in the presence of panics.

Part II

Understanding How Programmers Use Unsafe Rust

In the previous part of the thesis, we looked into how the strong guarantees available in *safe* Rust enable significantly reducing the verification effort. In the remainder of the thesis, we focus on enabling verification of *unsafe* Rust. However, unsafe code is a vast topic, and the Rust community still discusses the concrete rules describing what unsafe should be allowed to do. Therefore, it is crucial to understand how programmers use unsafe Rust in practice to help the discussion and to focus the initiatives working on improving unsafe code. In this part of the thesis, based on [76], we aim to provide the necessary understanding.

[76]: Astrauskas et al. (2020), 'How do programmers use unsafe Rust?'

As mentioned in the introduction, the purpose of unsafe Rust is to enable programmers to express patterns that cannot be expressed in safe code. By leveraging unsafe Rust, it is possible to implement, for example, cyclic data structures, hardware abstraction layers, and lock-free algorithms – features that are difficult or impossible to realise in purely safe Rust. However, this added expressiveness comes at a price. The compiler cannot enforce the strong guarantees available in safe Rust; this enforcement becomes the developer's responsibility, entailing significant cognitive effort even for small pieces of unsafe Rust. Examples of the subtleties involved when taking this responsibility are found, for instance, in the discussions of Rust's unsafe code guidelines working group [146]. Importantly, as noted by [147], the correctness of unsafe code may rely on invariants that could be invalidated by *all* functions modifying the same struct fields; a thorough code review of whether a block of unsafe code is acceptable therefore is not limited to the code within the block itself, but has to include at least all code which could modify these fields.

[146]: Unsafe Code Guidelines Working Group (2020), *Project website*

[147]: Jung (2016), *The Scope of Unsafe*

```rust
let x = 17;
let r = &x; // borrow x
// cast reference r to raw pointer
let p = r as *const i32;
unsafe { assert!(*p == 17); }
```

(a) Example of Unsafe Blocks

```rust
// only safe to call with x == 17
unsafe fn foo(x : i32) { ... }
fn bar() { // safe abstraction
    unsafe{ foo(17); } // safe for 17
}
```

(b) Example of Unsafe Functions

Figure 8.1: Examples of unsafe Rust.

Rust provides two primary forms of unsafe code with different purposes. The first form allows programmers to bypass compiler checks. Its core feature is an *unsafe block* defining the scope in which these checks are disabled. For instance, in Figure 8.1a, an unsafe block is required to dereference the C-style raw pointer `p`. By default, it is assumed that such an unsafe block contained within, say, a struct method should use unsafe features in a way which is *encapsulated* from callers of the function and thus provides a *safe abstraction*: it is the responsibility of the function and not the client code that this code will always execute safely. Alternatively, one can explicitly declare an *unsafe function* (a function annotated with the `unsafe` keyword). This feature is intended to indicate that the responsibility for the correctness of the unsafe code in the function body lies at least partially with its *callers*[1]. Figure 8.1b shows the syntax for such an unsafe function `foo`, which can be called only from within unsafe blocks. As with any unsafe block, the call within the body of `bar` combined with the fact that `bar` is *not* an unsafe function indicates that the implementer of `bar` intends that this usage of unsafe Rust is safely encapsulated from `bar`'s callers: the developer promises

1: In addition, the entire body of an unsafe function is implicitly treated as if it were enclosed in an unsafe block. Since the publication of our work [76], a feature was added to the Rust compiler that allows the developers to turn off this behaviour.

that `foo(17)` preserves Rust's safety guarantees even though the compiler cannot enforce them.

The second main form of unsafe Rust involves Rust *traits*, which are comparable to Java interfaces. Instead of turning off compiler checks, an *unsafe trait declaration* acts as a documentation feature: it warns developers that all implementations of the trait are expected to satisfy some additional semantic properties such as preconditions, postconditions, or invariants that are not checked, neither at compile nor at run time; furthermore, these properties may be depended upon for the safety of client code *using* these traits. For *trait implementations*, `unsafe` takes the role of an annotation by which developers acknowledge their responsibility to respect the required semantic properties. We consider usages of `unsafe` in the above sense as a documentation feature since the compiler does not check the properties mentioned above. However, using `unsafe` in these cases is *not* optional. Adherence to all documented properties is crucial for upholding Rust's safety guarantees: clients calling an unsafe trait's functions can rely on these properties to argue the safety of their own code. Conversely, for a trait not declared as `unsafe`, all of its clients must ensure safety for *every possible safe implementation* of that trait – regardless of how much it deviates from its originally intended purpose.

Unsafe code must be used with care to retain Rust's strong guarantees. The commonly advocated practice is that programmers should, as far as possible, use unsafe Rust according to the following three basic principles, which aim to limit the necessary scope of code reviews (cf. [148], [149, Ch. 19], and prominent sources from the Rust community, *e.g.* [150–153]):

[148]: Developers (2023), *The Rustonomicon*
[149]: Klabnik et al. (2019), *The Rust Programming Language*
[150]: Rust Secure Code Working Group (2019), *Mission Statement of the Secure Code Working Group*
[151]: Matsakis (2016), *Unsafe abstractions*
[152]: Fuchsia Team (2020), *Fuchsia Documentation - Unsafe Code in Rust*
[153]: Jung et al. (2021), 'Safe systems programming in Rust'

1. Unsafe code should be used *sparingly*, in order to benefit from the guarantees inherently provided by safe Rust to the greatest extent possible.
2. Unsafe code blocks should be *straightforward* and *self-contained* to minimise the amount of code that developers have to vouch for, *e.g.* through manual reviews.
3. Unsafe code should be *well-encapsulated* behind *safe abstractions*, for example, by providing libraries that do not expose the usage of unsafe Rust (via public unsafe functions) to clients.

Ideally, these principles are implemented by encapsulating unsafe code inside carefully reviewed and tested libraries, providing safe abstractions whose clients can be written in safe Rust and need not be aware of the presence of unsafe code. Parts of the Rust language documentation [148, 149] claim that programmers can use unsafe code according to these three basic principles – a claim that we refer to as the *Rust hypothesis*.

[148]: Developers (2023), *The Rustonomicon*
[149]: Klabnik et al. (2019), *The Rust Programming Language*

Understanding how Rust programmers use unsafe code and, in particular, whether the Rust hypothesis holds is essential for users of the Rust language. It allows project managers to judge to what extent they can rely on Rust's promise to eliminate certain errors, developers to follow (evolving) best practices, testers to determine which properties to check for which parts of the codebase, library designers to identify further idioms of unsafe code that could be safely encapsulated, language designers to devise safe solutions for commonly-used unsafe idioms, and tool builders to support common idioms of unsafe code and their interaction with safe code.

In this part of the thesis, we present an empirical study on how unsafe code is used in practice. This study goes significantly beyond existing studies by analysing a large corpus of Rust projects to assess the validity of the Rust hypothesis and to classify the purpose of unsafe code. To answer these questions, we identify queries that can be answered by *automatically* inspecting the program's source code, its intermediate representation MIR, as well as type information provided by the Rust compiler. For instance, to assess how often unsafe code is used to implement custom concurrency primitives, we collect information about concurrency-related compiler intrinsics such as calls to compare-and-swap. To obtain a deeper understanding of the semantics and intent of unsafe code, we complement this automatically-collected data by manual code inspection.

Our results support the Rust hypothesis partially. Most unsafe code is simple and well-encapsulated behind safe abstractions. However, unsafe code is used quite extensively, especially to interoperate with other programming languages. Interoperability is by far the most prevalent motivation for using unsafe code, followed by implementations of data structures requiring complex sharing (via raw pointers or mutable global data). Other purposes, such as using unsafe concurrency features and applying unsafe to document semantic properties that are critical for upholding Rust's safety guarantees (the second form of unsafe code mentioned above), are less common.

Contributions. The contributions of this part are the following:

- A classification of the motivations for using unsafe code. We identify six main purposes for unsafe code. This classification serves as a basis for our empirical study but is also useful for the systematic documentation of unsafe code and tailoring techniques such as test case generation and program analysis towards specific use cases of unsafe code.
- An empirical study of how unsafe code is used in practice. Our study shows that code that is not concerned with interoperability typically adheres to the Rust hypothesis.
- A discussion of the implications for reasoning about Rust code (in code reviews or during verification).
- Our reusable open-source infrastructure and the analysed data is available online [154].

[154]: Qrates Team (2020), *Qrates artefact*

Outline. This part of the thesis is structured as follows. Chapter 9 classifies the main usages of unsafe code into six different categories. Chapter 10 summarises the methodology of our empirical study and states our core research questions. Chapter 11 presents our general framework for analysing Rust code. The results of our study are presented in Chapter 12 and discussed in detail in Chapter 13. We discuss threats to validity in Chapter 14, summarise related work in Chapter 15, and conclude in Chapter 16.

Motivations for Using Unsafe Code

9

In this chapter, we discuss six main reasons for using unsafe code. As discussed in the introduction of this part of the thesis, unsafe code is used either to work around restrictions of safe code or to document the existence of conditions that have to be upheld by the developer to ensure memory safety. The former can be further divided into two large groups: working around the limitations of the ownership type system and accessing inherently unsafe operations. In the following sections, we discuss six main reasons for using unsafe, grouped into the three groups mentioned above.

9.1 Overcoming Type System Restrictions

As was shown in Part I, the Rust type system provides strong guarantees that rule out many errors and enable simpler reasoning about Rust programs. However, this simpler reasoning comes at a cost, as demonstrated by two scenarios discussed in the following subsections.

9.1.1 Data Structures with Complex Sharing

In Part I, we explained that in safe Rust, each place can either be unique and allow mutation or shared and be immutable. While this design has clear benefits, it also has two important limitations. First, a direct consequence of this design is that in pure safe Rust, we can express only tree-shaped data structures. Therefore, to implement cyclic data structures such as graphs or doubly-linked lists, programmers need to directly use unsafe features such as raw pointers or safe abstractions such as the reference counting pointer `Rc` implemented using unsafe code internally. Second, many patterns require mutation via shared references; for example, synchronising state between two threads via a shared mutex. Rust's standard library provides multiple *interior mutable* types like `Mutex` that enable safe mutation via shared references. These types are implemented internally using `UnsafeCell`: an unsafe primitive provided by the Rust language that allows shared mutation.

9.1.2 Incompleteness Issues

In Chapter 3, we mentioned that Rust's borrow checker is improved over time to enable it to accept more examples. However, there are still many cases where the borrow checker is too restrictive. Figure 9.1 shows one such example from the RFC that proposed non-lexical lifetimes [87]. The example was planned to be accepted by the non-lexical lifetimes. However, the proposed analysis was too slow, and support for this example had to be removed [96]. The developer can get this example to compile on stable

[87]: Developers (2017), *The Rust RFC Book: nll*

[96]: Matsakis (2018), *MIR-based borrow check (NLL) status update*

Rust by casting away the problematic lifetime, as shown on lines 5–7.

```
1  fn get_default<'r, K: Hash + Eq + Copy, V: Default>(
2      map: &'r mut HashMap<K, V>,
3      key: K,
4  ) -> &'r mut V {
5      // let map2 = unsafe {
6      //      &mut *(map as *mut HashMap<K, V>)
7      // };
8      let map2 = &mut *map;
9      match map2.get_mut(&key) {
10         Some(value) => value,
11         None => {
12             map.insert(key, V::default());
13             map.get_mut(&key).unwrap()
14         }
15     }
16 }
```

Figure 9.1: Problem case #3 from the non-lexical lifetimes proposal [87] supported by Polonius but not by the non-lexical borrow checker, available in the stable version of Rust. Replacing line 8 with lines 5–7 makes the example compile.

9.2 Using Inherently Unsafe Operations

Rust is a systems programming language and, therefore, needs to support working with inherently unsafe operations, such as directly accessing hardware or calling functions written in other programming languages. Below, we list three scenarios that require inherently unsafe operations.

9.2.1 Foreign Functions

Since Rust is still a relatively new programming language, most Rust programs have to interact with libraries written in other languages. For example, Rust's standard library depends on the C standard library `libc` to provide an abstraction layer over the operating system primitives. Therefore, Rust provides the keyword `extern` that can be used to declare unsafe bindings to functions defined in libraries written in other programming languages. Working with foreign functions is inherently unsafe because the programmer needs to guarantee that the requirements of **both** programming languages are ensured. For example, if a pointer to a Rust object is passed to a foreign function, the programmer needs to ensure that the function does not violate potential aliasing or immutability requirements imposed on that object.

Similar requirements are imposed on inline assembly blocks that can be declared in Rust by using `asm!` macro.

9.2.2 Concurrency through Compiler Intrinsics

Rust's standard library provides a set of safe concurrency primitives such as mutexes and atomic integers. However, the existing primitives are not always suitable for the task. For example, the implementation of a safe primitive may rely on the existence of some operating system service, which is not available if a programmer is trying to implement

an operating system itself. Therefore, Rust exposes low-level LLVM concurrency intrinsics [155] that can be either used directly or to implement new safe concurrency primitives. These low-level concurrency primitives are defined on unsafe types such as raw pointers; therefore, using them is inherently unsafe.

[155]: LLVM Team (2020), *LLVM Atomic Instructions and Concurrency Guide*

9.2.3 Performance

When Rust cannot ensure memory safety statically, it guarantees it with runtime checks. For example, accessing an element of an array checks at runtime whether the index is within bounds. Since such checks can cause runtime overhead, programmers who want to achieve the highest possible performance may desire to disable them. Therefore, many data structures provide unsafe versions of the functions that do not perform the checks, thus pushing the responsibility of memory safety to the programmer who dares to use them. For example, an array element can be retrieved without performing the array check using method `get_unchecked`.

9.3 Emphasise Contracts and Invariants

The scenarios we discussed so far are all related to working around restrictions of safe Rust. However, the `unsafe` keyword also has another purpose: documenting the fact that there is some property that the compiler could not ensure and that the programmer has to ensure manually. In these cases, the `unsafe` keyword marks functions and traits that require special care. Unsafe functions indicate that the programmer *using* them has to be careful. Functions are typically marked as unsafe in two cases: if they have some precondition that needs to be satisfied to maintain memory safety or if they can be used to break the invariant of a safe abstraction on which some other code relies for memory safety. An example of the former case would be method `get_unchecked` of an array that performs indexing into the array and requires its client to guarantee that the provided index is valid. An example of the latter would be method `set_len` on `Vec` shown in the following snippet.

```
pub unsafe fn set_len(&mut self, new_len: usize) {
    debug_assert!(new_len <= self.capacity());
    self.len = new_len;
}
```

This method contains only safe code, but it could be used to break the invariant of `Vec` type that first `self.len` elements are valid. If this invariant is violated, then, for example, indexing into a vector becomes unsafe because the runtime check is not guaranteed to work anymore.

Traits are marked as unsafe to indicate that the programmer writing the type that *implements* the trait has to be careful. For example, a developer, by implementing `core::marker::Send` trait for their type, promises that it is safe to transfer the type from one thread to another.

The uses of unsafe presented in this chapter are based on anecdotal evidence. In the remainder of this part of the thesis, we systematically study how unsafe is used in real-world Rust code.

10 Methodology

Recall that the main goal of our study is to gain insights into how unsafe Rust is used in practice. To this end, we aim to answer the following high-level questions:

- Does the Rust hypothesis hold?
- What are the most prevalent use cases programmers have for using unsafe code?

In this chapter, we first refine the above questions into five research questions (RQs) that guide our search for a better understanding of unsafe Rust. After that, we break each of these questions down into *more-specific queries*[1] which allow us to infer answers for the original questions. We aim to answer each specific query fully automatically; the details of our approach are presented in Chapter 11. The results gathered by these queries are discussed in Chapter 12.

1: We highlight all specific queries in *green italic text*.

10.1 The Rust Hypothesis – Do Developers Use Unsafe Rust as Intended?

As introduced in the introduction of this part of the thesis, there are three widely-advocated basic principles for using unsafe Rust:

1. Unsafe code should be used *sparingly*.
2. Unsafe code should be *straightforward* and *self-contained*.
3. Unsafe code should be *well-encapsulated* behind *safe abstractions*.

The Rust hypothesis is that developers typically can and do follow the above principles. To check whether general Rust code in the wild supports this hypothesis, we investigate each principle through dedicated research questions. We first explore how widespread unsafe code is:

RQ 10.1 (Frequency) *How often does unsafe code appear explicitly in Rust crates?*

The first principle of the Rust hypothesis predicts that unsafe code will rarely appear in our dataset compared to its overall size. To verify this claim, we take a two-pronged approach: First, we identify every usage of unsafe Rust in our dataset and specifically count how many crates (i.e., binaries or libraries) include *any* unsafe code. That is, we count *how many crates contain at least one unsafe block, function, trait definition, or implementation*; all other crates contain only safe Rust. Even small pieces of unsafe code require a significant cognitive effort by developers: they need to be aware of their responsibility to guarantee safety rather than relying on the Rust compiler. Determining what fraction of crates is completely safe allows us to measure how frequently developers aim to

avoid this burden by sticking to safe Rust, as we would expect by the first principle.

Second, evaluating frequency solely based on whether a crate contains any unsafe code may lead to a coarse impression. To compensate, we complement this data by measuring the *relative* amount of unsafe code in each crate, i.e. *the ratio of the size of both unsafe blocks and unsafe function bodies to the total size of the crate.* We discuss how we specifically measure the size of code alongside our second research question below.

By the second principle, unsafe code should be *straightforward* – an admittedly subjective notion. To evaluate whether developers prefer to keep their unsafe blocks simple, we measure the *size* of each unsafe block as a proxy for its complexity (where smaller size means lower complexity):

RQ 10.2 (Size) *What is the size of unsafe blocks that programmers write?*

Attacking this question requires a reasonable means of quantifying the size of an unsafe block. An obvious candidate is to count the lines of Rust code in each unsafe block. However, lines of code are a weak indicator for a block's complexity because some features, such as closures and macros, might realise quite complex behaviour with a few lines of code. Moreover, different indentation and whitespace schemes would inadvertently bias such measurements.

To obtain a measure that is more robust against programming styles of different verbosity, we turn to the Rust compiler's intermediate CFG representation (MIR) in which many (potentially complex) high-level Rust constructs have already been translated to MIR instructions. Our next query thus checks *how many MIR statements does the compiler generate for unsafe blocks.* We also use this query to determine the total amount of unsafe code in a crate, to complement the binary query above.

RQ 10.3 (Self-containedness) *Is the behaviour of unsafe code dependent only on code in its own crate?*

Unsafe code that is compliant with the second principle of the Rust hypothesis should rarely reach out to other crates in order to keep manual reviews of unsafe code as simple as possible. In particular, unsafe code which relies on the functional behaviour of code from other crates may become vulnerable due to updates to its compilation dependencies.

A naïve query to evaluate this principle would count how many function calls in unsafe blocks have a call target outside the current crate – a low number then indicates a high degree of self-containedness. However, not all call targets are equal: The standard library crates, i.e., `std`, `core`, `alloc`, and `proc_macro`, are used heavily in a wide variety of projects. Since they are thoroughly reviewed, relying on the behaviour of these libraries is – while still technically in violation of self-containedness – arguably less problematic. Moreover, some crates are intended to provide low-level access to libraries written in other languages. For example, so-called "`-sys`" crates (whose name, by convention, ends with `-sys`) mirror the interface of C libraries [156]. This can be seen as a separation of concerns since multiple safe abstractions of the same C library may be sensible: the `-sys` gives unfettered access and other crates can implement safe abstractions

[156]: Rust Team (2019), *The Cargo Book*

on top of it. Naturally, such a design leads to dependencies between crates that are justified but in direct conflict to self-containedness.

We consider call targets in the above two categories separately as they amount to *expected and intentional* violations of self-containedness. Hence, we use a refined query that counts *how many function calls in unsafe blocks have a call target which is located in (1) its own crate, (2) a crate belonging to the standard library, (3) a `-sys` crate, or (4) any other crate*. Calls in category (4) point to likely *unintended* violations of the second principle.

Notice that the above query requires detailed knowledge about the targets of function calls. We evaluate it for *standard function calls* whose call targets can always be determined at compile time from the call expression alone. Since unsafe code should be as simple as possible to facilitate manual reviews, we expect unsafe blocks to contain only a few other function calls involving trait methods, closures, or function pointers, which require more manual effort from code reviewers as they have to trace down all possible implementations. To validate our expectation and as another proxy for simplicity, we measure *how many function calls in unsafe blocks and unsafe functions are (a) standard function calls, (b) calls of trait methods, or (c) calls of closures or function pointers*.

RQ 10.4 (Encapsulation) *Is unsafe code typically shielded from clients through safe abstractions?*

The third principle of the Rust hypothesis requires programmers to shield unsafe code from clients through safe abstractions. In other words, clients should be oblivious to the fact that unsafe code is used internally within a crate. Checking whether developers *succeed* in constructing suitable abstractions amounts to a difficult – if not impossible – task for automatic analyses, for instance, because they would have to check whether executions may exhibit data races. Therefore, we focus on the apparent *design intentions* of developers. That is, are they trying to hide their unsafe functions from other crates as much as possible?

To answer this question, we take a closer look at Rust's concept of visibility. Broadly speaking, there are three main notions: The default is private, meaning only visible within the current module – a user-defined collection of Rust items, such as functions, traits, etc. Alternatively, an item can be visible within either only the current crate or all crates.

We then count *how many unsafe functions are (1) declared private, (2) visible within their crate, and (3) visible to other crates*. Queries (1) and (2) cover all of the unsafe functions that comply with the third principle. Query (3) collects uncompliant cases, i.e., unsafe functions that are exposed to other crates.

10.2 Measuring the Unsafe World – How do Developers Use Unsafe Code?

We now take a closer look at possible reasons for developers to rely on unsafe code. Recall from Chapter 9 our classification of use cases for writing unsafe code ranging from overcoming aliasing restrictions over emphasising contracts and invariants to accessing lower abstraction

layers. Our goal is to identify code where these use cases are applied to answer the following:

RQ 10.5 (Motivation) *What are the most prevalent use cases for unsafe code?*

In the following paragraphs, we present specific queries for identifying individual use cases. All of these queries search for syntactic patterns that are characteristic for the use case in question. As such, they collect evidence for particular use cases rather than precisely capturing them. We intentionally do not attempt to find perfect characterisations because the results of our queries should be gathered *automatically*, possibly with manual follow-up efforts. As is common for most automated program analyses, we thus rely on approximations.

Data structures with Complex Sharing. Since Rust's ownership rules prevent complex sharing, some data structures are notoriously difficult – or even impossible – to implement in safe Rust. Although a few patterns, such as the interior mutability pattern, have evolved in the Rust community to deal with limitations of the ownership system, they all ultimately rely on using raw pointers[2]. Hence, we consider raw pointer dereferences as an indicator of a programmer's intention to bypass the ownership system to allow for complex sharing. To identify this use case, we thus collect *all functions that contain a dereference of a raw pointer*. We explore pointer dereferences at the function level rather than, *e.g.* studying individual unsafe blocks, to obtain unified results for both safe functions (which need to use unsafe blocks) and unsafe functions (which do not). Moreover, some developers advocate using many minimal unsafe blocks whereas others prefer fewer and larger ones. By phrasing our query at the function level, we keep it agnostic to these different styles.

2: Another common pattern is to use a vector-backed custom heap and implement the data structure using integer indices instead of pointers. However, this approach stays within safe Rust and is, consequently, outside the scope of our study.

The above query risks overcounting how frequently developers rely on unsafe code to implement complex data structures because raw pointers are, for example, also used to interoperate with C libraries. To obtain a more conservative estimate, we filter out usages of raw pointers for which we can identify different intentions: we do not count raw pointers appearing in structs that are equipped with attributes, such as `#[repr(C)]`, indicating that they are used for interoperability.

Incompleteness Issues. It is not generally possible to precisely identify all cases in which developers work around incompleteness issues of Rust's type and ownership system[3]. Therefore, we focus on unsafe functions for which the Rust documentation lists overcoming limitations of the compiler as a use case: we *collect all calls of unsafe functions involving explicit type casts*. For instance, both the "incredibly unsafe" function `transmute` and its close relative `transmute_copy` reinterpret the bits of a value as another type and are suggested by the Rust documentation to work around limitations of lifetimes, *e.g.* extending a lifetime or shortening an invariant lifetime [157].

3: If it were, the compiler could use the same analysis to prevent the incompleteness in the first place!

[157]: Rust Team (2020), *Rust Documentation of Transmute*

Emphasise Contracts and Invariants. Recall that the `unsafe` keyword in Rust may also serve as a documentation feature when attached to functions or traits. To understand whether developers have used `unsafe` to document contracts and invariants, we run two queries: First, we

search for unsafe functions whose body contains only safe Rust code. Since there is no technical reason why these functions need to be declared as `unsafe`, we expect that any such functions are declared unsafe either accidentally, or in order to document some implicit contract or invariant that is critical for upholding safety (for example, of unsafe code in the same module). Second, we *count the number of both safe and unsafe traits declared*. For traits, `unsafe` is always a documentation feature. Based on the low number of unsafe traits in the Rust standard library (at the time of our publication [76], there were only eleven that are listed in Figure 10.1), we expect to find only a few unsafe trait declarations.

[76]: Astrauskas et al. (2020), 'How do programmers use unsafe Rust?'

Trait	Visibility
core.alloc.AllocRef	Public
core.alloc.GlobalAlloc	Public
core.array.FixedSizeArray	Public
core.iter.adapters.zip.TrustedRandomAccess	Crate
core.iter.traits.marker.TrustedLen	Public
core.marker.Freeze	Crate
core.marker.Send	Public
core.marker.Sync	Public
core.panic.BoxMeUp	Public
core.str.pattern.ReverseSearcher	Public
core.str.pattern.Searcher	Public

Figure 10.1: A list of unsafe traits in the Rust standard library.

Concurrency through Compiler Intrinsics. The Rust compiler provides access to low-level concurrency intrinsics through dedicated unsafe functions. To measure how frequently developers rely on them, we collect a list of unsafe functions wrapping concurrency intrinsics in the `std::intrinsics` module. We then run a query *collecting all unsafe blocks that call one of the collected functions.*

Foreign Functions. Interoperability with other languages, *e.g.* accessing C/C++ libraries, is mentioned by [158] as a frequent use case for writing unsafe code. To identify code that is likely to interoperate with foreign code, we exploit the following observations in our queries:

[158]: Qin et al. (2020), 'Understanding memory and thread safety practices and issues in real-world Rust programs'

- The attribute `#[repr(C)]` ensures that the memory layout of a type is interoperable with the C programming language; unsafe code that relies on such types is thus likely to exchange data with foreign code. Hence, we count *how many types are equipped with* `#[repr(C)]`.
- The name of crates that wrap C system libraries ends – by convention – with the suffix `-sys`. We thus classify *all* `-sys` *crates* as belonging to this use case.
- Rust allows functions to be declared `extern` with a custom Application Binary Interface (ABI) such that foreign code with the specified ABI can call them. Consequently, we determine *how many unsafe functions are declared with a foreign ABI*.
- Finally, we search for *usages of inline assembly* via the `asm!` macro. A closer analysis of the motivation for using inline assembly, *e.g.* low-level optimisations or interacting with hardware devices, is subject to manual inspection.

Performance. We consider two standard optimisations that use unsafe code to boost performance: First, we *search for unchecked functions* (those with "unchecked" in their name). By convention, these functions sacrifice run-time checks for better performance; they are consequently unsafe. The standard library, for example, adheres to this naming convention and frequently provides both a safe (checked) and an unsafe (unchecked) variant of the same functionality.

Second, we determine whether *unsafe blocks contain the special union type* `MaybeUninit`. The Rust documentation describes this type as a highly unsafe variant of optional types that avoids any safety checks at run time (cf. [159]). In fact, it may reintroduce dangling references as developers may use it to define uninitialised references in unsafe blocks.

[159]: Rust community (2018), *Rust: The Reference*

A Framework for Querying Rust Crates

11

To automatically evaluate the queries presented in the previous chapter, we developed a framework – called Qrates – for *Q*uerying *R*ust Cr*ates* for a large dataset of publicly available Rust code. It is inspired by both an idea proposed by [160] (one of the project leaders of the Rust compiler team) for adding Datalog output to the Rust compiler and commercial tools such as [161] for analysing other programming languages. In this chapter, we briefly outline the main components of Qrates.

[160]: Matsakis (2017), *Project idea: datalog output from rustc*

[161]: Code QL (2020), *Website of Code QL*

Intuitively, our framework works as follows: We first create a database of Rust crates enriched with metadata, *e.g.* the crates' origin and whether it compiles to a binary or a library. After that, we run a set of queries on the database to extract statistical data that enables answering the questions from the previous chapter. Answering a research question then amounts to combining the data gathered by one or more queries.

To construct the database, we implemented a plugin for the Rust compiler that extracts information such as the program's CFG as well as type information for a given Rust crate during compilation and stores it in a local database. Extracting data through a compiler plugin has immediate benefits: It integrates well into Rust's existing build infrastructure, including its widely-used package manager `cargo`. Moreover, we gain access to the analysed code in various intermediate formats, such as the CFG representation (MIR) in which all types and static function calls are fully resolved. Another consequence is that we consider only crates that compile successfully.

For the compilation, Qrates is based on the Rustwide library, which was mainly developed by [162] from the Rust infrastructure team. Its original purpose is to run ecosystem-wide tests of the Rust compiler. Rustwide provides a Docker image with the necessary dependencies needed to compile most publicly available crates.

[162]: Albini (2020), *The Rustwide library*

After creating local databases for all crates of interest, Qrates merges them into one comprehensive database with additional cross-link information. Once the final database has been constructed, the user can query it to gather statistics. Prime examples of supported queries include – but are not limited to – all *highlighted queries* from Chapter 10. In particular, Qrates can count how often a specific Rust feature, say a call to a function of interest, appears in (unsafe) blocks, functions, entire crates, or across all crates in the database.

It is noteworthy that, since crates may specify fixed versions for their dependencies, our database may contain different versions of the same crate. To prevent double-counting, our queries report results only for a single version of each crate (for this study, we chose the latest version). We still keep all crate versions in the database, because some queries are concerned with dependencies. For example, when analysing the call

targets of a function, some of them may be defined in older versions of a dependency.

Furthermore, our database keeps precise information about the origin of the analysed code in order to avoid counting the same code, *e.g.* statically linked libraries, twice.

We note that all queries concerning functions are based on Rust's intermediate CFG representation (MIR). On the one hand, basing on MIR gives us access to the type information, which is not available in the AST. On the other hand, this means the compiler has already performed various transformations, such as macro expansion, desugaring pattern-match constructs, and generating code for `#[derive(...)]` attributes. Consequently, some care is needed to check whether a piece of code can be attributed to the programmer, *e.g.* by considering the unsafe block check mode provided by the compiler; otherwise, we risk overcounting features that are predominantly introduced by the Rust compiler.

12 Empirical Results

In this chapter, we present the results automatically gathered by our queries, complemented by some manual inspections, and provide answers to the research questions from Chapter 10. We first discuss our data sets, then the experimental setup, and finally the results for each research question. We provide a detailed discussion of our findings in Chapter 13.

12.1 Datasets and Experimental Setup

We evaluated our queries on a dataset that comprises the most-recent version (as of 2020-01-14) of all 34445 *packages* published on the central Rust repository crates.io. The implementation of a package can be composed of multiple *crates*, one of which is usually primary and determines the name of the package. We excluded 5,459 packages (15.8%) whose most recent version did not successfully compile. For packages with conditional compilation features, we used the default flags specified in the manifest. In cases where a package failed to compile with the default flags, but succeeded with different ones (when compiled as a dependency of another package) we selected a random build for analysis. As a result, our dataset consists of 31867 crates. Most of these crates are compiled to Rust libraries (76.0%), or binaries (20.0%). The remaining crates are procedural macros (4.0%).

Our experiments were conducted on a computer equipped with an Intel Xeon E5-4627 processor (3.30GHz, 16 cores), 252 GB of RAM, running Ubuntu 16.04.6 as an operating system and version `nightly-2020-02-03` of the Rust compiler. Since our experiments do not depend on timings or performance, it should be straightforward to reproduce our results on different hardware. We collected all of our results in Jupyter notebooks for follow-up analyses using Python [163]; these are also available online [154].

[163]: Jupyter Team (2020), *The Jupyter project*

[154]: Qrates Team (2020), *Qrates artefact*

12.2 The Rust Hypothesis – Do Developers Use Unsafe Rust as Intended?

We first answer the research questions related to the Rust hypothesis, i.e., the claim that developers typically use unsafe Rust (1) sparingly and in a way such that its behaviour is (2) both straightforward and self-contained, and (3) well-encapsulated behind safe abstractions.

Table 12.1: Rust crates with and without any unsafe code grouped by feature. A crate may contain multiple unsafe features.

Unsafe Feature	**#crates**	%
None	24,360	76.4
Some	7,507	23.6
Blocks	6,414	20.1
Function Declarations	4,287	13.5
Trait Implementations	1,591	5.0
Trait Declarations	280	0.9

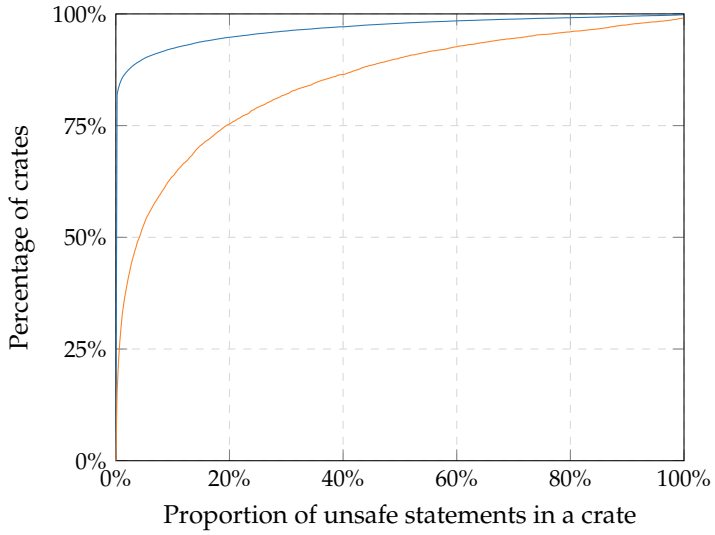


Figure 12.1: The cumulative proportion of statements in unsafe blocks and functions in all crates (blue) and in crates that have at least one such statement (orange).

12.2.1 RQ 10.1 (Frequency): How often does unsafe code appear explicitly in Rust crates?

Table 12.1 shows in both absolute and relative numbers how many crates contain unsafe code, and which unsafe features they use (our first query), while Figure 12.1 shows the relative amount of statements in unsafe blocks and functions in (1) all crates and, for readability, (2) crates that contain at least one unsafe statement (our second query)[1]. The majority of crates (76.4%) contain no unsafe features at all. Even in most crates that do contain unsafe blocks or functions, only a small fraction of the code is unsafe: for 92.3% of all crates, the unsafe statement ratio is at most 10%, i.e., up to 10% of the codebase consists of unsafe blocks and unsafe functions. However, with 21.3% of crates containing *some* unsafe statements and – out of those crates – 24.6% having an unsafe statement ratio of at least 20%, we cannot claim that developers use unsafe Rust sparingly, i.e., *they do not always follow the first principle* of the Rust hypothesis.

1: Notice that Figure 12.1 does not account for unsafe traits. Consequently, the percentage of crates without any unsafe blocks or functions (78.7%) is slightly larger than the percentage of entirely safe crates (76.4%).

Nevertheless, if we compare our results with the ones from [164], we can see that they report *higher* percentages of unsafe crates across *all* features in their experiments. Their experiments are based on a 16 months *older* snapshot (from September 2018) of the central Rust repository crates.io. In the meantime, more than 10,000 crates have been added to the repository and, in particular, the percentage of unsafe crates dropped from 29% to 23.6%. This finding differs from [164], who observed that the amount of unsafe code in the most downloaded crates slightly *increased* over 10 months. One possible explanation for these observations is that the most

[164]: Evans et al. (2020), 'Is Rust used safely by software developers?'

[164]: Evans et al. (2020), 'Is Rust used safely by software developers?'

downloaded crates provide the necessary extensions to the language or standard library (for example, an efficient random number generator) that cannot be implemented in safe code and, therefore, the amount of unsafe in the most popular crates does not change while a significant portion of the newly added crates are application code that does not need to use unsafe. Another possible explanation of these observations is that they may reflect concerted efforts within the Rust community to reduce the overall usage of unsafe code, such as the "Rust Safety Dance" project by the security working group of the [150].

[150]: Rust Secure Code Working Group (2019), *Mission Statement of the Secure Code Working Group*

From Table 12.1, we can also see that the most used unsafe features are unsafe blocks and unsafe function declarations. Both unsafe trait declarations and unsafe trait implementations are rare – the former are found in less than 1% of all crates; given that implementations generally do have interesting contracts and invariants, this low number suggests that programmers do not find it useful to highlight those via the `unsafe` keyword.

12.2.2 RQ 10.2 (Size): What is the size of unsafe blocks that programmers write?

Recall from Section 10.1 that we measure the number of MIR statements the compiler generates for an unsafe block, #MIR for short, as a proxy for its code complexity. Figure 12.3 shows the cumulative distribution of MIR statements generated for each unsafe block, cropped at 100 #MIR to improve readability; the depicted graph covers 97.4% of all unsafe blocks. The size of most blocks is quite small: 75% of all unsafe blocks comprise at most 21 #MIR, which almost coincides with the mean of 22.0 #MIR. For comparison, the compiler already generates 12 MIR statements – more than the overall median of 10 – for the small unsafe block shown in Figure 12.2. Upon closer manual inspection, there is a significant share, namely 14.4%, of tiny unsafe blocks that either wrap an expression (without function calls) or call a single unsafe function whose arguments were computed before the unsafe block. Conversely, there is a small number (78 or 0.02%) of huge outliers whose size ranges from 2,000 to 21,306 #MIR. Most of these unsafe blocks are automatically generated, *e.g.* through user-written macros or external scripts.

```
1 fn nop(x: u32) {}
2
3 fn main() {
4     let x = 42;
5     unsafe {
6         nop(x);
7         nop(x);
8     }
9 }
```

Figure 12.2: Unsafe block of size 12 #MIR, more than the overall median of 10.

In summary, the size of unsafe blocks is typically small. Assuming that the number of MIR statements adequately approximates the complexity of unsafe blocks, we conclude that *most developers keep their unsafe blocks simple*, which supports the second principle of the Rust hypothesis.

12.2.3 RQ 10.3 (Self-containedness): Is the behaviour of unsafe code dependent only on code in its own crate?

In our dataset, we have in total 772,228 calls in unsafe blocks. As shown in Figure 12.4, more than three-quarters of them are calls to standard functions while calls to trait methods and calls to closures and function pointers are only 18.0% and 3.6%, respectively. If we compare with the distribution of the entire dataset (shown in Figure 12.5) that includes

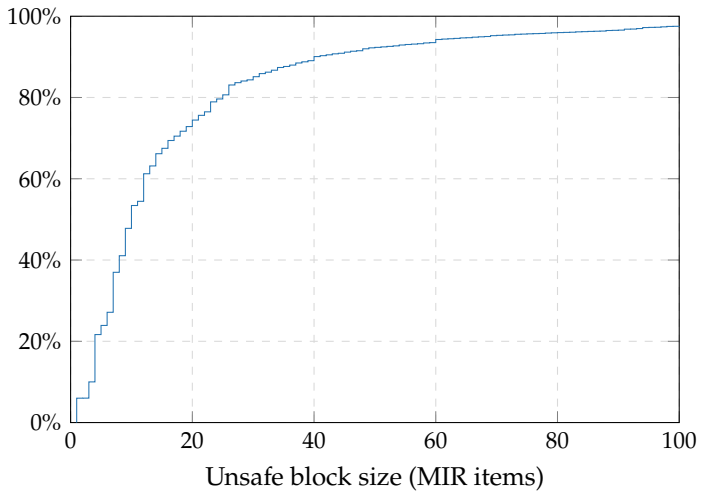


Figure 12.3: Cumulative distribution of the size of unsafe blocks, cropped at 100 MIR instructions (#MIR).

also the compiler-generated code[2], we can see that calls in unsafe blocks have a significantly larger proportion of standard function calls (78.3% in unsafe blocks vs. 56.3% in the entire dataset) and, correspondingly, a significantly smaller proportion of calls to trait methods (18.0% in unsafe blocks vs. 42.9% in the entire dataset). Even though 18.0% is still a substantial proportion, the relatively low number confirms our expectation that developers avoid those calls in unsafe blocks to keep the code simpler and more self-contained. In particular, a manual inspection of 100 randomly-selected calls to trait methods in unsafe blocks revealed that in 82 cases, the call target can be determined statically, just by looking at the function containing the unsafe block. Therefore, these calls do not add substantially to the complexity of the unsafe code.

2: The compiler-generated unsafe code contains only standard function calls.

To understand why the proportion of calls to closures and function pointers is larger in unsafe blocks than in all code (3.6% vs. 0.7%), we manually looked into several examples and observed three main patterns. The first one is parameterising the behaviour of unsafe code with a closure that is passed in as an argument to the safe wrapper. A typical example of this pattern is the `sort_by` function on the primitive type slice, which takes a comparison function as an argument. The second pattern is using function pointers to call functions from dynamically-loaded libraries (which can be done only from within an unsafe block), and the third pattern is using function pointers to implement callbacks to system libraries.

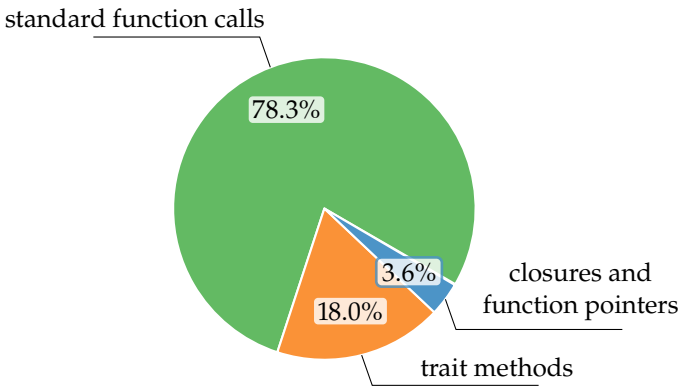


Figure 12.4: Distribution of types of calls in unsafe blocks.

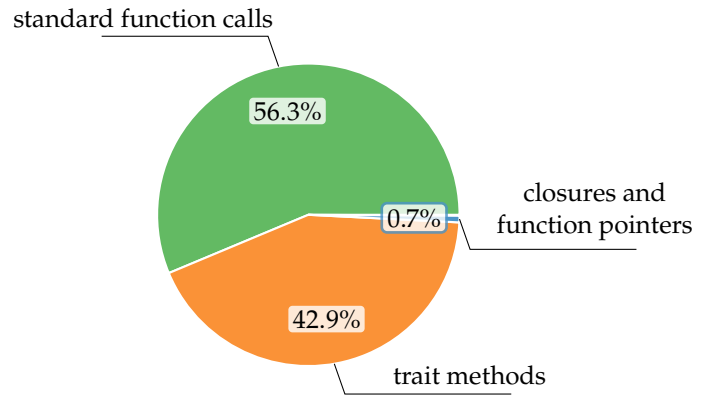


Figure 12.5: Distribution of types of calls in the entire dataset (including the compiler-generated code).

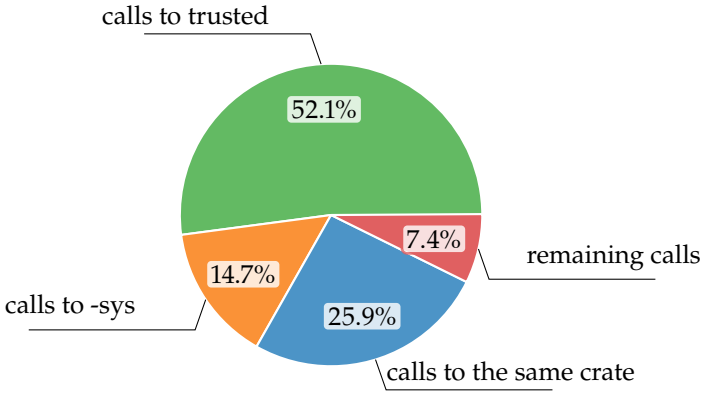


Figure 12.6: The call targets of standard calls.

Besides the different forms of calls, we analysed where the call targets of standard calls are implemented, to assess the extent to which unsafe code is self-contained. Figure 12.6 illustrates the distribution of targets of calls to standard functions, grouped into four categories. The majority (52.1%) of all function calls are into the standard library; as argued in Section 10.1, we consider such function calls only a minor violation of self-containedness. Most of the remaining calls (25.9%) stay within the same crate. Only 7.4% of all calls targets are located in other crates. We manually inspected a few of these crates and found that most of them, similarly to `-sys` crates, encapsulate system libraries. So in summary, for codebases written purely within Rust, very few calls actually violate the self-containedness principle of the Rust hypothesis.

We also analysed how the distribution of call targets changes when we consider only calls to *unsafe* functions (which, as we will see in Section 12.2.5, is the most common motivation for using an unsafe block). As shown in Figure 12.7, the share of calls to `-sys` crates is significantly higher, whereas the share of calls that stay within the same crate remains almost the same. This suggests that developers hesitate to call *unsafe* functions that reach out to other crates unless they explicitly wish to interact with system libraries.

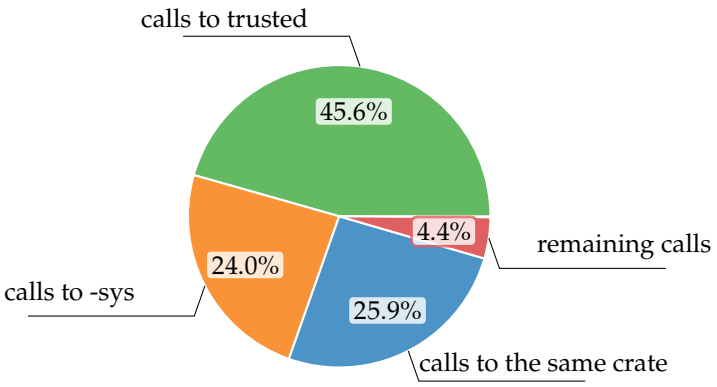


Figure 12.7: The call targets of standard calls when only calls to *unsafe* functions are considered.

12.2.4 RQ 10.4 (Encapsulation): Is unsafe code typically shielded from clients through safe abstractions?

In Table 12.2, we classify unsafe functions based on their visibility, which may be private (only callable from this submodule), visible within a restricted module, or public. We use visibility as an indication for the programmer's intention to encapsulate unsafe implementations from client code. Our metric is based on the information in a function's declaration, and does not differentiate between using the public modifier to enable calls from other submodules within the *same* crate, or from different crates entirely. For the latter, the functions would also need to be declared visible in the root module of the crate, which is a separate decision. Note that as soon as a function is declared public, its call-sites are in general unknown and may change over time. We removed from this analysis all unsafe trait methods (only 687, 0.1% of all unsafe functions), as their visibility is implicit.

We observed that only 12.0% of unsafe functions are not visible to arbitrary code (say, in other crates) because they are either private or restricted to a module. The vast majority (88.0%) of unsafe functions are declared to be public. At first glance, this suggests that programmers rarely shield their unsafe code from clients. To investigate this, we also studied the *ratio* of public unsafe functions compared to all unsafe functions *in each crate* containing at least one unsafe function. That is, a ratio close to 1 indicates that a crate poorly encapsulates unsafe functions (as all of these functions are public). Conversely, a ratio close to 0 indicates strong encapsulation as almost all unsafe functions are not publicly visible. The results are depicted in Figure 12.8. Based on this metric, we get a clearer picture: most crates (78.5%) have either all or none of their unsafe functions declared public. In particular, 34.7% of all crates seem to be well encapsulated: they declare unsafe functions but none of them are visible from the outside.

Moreover, 43.8% of crates declare all of their unsafe functions public; more precisely, these crates contain 274,434 (49.2%) unsafe functions[3].

3: All percentages below are with respect to this number of unsafe functions.

Continuing our investigation, we queried how many public unsafe functions within these crates provide raw bindings to system libraries, as it is common practice to make these bindings public. At first, we checked the ABIs of the functions. We found that 163,650 (59.6%) have foreign item ABI, which means that they are bindings of foreign items (most

Visibility	# functions	%
Private	65,230	11.7
Restricted	1,535	0.3
Public	489,928	88.0

Table 12.2: Visibility of unsafe function definitions (excluding trait methods).

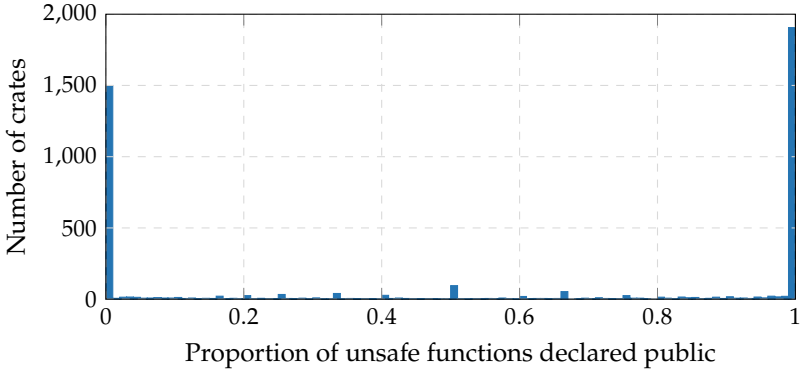


Figure 12.8: The number of crates for each ratio of public unsafe functions compared to all unsafe functions.

likely C functions). We also found that 571 functions (0.2%) have C ABI, which means that they can be called from C code and, therefore, it makes sense to have them public. The vast majority of the remaining functions (110,212 or 40.2%) have Rust ABI and, therefore, it is hard to automatically tell whether they are bindings or not. Therefore, we checked the meta information of the crates that contain these functions and found that 9,642 (3.5%) are assigned to categories that indicate them as crates that wrap system libraries and 49,363 (18.0%) are assigned to categories related to embedded programming. Finally, we manually reviewed 30 crates from the remaining list that have most unsafe functions (in total 41,063 functions or 15.0%) and found that they either provide APIs to microcontrollers or OpenGL bindings. After our analysis we are left with only 10,148 functions (3.7%) that are public and which may not be from the crates that provide bindings.

To summarise, even though the large number of crates that provide bindings make it hard to draw definitive conclusions, it seems that Rust programmers at least attempt to not expose unsafe functions to their clients because we found that 34.7% of all crates using unsafe functions do not declare a single public one; conversely crates that declare a lot of public unsafe functions can often be attributed to cases where encapsulation is not intended.

12.2.5 RQ 10.5 (Motivation): What are the most prevalent use cases for unsafe code?

To answer this question, we first identified a set of independent reasons for which the compiler requires unsafe blocks and functions to be declared unsafe. We extracted these reasons from the source code of the Rust compiler [165]; they are therefore complete. Then, we collected which reasons apply to the implementation of each function (either the body of an unsafe function or the unsafe blocks inside a safe function). The results are summarised in Table 12.3. As the data shows, calls to unsafe

[165]: Rust Team (2020), *File: check_unsafety.rs*

functions are by far the main reason why unsafe code is unsafe, followed by dereferencing raw pointers[4].

4: While all reasons mentioned in the table *should* require to use unsafe code, the reason "borrow of a packed field" does not due to a compiler bug; see https://github.com/rust-lang/rust/issues/27060 for details.

A block or function may be unsafe for multiple reasons. We found that for 83.5% of all functions that have at least one reason of unsafety, calling unsafe functions is the *only* reason of unsafety. In 93.6% of the functions, unsafety is due only to the first 2 entries of the table, and that in 99.4% of the functions, all reasons for unsafety are among the top 3 entries of the table. This data will enable, for instance, developers of static analysers and verification tools for Rust to prioritise which features of unsafe Rust code should be supported in their tools.

The classification in Table 12.3 indicates why the compiler requires a block or function to be declared unsafe, but does not explain why programmers chose to use these unsafe features. To understand their motivation, we studied the prevalence of the specific use cases proposed in Section 10.2, as we discuss next.

12.2.5.1 Data Structures with Complex Sharing

To assess how often unsafe code is used to implement data structures with mutable aliases, we measured how many functions dereference a raw pointer and how many structs have raw pointer fields. Our database contains 7,385,690 function definitions, out of which only 46,263 (0.6%) dereference a raw pointer in their implementation. In particular, this is done in 9,273 out of a total of 557,380 unsafe functions (1.7%), in 35,761 out of 6,221,053 safe functions (0.6%), and in 1,229 of 607,257 closure declarations (0.2%). Overall, 7.0% of all crates have unsafe code that dereferences at least one raw pointer. Regarding the raw pointer fields, we found that 6.6% of all crates have types with raw pointer fields. After filtering out raw pointers in structs whose attributes indicate that they are likely intended for interoperability, this number reduces to 4.6% of all crates.

Given that the restriction to tree-shaped data structures seems to be a major limitation of safe Rust, the number of raw-pointer dereferences is sizeable, yet rather low. It seems that raw pointers are rarely used to implement more complex data structures. A possible explanation is that sharing occurs especially in standard data structures, such as cyclic lists, doubly-linked lists, smart pointers, and trees with parent-pointers. However, such data structures are provided by the standard library and, thus, do not often occur in the form of custom implementations in application code. Another possible explanation is that Rust programmers choose designs that can be implemented without mutable sharing in safe Rust rather than resorting to unsafe manipulation of raw pointers. Finally, developers may circumvent the ownership system while staying within safe Rust by relying on custom vector-backed heaps and using less constrained integer indices instead of references.

Even though not necessarily related to data structures, the use of mutable static variables (the third most-prevalent reason for unsafety in Table 12.3) is also a form of sharing because global data can be accessed and mutated by multiple functions – behaviour that one could alternatively achieve via aliased raw pointers.

Table 12.3: Reasons why blocks and functions need to be declared unsafe, aggregated on the function level. For a specific function, there can be more than one reason why it needs to be declared as unsafe.

Reason	#functions	%
call to unsafe function	403,307	89.76
dereference of raw pointer	46,263	10.30
use of mutable static variable	25,888	5.76
access to union field	1,426	0.32
use of extern static variable	548	0.12
use of inline assembly	493	0.11
borrow of packed field	326	0.07
initialising type with `rustc_layout_scalar_valid_range` attr	41	0.0
assignment to non-Copy union field	3	0.0
pointer operation (in a `const` function)	2	0.0
cast of pointer to int (in a `const` function)	1	0.0
borrow of layout-constrained field with interior mutability	0	0.0
mutation of layout-constrained field	0	0.0

12.2.5.2 Incompleteness Issues

In safe Rust, the type system is able to prevent, for instance, usage of references whose target *might* have been deallocated in some preceding conditional branch. These checks are a form of static analysis subject to incompleteness, as they conservatively reject some otherwise valid programs. Since incompleteness cases cannot be precisely identified automatically, we instead measured the calls to unsafe functions involving explicit type casts (`transmute` and `copy_transmute`) as a proxy to assess how frequently programmers need to work around incompleteness issues of the type checker. We found that 28,469 out of 319,600 unsafe blocks (8.9%) call a `transmute` function, and that 4.5% of all crates contain at least one call to a `transmute` function. Interestingly, only 1.7% of all crates have more than 3 unsafe blocks with a call to those functions. This confirms our expectation that calls to `transmute` functions, including workarounds for incompletenesses of the compiler, are rare, and that when crates have to make those calls, they use them sparingly. However, there still exist some outliers that make thousands of calls to `transmute`. After manual inspection, we found these crates to contain code generated by scripts or recursive macros, which explains the anomaly.

12.2.5.3 Emphasise Contracts and Invariants

To check how prevalent `unsafe` is used as a documentation feature, our queries gathered data for unsafe traits and unsafe functions with safe implementations.

We found 1,093 unsafe trait declarations, which amounts to only 2.5% of all trait declarations. We conclude that developers rarely use unsafe traits, possibly because (1) the compiler never forces them to and (2) there are no decisive guidelines for using unsafe traits. Instead, the Rust documentation seems to discourage developers from frequently declaring traits as unsafe [148, Ch. 1.1]. Notably, we observed that a few developers embraced unsafe traits enthusiastically: Five crates are responsible for 40.4% of all unsafe trait declarations.

[148]: Developers (2023), *The Rustonomicon*

Regarding unsafe functions, our experiments yield that 36.1% of all unsafe functions are written in completely safe Rust. We found this number surprisingly high. After all, the compiler does not force developers to declare such functions as `unsafe` – in contrast to other unsafe features. Rather, a programmer has to intentionally type an additional keyword. Hence, at first glance, it seemed that developers frequently were using unsafe functions for the same reason as unsafe traits: to document properties, *e.g.* invariants that are potentially critical for upholding Rust's safety guarantees.

To find explanations for the surprisingly high number of unsafe functions with safe implementations, we performed manual inspections: We manually inspected the ten crates with the highest overall count of unsafe functions with completely safe bodies. All of these crates are automatically generated to provide peripheral access to various microcontrollers. The involved code generation seems to be conservative and frequently use unsafe functions even if it does not have to. Moreover, we randomly selected a few additional unsafe functions with safe bodies for manual inspection. Among these functions, a few were equipped with explicitly documented invariants. Other functions seem to be marked as unsafe primarily for legacy reasons. So these extra inspections suggest that most of these functions are declared unsafe almost accidentally, rather than to intentionally highlight contracts and invariants. Therefore, the discrepancy between unsafe traits and unsafe functions seems much smaller than the initial numbers suggest.

Overall, there is no clear evidence that unsafe functions are frequently used for documenting contracts and invariants, except when those contracts overlap with (and perhaps protect against) situations in which unsafe Rust features are used in the functions' implementations.

12.2.5.4 Concurrency through Compiler Intrinsics

To measure to what extent compiler intrinsics are used to implement fine-grained concurrency, we collected all unsafe blocks that call one of the 89 compiler concurrency intrinsics defined in the `core::intrinsics` module, or their re-export from `std::intrinsics`. These functions are used by only 4 crates in our dataset: `core` (8 calls), `compiler_builtins` (7 calls), `rs_lockfree` (6 calls), and `hsa` (1 call). We thus conclude that compiler intrinsics are not widely used, probably because they are still marked as *experimental* and require a nightly version of the compiler.

Interestingly, while analysing the results, we found that the concurrency intrinsics exposed by the `compiler_builtins` crate are incorrectly *not* marked as `unsafe` even though they internally dereference a raw pointer passed as parameter. It is, thus, possible for safe code to dereference a null pointer from safe Rust code by calling these intrinsics. We reported this unsoundness in the API, which was confirmed by the library developers [166].

[166]: Compiler-builtins developers (2020), *Safety of intrinsics*

12.2.5.5 Foreign Functions

To detect interoperability with other languages we first measured how many types are equipped with `#[repr(C)]`, to have a memory layout com-

patible with C structures. Out of 1,486,978 definitions of structures and enumerations, we found that only 3.9% are annotated with `#[repr(C)]`. This annotation is used in 6.2% of all crates.

As a second query, we collected all crates whose name ends with `-sys`, to find those that adhere to the `-sys` naming convention for providing public bindings to a C system library. We found 650 crates (2.0% of all crates) whose names end with `-sys`, but we also noticed that other crates use different naming conventions: for 24 crates the name ends with `-ffi`, for 13 with `-bindings`, and for 10 with `-bindgen`. These suffixes all clearly mark a crate that provides public bindings to C libraries, as `-sys` crates should do. By further manual inspection of popular crates, we also found various crates such as `libc`, `gl`, and `winapi` that provide bindings to system libraries without using any naming convention. This plethora of cases suggests that the `-sys` convention is known, but not consistently applied by library developers.

As a third query, we measured how many unsafe functions are declared with a foreign ABI, to detect bindings to system library functions. We found that 248,522 (44.6%) out of 557,380 unsafe function definitions are actually static bindings to foreign items. This large percentage – which does not include functions that provide bindings to *dynamically* loaded libraries – shows that interoperability with foreign functions is actually a very common pattern of unsafe code. Overall, 1,599 crates (5.0% of all crates) contain at least one function with a foreign ABI. This reinforces the hypothesis that the `-sys` naming convention by itself is not enough to completely detect the crates that wrap system libraries.

Finally, as a fourth query, we measured usage of inline assembly. Out of more than 7 million function definitions we only found 493 cases of functions that use assembly. In particular, we found that 10 low-level and hardware-related crates actually contain 69.8% of all the functions that make use of inline assembly. This strongly suggests that inline assembly is in general rarely used.

12.2.5.6 Performance

Regarding our anticipated usages of unsafe code to improve performance, we found that 5.9% of unsafe calls in unsafe blocks involve unchecked functions spread across 4.3% of all crates. Avoiding run-time checks does not appear to be frequently used by all developers. However, it plays a significant role for some performance-oriented crates. For example, the Rust bindings of the X Window System call 4,852 unchecked functions in a single crate.

Developers rarely use the union `MaybeUninit`, which allows declaring uninitialised variables: We detected it in only 1,816 unsafe blocks, which appear in 0.55% of all crates.

In summary, performance optimisations using unsafe Rust seem to be a niche problem: They are mostly concentrated among a few crates. Within these crates, however, they are heavily used.

Discussion 13

We now discuss the overall results of our study of unsafe Rust. Moreover, we address possible implications of our findings for the Rust community.

13.1 Does the Rust Hypothesis Hold?

One of the main research questions which motivated our study is the Rust hypothesis, i.e., the claim that Rust developers both can and typically do write unsafe code according to the three basic principles introduced in the introduction of this part of the thesis. That is, unsafe code should be (1) *used sparingly*, (2) *straightforward*, *self-contained*, and (3) *well-encapsulated*. While this claim is widely-advocated in the Rust community, the results of our study support it *only partially*, and must be qualified by the fact that we discovered that unsafe code concerned with interoperability is far more prevalent than might be expected.

We found strong evidence that programmers usually adhere to the second principle, i.e., they keep their unsafe blocks simple (RQ 10.2) and self-contained (RQ 10.3). This is encouraging, as the rationale underlying this principle is to reduce the amount of code whose safety relies on manual efforts by programmers instead of automated guarantees by the compiler.

However, the first principle is not widely adhered to if one examines our data set as a whole; crates containing at least some unsafe code are not at all uncommon: we discovered that almost a quarter of all crates contain at least some unsafe code (RQ 10.1). The total ratio between safe and unsafe code also indicates that unsafe code is used quite extensively. When compared to previous studies, we observed that the usage of unsafe Rust is most-likely *decreasing* over time: possibly a reflection of efforts in the Rust community to reduce the overall reliance on unsafe code.

Regarding the third principle, while our initial measurements suggest that unsafe functions are extremely prevalent, our follow-up steps paint a clearer picture: Rust developers frequently seem to attempt to hide all of their unsafe functions from clients. Moreover, a great many of the exposed unsafe functions originate from crates that provide bindings for interoperating with hardware or libraries written in other languages (typically C and C++). After several attempts to classify these cases, we were left with sufficiently few public unsafe functions to conclude that for unsafe functions *implemented in Rust alone*, programmers avoid making at least the vast majority publicly visible (RQ 10.4). A precise measurement is made challenging by the fact that crates performing interoperability not only contribute many unsafe functions to our data set but do not, in general, adhere to naming conventions designed to make them easy to identify (*i.e.*, `-sys` suffix for crates providing C library bindings).

Overall, our results appear to support the Rust hypothesis for at least the majority of Rust code which is not for interoperability. However, there appear to be non-trivial exceptions to all three principles; something that project managers may want to keep in mind when adding dependencies to their own codebases, and software testers should consider paying close scrutiny to. Our Qrates analysis framework could be repurposed in this setting as a means of gathering important metadata about the usages of unsafe code in a crate under consideration. Simple checks could also be added to the official Rust linter Clippy [91] to warn of potentially-risky visibility of unsafe functions.

[91]: Clippy contributors (2023), *Clippy*

13.2 How Is Unsafe Rust Used?

The second key question of our study was concerned with finding the most prevalent use cases of unsafe Rust. To this end, we explored all reasons that the Rust compiler used to enforce the use of unsafe blocks, and considered six specific use cases based on our own manual classification.

In general, calls to unsafe functions suffice to explain why 83.5% of all unsafe blocks and unsafe functions need to be declared unsafe; unsafe functions are the main feature that tool developers who wish to support unsafe Rust should prioritise. This percentage climbs to 93.6% if we additionally consider raw pointer dereferences and to 99.4% by also including access to mutable static variables.

Taking a closer look at the individual use cases, we observed that the number of raw pointer dereferences is sizeable, yet low compared with calls of unsafe functions. It seems that Rust developers rarely use raw pointers for implementing custom C-style data structures; they seem to prefer either the standard library abstractions, or different implementation patterns, such as using vector-based heaps and integer indices. This has interesting consequences for tool developers, as many automated program analysis techniques, *e.g.* shape analysis [167], work well on tree-shaped data structures, but suffer dramatically in the presence of complex sharing. If complex sharing is predominantly achieved through a few standard abstractions and patterns, tools may have a chance to recognise and exploit idiomatic usages of popular abstractions.

[167]: Wilhelm et al. (2002), 'Shape Analysis and Applications'

Rust's type system and run-time checks rarely seem to concern developers enough to take the risk of circumventing the rules; they rarely (in fewer than 9% of all unsafe blocks) opted to disable compiler checks or transmute types to either gain performance or manually override the standard type system.

While concurrency-related compiler intrinsics appear only in a tiny fraction of crates, our manual follow-up on our experiments confirms that they can be quite dangerous: during our analysis, we discovered a bug that allowed us to dereference a null pointer from safe Rust code by calling one of them (albeit in an unstable feature under development).

We observed that many functions are declared unsafe despite not employing *any* of the language features which the compiler requires to be used only in unsafe code. This suggests either confusion on the part of

the programmer as to when such blocks are required, legacy reasons (*e.g.* the same function must be unsafe on a different platform), or that programmers employ such blocks to document either their own ad hoc correctness properties, or properties that are required for upholding Rust's safety even though the Rust compiler neither checks them nor associates them with unsafe Rust. An example of the latter case could be to indicate a precondition for a function which is necessary for the preservation of some *invariant* that the safety of *other* functions depends upon. While the latter case matches with the community's expectations[1], we encountered only few instances in which unsafe functions explicitly document contracts or invariants. Similarly, hardly any programmer declares unsafe traits, which act purely as documentation features (warning implementers to take care of a particular property). One reason for the low adoption of this feature could be that contracts and invariants are not enforced by the compiler. This could be changed by allowing developers to attach more formal specifications to traits and functions, for example, by developing a standard specification language used by all verification tools.

1: As expressed in communications with developers via the Rust Secure Code Working Group.

Finally, we observed that a large amount of unsafe code serves to interoperate with libraries written in other programming languages. Many crates purely act as an interface for these libraries. To facilitate future code analyses, it would be beneficial if these could be made easier to detect and classify, *e.g.* through an explicit attribute for low-level crates. We observed that naming conventions (in particular, the `-sys` convention for crates providing C bindings) appear to be insufficient for this purpose, presumably because programmers have their own preferences for crate naming.

14 Threats to Validity

In this chapter, we present threats to the validity of our empirical study and its evaluation.

14.1 Dataset

The main threat to validity is the selected dataset, and any biases this may introduce compared with current practice using the Rust language in general. By taking all currently-compiling crates from `crates.io`, we have a substantial slice of the open-source Rust code available, but other complementary sources (such as `github.com`) could also be solicited.

On the other hand, it is possible that we are sampling *too much* code from our chosen source; our dataset likely includes crates which are no longer maintained, for example. Arguably, these crates are not flawed as sources for our study, but the older they are, the less significant their features are for evaluating current practice.

Since we currently pre-filter crates by whether they compile, there is a chance that some crates have been missed because we were not able to identify the right compiler version or settings. We mitigated this last specific threat by taking such compiler arguments from the crate's own manifest; unfortunately, determining the intended compiler version is not possible in general.

14.2 Generated Code

Some occurrences of unsafe code are generated internally by the compiler. For example, the desugaring of pattern-matching constructs can yield unsafe code, even for source code which never uses unsafe Rust. To avoid, for example, potentially classifying such crates as containing usages of unsafe (RQ 10.1), we eliminate all compiler-generated unsafe blocks in advance (the compiler never generates unsafe functions) for all relevant queries in our experiments[1]. Since we filter out compiler-generated unsafe blocks but keep all compiler-generated safe code, our measurements may underestimate the ratio of unsafe to safe code. Similarly, the distribution of types of function calls reported for our entire dataset (Figure 12.5) may change when filtering out all compiler-generated code.

The measurements for RQ 10.2 (Size) show a wide distribution, with a few huge outliers. As explained in Chapter 12, rather than simply reporting our answers (and averages) directly, we were able to identify manually that these outliers are due to generated code. However, our dataset might contain other forms of non-standard Rust code (*e.g.* generated by scripts), which we did not identify. Nevertheless, the cumulative distribution we

1: We are grateful to Ana Nora Evans for showing us how to avoid pitfalls related to code generated by macros when extracting information from the Rust compiler.

chart forms an adequate basis for our conclusion that the vast majority of unsafe blocks are small.

14.3 Comprehensiveness of Queries

Our results are mostly based on queries that collect data automatically. Some of these queries check proxies for the properties of interest, for example, the size of unsafe blocks as a measure of complexity and the visibility of unsafe functions as an indication for the presence of safe abstractions. There is a risk that our choice of proxies (and queries) does not faithfully capture the properties of interest. To mitigate this risk, we complemented our automatic queries by manual code inspections.

14.4 Overlapping Motivations

Our presented analysis of the motivation for programmers using unsafe Rust (RQ 10.5) does not include a detailed analysis of the *overlap* between multiple motivations (for example, blocks containing both raw pointer operations and assembly code). Our framework and data set *do* provide this information, and exploring these correlations could be interesting future work.

14.5 Errors in the Implementation

Most of our results are based on our Qrates framework and subsequent data processing in Jupyter notebooks. Errors in the implementation could invalidate our findings. To mitigate this risk, we subjected all implementations to careful code reviews and tested them extensively.

14.6 Errors in the Rust Compiler

Some of our measurements, *e. g.* the reasons for unsafety in Table 12.3, rely on internal data computed by the Rust compiler – they are, consequently, sensitive to compiler bugs. While we checked Rust's bug tracker and asked for feedback from the Rust community to account for known compiler issues, our results may be influenced by so-far unknown bugs.

15 Related Work

In this chapter, we focus mainly on other studies of Rust code; other references to relevant related work are mentioned throughout this part of the thesis where appropriate. We start with the related work that appeared before the publication of our paper [76] in Section 15.1. Then, Section 15.2 discusses studies published later.

[76]: Astrauskas et al. (2020), 'How do programmers use unsafe Rust?'

15.1 Prior Work

Closest to our work is a study by [164] that performs a quantitative evaluation of unsafe Rust. Since their dataset relies on a 16 months older snapshot of the same repository (`crates.io`), we can observe a few developments over time: in particular, the percentage of crates containing *any* unsafe code (a measurement made in both studies) has dropped from 29% to 23.6% (noting though that these comparisons relate figures which were computed by different experimental methods).

[164]: Evans et al. (2020), 'Is Rust used safely by software developers?'

Apart from such basic statistics, the two studies are complementary as they take fundamentally opposite perspectives on the usage of unsafe code: we study (intended) compliance with the Rust hypothesis, i.e., commonly advocated best practices for writing unsafe code; in particular, our work takes into account whether programmers aim to shield their unsafe code behind safe abstractions. By contrast, [164] measure how the unsafe keyword permeates call chains and, thus, how many functions *transitively* depend on unsafe code; they consider all of these functions as tainted, or, in their terminology, "possibly unsafe". This notion ignores whether a function (1) depends on unsafe code through a safe abstraction that *takes responsibility* (by declaring exposed functions as safe) for Rust's guarantees, or (2) is *explicitly exposed* to unsafe code (by declaring all functions in the call chain as unsafe).

[164]: Evans et al. (2020), 'Is Rust used safely by software developers?'

These different perspectives can lead to quite different results for the same data set. For example, assume the corpus under analysis consists of two idealised codebases, say A and B, where A strictly adheres to the Rust hypothesis, and B frequently violates it. Moreover, suppose that many (declared safe) functions are at least transitively clients of the unsafe code in A and B, respectively. Our methodology *distinguishes* these codebases, concluding that some developers (those of A) aim to apply commonly advocated guidelines, and some do not (those of B). In contrast, the approach of [164] cannot distinguish them: both codebases are possibly unsafe. Furthermore, our study provides an in-depth analysis of the reasons *why* programmers employ unsafe code, which was not an apparent primary focus of [164] (where this question was explored via a survey rather than by analysing a codebase).

[164]: Evans et al. (2020), 'Is Rust used safely by software developers?'

[164]: Evans et al. (2020), 'Is Rust used safely by software developers?'

Our work evaluates the extent to which programmers *intend* for their unsafe code to be encapsulated from clients. However, we do not attempt

to judge whether the programmer's implementation *correctly* provides such an abstraction. This question has many subtle dimensions (as has been explained by *e.g.* [147]), and addressing it, in general, requires extremely sophisticated formal reasoning, as explored in the context of the RustBelt project [71].

[147]: Jung (2016), *The Scope of Unsafe*

[71]: Jung et al. (2018), 'RustBelt: securing the foundations of the Rust programming language'

[158] perform manual code inspections of 850 selected instances of unsafe Rust, i.e., unsafe blocks or functions, to elicit the motivation for using unsafe code and to explore the memory safety and concurrency errors caused by these instances of unsafe code. They select examples for their study by analysing bug reports and filtering commit messages for keywords indicating memory errors. Their work includes an analysis of incorrect usages of unsafe code but does not assess how and why unsafe code is used in general. Our much larger dataset yields a different perspective in several respects: for example, they observe that "calling unsafe functions counts for 29% of the total unsafe usages", whereas the percentage across our full data set is much larger, namely 84.6%. We can also confirm over our larger data set their observation that a significant number of unsafe functions need not (from the compiler's perspective) be labelled as unsafe.

[158]: Qin et al. (2020), 'Understanding memory and thread safety practices and issues in real-world Rust programs'

[168] performs an early study of how unsafe Rust is used. The high-level approach is somewhat similar to our own; source code analysis is performed by a compiler plugin that collects statistical data about unsafe blocks and functions. The presented results are mostly covered by our first two research questions. In particular, Ozdemir notes that 30% of all crates contain some unsafe code; this confirms the trend we noted in Chapter 12 alongside RQ 10.1 that the overall number of unsafe crates appears to be declining over time. Given the significant changes to the Rust language and its user base over the last four years, we do not compare all results in detail.

[168]: Ozdemir (2016), *Unsafe in Rust: Syntactic Patterns*

In addition to the above studies, which specifically consider unsafe Rust, there have also been qualitative studies on how well-suited Rust could be for certain applications: [169] study the usability of Rust's cryptographic libraries, whereas both [170] and [171] evaluate Rust in general as a systems programming language.

[169]: Mindermann et al. (2018), 'How Usable Are Rust Cryptography APIs?'

[170]: Balasubramanian et al. (2017), 'System Programming in Rust: Beyond Safety'

[171]: Levy et al. (2017), 'The Case for Writing a Kernel in Rust'

15.2 Later Studies

The studies published after the publication of our paper [76] mostly focused on understanding bugs and vulnerabilities related to unsafe code. [172] presented a static analyser RUDRA that enabled the authors to scan the entire Rust ecosystem for specific problematic patterns related to unsafe code. The scan revealed 264 previously unknown bugs caused by implicit execution paths introduced by panics and incorrect bounds of `Send` and `Sync` traits. [173] analysed all existing Rust common vulnerabilities and exposures (CVEs) of memory-safety issues by 2020-12-31. They found that all bugs either required to use unsafe or exploit Rust compiler bugs. [174] focused on the lifecycle of vulnerabilities discovered between 2014-11-11 and 2022-05-24. They also found that unsafe code is more likely to be vulnerable, but they also noticed that programmers tend to fix vulnerable unsafe blocks by replacing them with safe code. All

[76]: Astrauskas et al. (2020), 'How do programmers use unsafe Rust?'

[172]: Bae et al. (2021), 'Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale'

[173]: Xu et al. (2022), 'Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs'

[174]: Zheng et al. (2023), 'A Closer Look at the Security Risks in the Rust Ecosystem'

these studies demonstrate that using unsafe code correctly is notoriously hard and that there is a clear motivation for developers to write the code according to the three principles mentioned in the introduction. However, as discovered by [174], some developers do that only after they find vulnerabilities in their unsafe code, which could explain why unsafe code is used more commonly than expected.

[174]: Zheng et al. (2023), 'A Closer Look at the Security Risks in the Rust Ecosystem'

[175] analysed the documentation of unsafe methods in the Rust standard library and found that even this well-maintained documentation sometimes misses important information or has other inconsistencies. Together with our finding that many unsafe functions do not document requirements imposed on callers, this study shows an important weakness in Rust development practices.

[175]: Cui et al. (2023), 'Is unsafe an Achilles' Heel? A Comprehensive Study of Safety Requirements in Unsafe Rust Programming'

Conclusions of Part II 16

We have presented a large-scale study addressing the current practices of Rust programmers with respect to unsafe Rust, as well as an investigation of the *motivations* for why unsafe code is employed in practice. We identified three commonly held expectations regarding unsafe Rust practice (the Rust hypothesis), and our study showed partial support for these. In particular, while our study shows that unsafe code is very commonly used in small and self-contained quantities, it is much less scarce overall than one might expect, and in total, a great many unsafe functions are exposed to arbitrary client code across crate boundaries. On the other hand, a very sizeable portion of these functions ultimately result from the need to provide interoperability with custom hardware and native code written in C; when one eliminates the vast majority of these cases, it becomes clear that most unsafe functions written in Rust are not actually publicly-accessible, indicating a common effort by programmers to encapsulate the unsafe aspects of their implementations. On the other hand, many of the unsafe functions we investigated manually did not document the intended requirements imposed on their callers; this suggests a weakness in development practice which programmers should be aware of and software testers should look to highlight.

We have also presented the Qrates framework with which we have carried out our study, along with a large repository of harvested data on unsafe code usage. Our framework and experimental methodology can be straightforwardly reused for a wide variety of Rust-related empirical studies (not limited to unsafe code). They could, for example, be used to complement the annual Rust Survey [176] with data on current coding practice in the community. This survey asks specifically for recommendations for improvement; the widespread reuse of C libraries we have observed during our study (*e.g.* those providing GUI libraries such as Qt with no existing Rust alternative, or facilities to dynamically load code, which is not natively possible in Rust) provides empirically-justified directions for future language and library development.

[176]: Rust Language Team (2019), *Rust Survey 2019 Results*

Our investigation of the *reasons* why unsafe code is employed shows that the vast majority of unsafe code is used to call unsafe functions, while only a few other causes arise commonly. These results have important implications for the potential development of Rust analysis tools, particularly where the aim is to help programmers reason about whether their unsafe code is correct; an ability to specify or otherwise support reasoning about external function calls is a critical concern, while our study shows that support for, say, inline assembly need not be prioritised.

In the next part of the thesis, we present a technique for verifying safe abstractions that are supposed to safely encapsulate unsafe code inside them.

Part III

Verifying Mixed Safe and Unsafe Rust Code

In Part I, we showed how to leverage the memory safety guarantees provided by the Rust type system to enable lightweight verification. Unfortunately, unsafe Rust can break the memory safety guarantees the compiler provides. In Part II, we discussed that the safety of unsafe code depends not only on code within, for example, an `unsafe` block but also on safe code surrounding the unsafe block and even on safe code in other methods. Multiple studies have shown that ensuring the correctness of such code is crucial. Our study (presented in Part II) found that unsafe code is relatively common, with more than one in five crates (Rust packages) containing some unsafe code. [164] found that a large portion of safe Rust transitively depends on some unsafe code. All studies that focused on vulnerabilities [158, 172–174] found that almost all vulnerabilities in Rust projects are caused by bugs in code related to unsafe. Therefore, in this part of the thesis, we focus on verifying functional correctness and memory safety of mixed safe and unsafe code. We aim to maintain the benefits of our work from Part I and keep verification of safe code simple even if the same function contains some unsafe code. Achieving this goal is challenging because, as discussed in the thesis introduction, simplicity is often in direct conflict with expressivity, which is needed for verifying complex unsafe code.

[164]: Evans et al. (2020), 'Is Rust used safely by software developers?'

[158]: Qin et al. (2020), 'Understanding memory and thread safety practices and issues in real-world Rust programs'
[172]: Bae et al. (2021), 'Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale'
[173]: Xu et al. (2022), 'Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs'
[174]: Zheng et al. (2023), 'A Closer Look at the Security Risks in the Rust Ecosystem'

As we discussed in Part II, the programmers writing libraries are supposed to hide unsafe code behind safe abstractions so that the safe clients of these libraries do not need to worry about memory safety [72]. To completely relieve the programmer writing the client code from the cognitive load of thinking about memory safety, a safe abstraction is required to ensure memory safety even when used incorrectly. However, [172, 173] found that programmers writing libraries that provide a safe interface over an unsafe implementation often make incorrect assumptions about their safe clients. For example, they forget that callbacks provided by clients may panic. One of the challenges with correctly handling panics is that panic implicitly executes drop handlers, which may observe invalid memory and cause a memory error. Since most Rust code is unverified, verifying that a safe abstraction guarantees memory safety even when used by unverified clients has potentially a much more significant impact than verifying that it is functionally correct. Therefore, in this part of the thesis, we aim to support verifying two complementary use cases: mixed safe-unsafe code is functionally correct and safe abstractions guarantee memory safety even when used by unverified clients written in safe Rust.

[72]: Developers (2023), *The Rustonomicon: How Safe and Unsafe Interact*

[172]: Bae et al. (2021), 'Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale'
[173]: Xu et al. (2022), 'Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs'

Supporting verifying all aspects of unsafe code within a single thesis is, unfortunately, unrealistic, partly because the specific rules describing what should be allowed in unsafe code are still under active discussion. Our study found that calls of unsafe functions and uses of raw pointers account for 93.6% of unsafe code uses. Moreover, [174] found that 59.7% of vulnerabilities are caused by bugs that are typical for code that uses raw pointers: buffer overflows, use-after-free and null pointer dereference. Therefore, as a first step towards verification of complex unsafe code, we decided to focus on code that uses unsafe functions and raw pointers with a focus on guaranteeing the absence of the standard memory safety errors such as the ones found by [174]. In particular, we do not aim yet to support verifying that code adheres to the aliasing requirements proposed in either Stacked Borrows [126] or Tree Borrows [127, 128]

[174]: Zheng et al. (2023), 'A Closer Look at the Security Risks in the Rust Ecosystem'

[174]: Zheng et al. (2023), 'A Closer Look at the Security Risks in the Rust Ecosystem'
[126]: Jung et al. (2020), 'Stacked borrows: an aliasing model for Rust'
[127]: Villani (2023), 'Tree Borrows'
[128]: Villani (2023), *Tree Borrows: A new aliasing model for Rust*

(currently, there is no verification approach that would support them). However, as we explain below, our automation technique relies on non-aliasing guarantees provided by these models[1]. We leave extending our verification approach to support checking that code adheres to these models to future work.

1: If the code being verified does not uphold the required non-aliasing guarantees, the verifier may report an unexpected verification error.

Currently, two logics can be used for modular unbounded verification of unsafe Rust code: RustBelt [71] and RustHornBelt [133]. We already discussed both in Chapter 7. RustBelt enables verifying that an internally unsafe function is safe against arbitrary safe clients, while RustHornBelt enables proving functional specifications for both safe and unsafe code. However, there are at least three downsides to using RustBelt and RustHornBelt for verification. First, both RustBelt and RustHornBelt require manually translating Rust programs into λ_{Rust} and verifying them manually in a proof assistant. Manually translating code is a tedious task; as a result, the translated examples typically focus on a few key aspects of the example, which leads to a large gap between the actual Rust code and the verified λ_{Rust} model. For example, neither approach supports verifying panic safety. Second, while RustBelt and RustHornBelt are similar, they are still two different verification approaches that make different assumptions about clients: RustBelt assumes that clients are safe (or unsafe and verified) while RustHornBelt assumes that clients are verified. As a result, if we wanted to verify both the safety of a safe abstraction and its functional correctness, we would have to do two completely separate proofs, one in each of these logics. Third, our study found that for 92.3% of the crates containing some unsafe code, no more than 10% of all code is unsafe. While RustBelt and RustHornBelt can be used to verify not only unsafe code, but also safe and mixed safe-unsafe code, these approaches are significantly less automated than SMT-based approaches and require deep-expert knowledge. In Chapter 7, we also discussed Verus [109], a verifier for a dialect of Rust with its own primitives for implementing patterns such as double-linked list that in Rust require using unsafe code. While Verus is lightweight and reasonably expressive, it requires all code to be verified in Verus [111] and neither supports Rust's raw pointers nor verifying that a safe abstraction is actually safe. To summarize, currently, no approach is lightweight enough to be usable for verifying real code bases and capable of handling the critical aspects of unsafe code.

[71]: Jung et al. (2018), 'RustBelt: securing the foundations of the Rust programming language'

[109]: Lattuada et al. (2023), 'Verus: Verifying Rust Programs using Linear Ghost Types'

[111]: Developers (2023), *Verus Tutorial and Reference: Memory safety is conditional on verification*

In the introduction of the thesis, we stated our goal to develop an incremental verification approach that is lightweight for common cases and enables a smooth transition to more powerful verification methods when needed. In this part of the thesis, we present an approach for verifying mixed safe-unsafe Rust. Similarly to the approach presented in Part I, this approach is based on generating the core proof in implicit dynamic frames and layering functional specifications on top of it. However, it solves the challenges related to unsafe code and the interaction of unsafe code with safe code we mentioned above. For entirely safe code, the approach presented in this part is as lightweight as the one presented in Part I and its complexity for verifying unsafe code depends directly on the complexity of the unsafe code[2]. Enabling such lightweight verification requires completely automatically deriving the core proof for some part of the function, even in the presence of unsafe code, while at the same time giving the user enough expressive power to verify the

2: Complexity and amount of unsafe code are not always directly related: if a programmer omitted the runtime bound checks for accessing an array, the programmer might not need to write any specifications at all because, for example, the checks are implied by the loop condition. However, showing that a single unsafe function call is safe may require an arbitrarily complex argument.

unsafe elements. The approach presented in Part I relied on the absence of mutable access via multiple aliases. However, this property can be violated by using raw pointers. Our approach relies on two key observations. Our first observation is that the non-aliasing requirements intended[3] to enable the compiler to optimise Rust code (such as the ones imposed by Stacked Borrows or Tree Borrows) enable us to slice a Rust function into *managed* and *non-managed* parts. As a result, we can reuse techniques from Part I to automatically derive the core proof for the managed part, thus requiring the developer to provide specifications related to memory safety only for the non-managed part. Our second observation is that the two goals (verifying a safe abstraction's correctness and safety in the presense of safe unverified clients) have a substantial common part: they both require proving the memory safety of non-panicking executions. Therefore, we are able to reuse the memory safety specifications for both goals, significantly reducing the amount of specifications the user has to write.

3: The requirements are still under active discussion and, therefore, most of them are not yet used for optimisations.

Contributions. In this part of the thesis, we make the following contributions:

1. We refine the core proof presented in Part I to enable verification of unsafe code that uses raw pointers. We focus on verifying the ownership-related memory safety properties and leave checking of provenance and aliasing of raw pointers to future work since these topics are still under active discussion in Rust community.
2. We present a modular verification and specification approach that enables the verification of safe and unsafe functions containing safe and unsafe code. The presented approach enables verifying both safety and functional correctness together. The verification effort needed for verifying completely safe code is equivalent to the approach presented in Part I while the effort for verifying mixed safe-unsafe code depends directly on the complexity of the unsafe code.
3. We show how to extend our specification and verification approach to enable verifying memory safety of code containing implicit control flow paths caused by potential panics and drop handlers.
4. We evaluated our approach by extending Prusti and attempting to verify challenging examples, which revealed interesting limitations in the underlying Viper infrastructure.

Borrowing. In this part of the thesis, we focus on verifying mixed safe-unsafe code and consider borrowing an orthogonal concern. However, to be able to evaluate our approach on interesting examples, we have to support reasoning about borrows. Unfortunately, the model of borrows we presented in Part I does not model lifetimes, which are used extensively to ensure the memory safety of safe abstractions. Therefore, we develop a new model of borrows based on RustBelt and RustHornBelt. Since the model is still a work in progress, we do not claim it as a contribution yet.

Verified Properties and Trusted Assumptions. The methodology and its prototype implementation presented in this part of the thesis ensures

two properties. The first one is the same as presented in Part I: if a Viper backend verifier accepts the generated proof, every execution of the original Rust function that satisfies the user-written precondition is guaranteed to execute without memory and runtime errors and to satisfy the user-written postcondition. The key difference from Part I is that we extend our work to unsafe Rust. The second property is specific to safe abstractions: if a Viper backend verifier accepts the generated proof of a Rust function declared as safe (which can still contain unsafe code inside it), every execution of the original function with function arguments matching their types is guaranteed to execute without memory errors. These two properties are ensured under three assumptions. Two of them are the same as in Part I: we are assuming that Viper is sound and our modelling of Rust types and operations in Viper is correct. The third assumption is that our new model of borrows based on RustBelt and RustHornBelt is correct. We discuss this assumption in more detail in Chapter 21, which presents our new model of borrows. It is important to note that our new model of shared borrows does not depend on the correctness of the borrow checker, unlike the one presented in Part I.

Outline. This part of the thesis is structured as follows. In Chapter 17, we present a motivating example and discuss the challenges for verification brought by unsafe code. Then, we present our approach bottom-up. In Chapter 18, we present the ingredients needed to manually construct a core proof in Viper suitable for verifying unsafe code with raw pointers and without references. In this chapter, we focus on presenting the ingredients and postpone the discussion on how to automate the generation of the core proof for later. We call the presented core proof *unsafe core proof*; it is suitable for verifying unsafe, safe, and mixed code. In this part of the thesis, we will refer to the core proof introduced in Part I as *safe core proof*. In Chapter 19, we present the additional tools for checking that safe abstractions guarantee memory safety even in the presence of unverified safe clients. With the ingredients covered, Chapter 20 shows how the new core proof for unsafe code can be linked to ideas presented in Part I. More specifically, we show how and to what extent we can reuse our PCS-based automatic generation of the core proof and extend the specification language from Part I to enable specifying mixed safe-unsafe code. We finish this part by describing our new model of borrows in Chapter 21, presenting our implementation and evaluation in Chapter 22, discussing related work in Chapter 23, and concluding in Chapter 24.

17 Motivating Example

In Chapter 5, we showed a linked list written in safe Rust for which we specified its functional behaviour. Figure 17.1 repeats the example but shows only the parts visible to the client. In the original example, methods `index` and `index_mut` are implemented by recursively traversing the list, which is inefficient. Therefore, we would like to implement a more efficient data structure using unsafe code. However, the client should not be able to tell the two data structures apart (except for differences in performance), which means that our implementation must satisfy two properties. First, it must be a safe abstraction that ensures memory safety even when used by unverified safe clients. Second, it must satisfy the specifications given in Figure 17.1.


```
pub struct Node<T> { /* ... */ }
impl<T> Node<T> {

    #[ensures(result.len() == 0)]
    pub fn new() -> Self { /* ... */ }

    #[pure]
    pub fn len(&self) -> usize { /* ... */ }

    #[pure]
    #[requires(0 <= i && i < self.len())]
    pub fn index(&self, i: usize) -> &T { /* ... */ }

    #[requires(0 <= i && i < self.len())]
    #[ensures(result === old(self.index(i)))]
    #[after_expiry(
        self.len() == old(self.len()) &&
        forall(|j: usize|
            0 <= j && j < self.len() && j != i ==>
            self.index(j) === old(self.index(j))
        ) &&
        self.index(i) === before_expiry(result)
    )]
    pub fn index_mut(&mut self, i: usize) -> &mut T {
        /* ... */
    }
}
```

Figure 17.1: Linked list interface visible by the client.

17.1 Memory Safety

Figure 17.2 shows `ArrayList`, a safe abstraction similar to `Vec` from the standard library but providing the same interface as our linked list from Figure 17.1. Before we discuss what we need to guarantee to show that

`ArrayList` is a safe abstraction, we need to discuss how we can reason about its memory safety. According to the `Vec`'s documentation [177], `Vec` can be understood as a triplet (`ptr`, `cap`, `len`) where `ptr` points to a memory block that can store `cap` elements of type `T`, the first `len` of which must be initialised. Our struct `ArrayList` contains three fields each corresponding to one element of the triplet: `ptr` (line 2), `cap` (line 3), and `len` (line 4). `index_unchecked` (lines 24–28) is an unsafe method that allows the client to obtain a shared reference to an element. The caller of this method must ensure three properties so that this method can execute successfully and return a shared reference. First, the caller needs to ensure the property specified in the `Vec` documentation saying that `ptr` points to a block of `len` initialised elements. Second, the caller must ensure that index `i` is in range. If a programmer called `index_unchecked` with argument `i` that is larger than capacity `cap`, pointer `element_ptr` would point to unallocated memory and dereferencing it would lead to a memory error. Similarly, if the function was called with `i` larger than length `len` but smaller than the capacity `cap`, the pointer `element_ptr` would point to potentially uninitialised memory and may lead to a memory error. Third, Rust shared references must be dereferenceable for the duration of their lifetime, or as we would say in the terminology of Part I, they carry the read capability. Therefore, the caller must give the capability to `index_unchecked` to create such a reference for the specified lifetime. Out of these three properties, the specification language presented in Part I allows specifying only that parameter `i` is in range, but not the other two.

[177]: Developers (2023), *Struct Vec*

To support specifying the required properties, we need to extend our specification language from Part I with support for specifying allocation, initialisation, and owned capabilities of memory pointed at by a raw pointer. Since `index_unchecked` reads from pointer `element_ptr`, the pointer must point to initialised memory. If the pointer were used to write to a memory location, it would be sufficient for the memory to be only allocated. To support these two cases, we need to be able to distinguish between allocation and initialisation in our specification language. In addition to these two properties, unsafe functions may have additional requirements that have to be ensured to guarantee memory safety. For example, the deallocation function `dealloc` can be called only for memory blocks allocated on the heap, and it has to be called with exactly the same arguments as the ones used when allocating the memory block. Besides being able to specify allocation, initialisation, and capabilities of contiguous memory blocks, our specification language must be able to capture more complex patterns to support realistic examples. For example, function `std::ptr::copy` [178] copies data from one memory block to another, and the blocks can be *overlapping*.

[178]: Developers (2023), *Function std::ptr::copy*

```rust
pub struct ArrayList<T> {
    ptr: *mut T,
    cap: usize,
    len: usize,
}
impl<T> ArrayList<T> {
    pub fn new() -> Self {
        let cap = if size_of::<T>() != 0 {
            0
        } else {
            isize::MAX as usize
        };
        Self {
            ptr: dangling_ptr(),
            len: 0,
            cap,
        }
    }

    pub fn len(&self) -> usize {
        self.len
    }

    pub unsafe fn index_unchecked(&self, i: usize) -> &T {
        let element_ptr = self.ptr.add(i);
        let result = unsafe { &*element_ptr };
        result
    }

    pub fn index(&self, i: usize) -> &T {
        assert!(i < self.len);
        unsafe { self.index_unchecked(i) }
    }

    pub fn index_mut(&mut self, i: usize) -> &mut T {
        /* Similar to index. */
    }
}
impl<T> Drop for ArrayList<T> {
    fn drop(&mut self) {
        if size_of::<T>() != 0 && self.cap != 0 {
            let mut i = self.len;
            while i > 0 {
                i -= 1;
                let element_ptr = self.ptr.add(i);
                unsafe { drop_in_place(element_ptr); }
            }
            let ptr = self.ptr as *mut u8;
            let layout = Layout::array::<T>(self.cap);
            unsafe { dealloc(ptr, layout); }
        }
    }
}
```

Figure 17.2: A safe abstraction of a list using unsafe code.

17.2 Encapsulating Unsafety

We discussed in the previous section that method `index_unchecked` is unsafe, which means that ensuring memory safety is a shared responsibility between the implementation and the caller. The developer writing the implementation of the method has to explicitly specify the conditions necessary for the method to execute without memory errors. The developer writing the implementation of the caller has to uphold the specified conditions. Method `index` (lines 30–33) is a safe wrapper around `index_unchecked`. As such, it is the sole responsibility of `index` to ensure the three properties we mentioned in the previous section that are needed to guarantee that `index_unchecked` executes safely. Importantly, `index` must ensure these properties even when it is used by unverified safe clients that may not respect its precondition. As a result, `index` is not allowed to make assumptions about its clients for memory safety, meaning its precondition for memory safety must be `true`.

Rust programmers use two key techniques for ensuring memory safety of safe abstractions, such as method `index`: *runtime assertions* and *invariants*. `index`'s implementation uses the assert on line 31 to ensure the requirement that parameter `i` is in the valid range. This assertion illustrates an important difference between assertions when verifying functional correctness, our primary focus in Part I, and when verifying memory safety of safe abstractions. In the former, our goal was to prove that assertions never fail. In the latter, we must prove that the runtime assertions are sufficient to guarantee memory safety. An example of an invariant would be the requirement from `Vec`'s documentation that `ptr` points to a memory block of `len` initialised elements. `Vec`'s developers aim to ensure that this informal invariant is maintained by all `Vec`'s methods.

Runtime assertions and invariants are powerful tools for ensuring that safe abstractions are memory-safe even when used by arbitrary safe clients. However, using them correctly is challenging, especially because invariants written in English prose are prone to be imprecise. First, since the safe abstraction is required to work with arbitrary safe clients, the programmer must consider all possible edge cases. For example, generic parameter `T` in our example could be instantiated with a zero-sized type (ZST). The Rust API for allocating raw memory blocks requires allocated blocks to have a positive size and, therefore, the `ArrayList` is required to avoid calling the allocation API when it is instantiated with a zero-sized type. As a result, constructor `new` (lines 7–18) and destructor `drop` (lines 40–52) treat ZSTs specially. Second, the programmer has to ensure that the invariants and runtime assertions are sufficient to guarantee memory safety. For example, if we omitted the assert (line 31) in method `index`, the method would still be functionally correct: it is guaranteed to return the correct element when called in a context that satisfies the three safety requirements. However, it would not guarantee memory safety. Third, the programmer has to ensure that safe clients cannot violate the invariant. Our `ArrayList` ensures this property by making the fields private and requiring that all safe methods that may modify these fields preserve the invariant. Solving all these challenges requires great expertise and care from the programmer to consider all cases. However, the situation is aggravated by Rust features like panics that can lead to implicit control flow, as elaborated next.

17.3 Panic Safety and Drop Handlers

In the previous section, we showed that programmers use runtime assertions to prevent executions that may lead to memory errors. `assert!(...)` macros in Rust work similarly to `assert` statements in Java. When a Java `assert` statement fails, an exception is thrown, and the execution is propagated up the stack until it is caught or reaches the top of the stack and terminates the thread. Similarly, when a Rust `assert!(...)` fails, a panic is raised, and the stack is unwound by implicitly propagating the execution up the stack until the panic is caught or the top of the stack is reached, which results in terminating the thread[1]. Panics significantly complicate reasoning about code because Rust does not have a notion of non-panicking functions and, therefore, the programmer should expect that any called function, whose implementation they do not know, could panic, causing stack unwinding at the most unexpected program points. Unfortunately, even the documentation for functions in the standard library often does not explicitly mention whether a function could panic. [172, 173] found that especially problematic are client-provided functions, whose panics programmers implementing safe abstractions forget often to handle. For example, the comparison `a == b` in the following snippet is (in general) a call to a user-provided function and may panic.

1: It is possible to configure the compiler to abort the process immediately instead of unwinding the stack. This approach is sometimes used as a workaround for problems caused by panics. However, unwinding on a panic is the default and more common configuration, and therefore, it is important to ensure that unwinding does not cause memory errors.

[172]: Bae et al. (2021), 'Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale'

[173]: Xu et al. (2022), 'Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs'

```
pub fn compare<T: Eq>(a: &T, b: &T) -> bool {
    a == b
}
```

In the previous section, we mentioned that safe abstractions rely on invariants to prevent memory errors; therefore, it is crucial for the code to maintain them. However, methods often need to temporarily break an invariant, which is fine as long as the invariant is restored before calling methods that expect the invariant to hold. Panics may lead to calling methods when an invariant is broken in two ways. First, drop handlers such as the one on lines 40–52 in Figure 17.2 typically assume that the invariant of the deallocated type holds. When a stack is unwound by a panic, all local variables are deallocated by calling their drop handlers. If any of the variables is in an invalid state, a memory error may be caused when executing its drop handler. Second, a panic is an implicit return, and it is the responsibility of the safe abstraction to restore the invariant when returning from a method. For example, the following snippet shows a method that overrides the contents of `ArrayList` with new data from some source `input`.

```
pub fn override_from_input<T: Input>(
    &mut self,
    input: &mut T,
) {
    unsafe {
        let len = read_into(input, self.ptr, self.cap);
        self.len = len;
    }
}
```

If function `read_into` panics, the execution will return to the caller with the number of initialised elements not matching the value stored in the `len` field. The caller then can catch the panic and then trigger a memory error

by calling `index` with an argument that accesses an uninitialised element or by calling the drop handler that tries to deallocate an uninitialised element.

Programmers use three techniques to prevent memory errors caused by panics when an invariant is broken. The first technique programmers use is ensuring that no code can panic when an invariant is temporarily broken. For example, the following snippet shows a comment from the implementation of the `swap` function in the standard library [179] stating an assumption that none of the called functions are supposed to panic. However, this property is not even mentioned in the documentation of the called functions.

[179]: Developers (2023), *Function std::mem::swap*

```
// SAFETY: exclusive references are always valid to
// read/write, including being aligned, and nothing
// here panics so it's drop-safe.
unsafe {
    let a = ptr::read(x);
    let b = ptr::read(y);
    ptr::write(x, b);
    ptr::write(y, a);
}
```

The second technique is ensuring that the invariant of the data structure always holds [180]. For example, the following snippet shows a variation of method `override_from_input` from above with an additional statement that sets `len` to zero before calling `read_into`, making all elements unreachable in case of a panic.

[180]: Beingessner (2015), *Pre-Pooping Your Pants With Rust*

```
pub fn override_from_input<T: Input>(
    &mut self,
    input: &mut T,
) {
    unsafe {
        self.len = 0;
        let len = read_into(input, self.ptr, self.cap);
        self.len = len;
    }
}
```

The downside of this approach is that the initialised elements will be leaked in the case of a panic. While this solution is not ideal, it guarantees memory safety because leaking resources in Rust is considered safe (programmers can explicitly leak resources in safe code using the safe function `std::mem::forget` [181]).

[181]: Developers (2023), *Function std::mem::forget*

In some cases, memory leaks are avoided using the third and last technique: fixing the broken invariant when a panic occurs. This technique relies on the fact that drop handlers are executed even when a panic occurs, which allows the execution of custom code to fix the invariant. Examples we found that use this technique are large and complex. Figure 17.3 shows pseudo-code for the method `dedup` that removes duplicate elements from `ArrayList`. The pseudo-code is based on the implementation of `Vec::dedup_by` [182]. Lines 7–10 declare a method-private struct `FillGapOnDrop` that takes a mutable reference to the array list and also keeps track information which elements are properly initialised.

[182]: Developers (2023), *Method Vec::dedup_by*

An instance of the struct is created on line 17 and stored in variable `gap`. The while loop that does the actual deduplication calls the equality and drop methods on instances of `T`. These methods are provided by the client code and can panic. If a panic occurs, the destructor of `FillGapOnDrop` on lines 11–15 is called, which restores the invariant of the list. If the execution does not panic, the variable `gap` is forgotten on line 29, which prevents the execution of its drop handler. While the example shown in Figure 17.3 focuses on the last technique, the actual implementation of `Vec::dedup_by` also relies on parts of the code not panicking. All three techniques are error-prone and require extreme care from the programmer to ensure that memory safety is guaranteed in all cases, which makes those panic safety properties an important target for verification.

```
1  pub fn dedup(&mut self) {
2      let len = self.len();
3      if len <= 1 {
4          return;
5      }
6
7      struct FillGapOnDrop {
8          list: &mut ArrayList<T>,
9          /* ... */
10         }
11     impl Drop for FillGapOnDrop {
12         fn drop(&mut self) {
13             /* ... */
14         }
15     }
16
17     let mut gap = FillGapOnDrop {
18         list: self,
19         /* ... */
20     };
21     let ptr = gap.list.ptr;
22
23     /* While loop that does the actual deduplication. */
24
25     // Fix the invariant.
26     gap.list.len = /* ... */
27
28     // Forget the drop guard.
29     mem::forget(gap);
30 }
```

Figure 17.3: A pseudo-code for `dedup` method for `ArrayList` based on `Vec::dedup_by` implementation [182].

One aspect hidden in the simplified example illustrating the third technique (Figure 17.3) is that the drop trait implementation used for fixing the broken invariant is often not a behavioural subtype of trait `Drop`. Method `drop` of trait `Drop` must be callable on any valid instance of the type, which means that its precondition should be simply `true`. However, the drop handlers used to fix invariants typically have additional requirements to be ensured at the call site. Drop handlers not being behavioural subtypes of the `Drop` trait is acceptable because structs like `FillGapOnDrop` are method-private and never passed as generic arguments. As a result, the drop handler is invoked only from program points where it is known what implementation will be executed.

It is worth noting that drop handlers are not entirely normal Rust functions. In particular, their signature is a lie: `&mut self` in a regular Rust method means that the method can assume that the reference `self` is pointing to a valid value when the method is called **and** that the method needs to ensure that the reference is still pointing to a valid value at the end of the method. However, drop handlers are not intended to ensure that the value is completely valid at the end of the method because they would not be able to deallocate memory.

17.4 Verification Approach Overview

In the remaining chapters of this part, we show how we generalise the approach we took in Part I to address the challenges mentioned in this chapter. We start by extending the core proof presented in Part I to support verifying memory safety of mixed safe-unsafe code in Chapter 18. In that chapter, we present four major changes. First, we distinguish between *allocation* and *initialisation* capabilities. An allocation capability expresses ownership of an allocated but potentially uninitialised memory block. An initialisation capability expresses ownership of a memory block that is allocated and initialised (in Part I, we had only initialisation capabilities). Second, we change our model of addresses to support raw pointers and pointer arithmetic. Third, we introduce a new kind of specification (such as preconditions, postconditions, and invariants), which we call *core* specifications. Unlike regular specifications for verifying functional correctness and the absence of panics, core specifications enable users to specify conditions required to ensure memory safety such as the ownership and status of memory referenced by raw pointers. Since we base our core proof on implicit dynamic frames, a permission logic, it naturally captures capabilities to heap-allocated memory. Fourth, we enable users to formally state their intended invariants like the one mentioned in the documentation of `Vec` by writing them as *core type invariants* that express when a value is considered a valid instance of its type. We require the core type invariants to hold always when a value is moved, copied, or borrowed (we treat returning from a function as implicitly moving the borrowed values back to the caller) because these are the locations at which Rust requires values to be valid[2].

2: In Rust, using unsafe code to store the number 5 in a variable of type `bool` is allowed. However, moving or copying this variable while it stores 5 would immediately trigger undefined behaviour.

We build on this generalised core proof to enable verification of the two goals we identified in the introduction of this part: encapsulating unsafety in safe abstractions and functional correctness. As mentioned above, these two goals require different interpretations of specifications and runtime assertions. When verifying the memory safety of a safe abstraction, safe functions cannot rely on preconditions for memory safety, and runtime assertions are just early returns. When verifying functional correctness, runtime assertions must succeed when preconditions hold. We address this conflict by verifying each function twice: in *memory-safety* mode, we verify the function's memory safety by omitting functional preconditions (unsafe functions can rely on core preconditions) and treating each runtime assertion as an early return; in *functional-correctness* mode, we verify that runtime assertions cannot fail when the function's preconditions are upheld. An important aspect of this design choice to verify twice is doubling the work for the verifier rather than for the user.

We explain the differences between the two modes and how we handle the challenges related to verifying the safety of abstractions in Chapter 19. In Chapter 20, we present how we extend the automation presented in Part I to mixed safe-unsafe code. For example, even though method `len` on lines 20–22 in Figure 17.2 is declared on an internally unsafe type, it is entirely safe and we would like to be able to verify it without any additional input from the user. In that chapter, we present how we solve this challenge and enable lightweight specifications like the ones shown in Figure 17.1 for safe abstractions.

Our approach presented in this part of the thesis extends Part I with support for raw pointers, unsafe functions, and drop handlers. While we rely on guarantees given by Stacked Borrows or Tree Borrows models, we do not verify that programs adhere to these models. Our refined model of references is still a work in progress.

18 Verifying Memory Safety

Constructing the core proof for safe Rust in Part I was a two-step process. First, we defined the ingredients of the core proof: a model of memory locations, a model of Rust (MIR) operations, and a model of capabilities as permission logic resources together with ghost operations for transforming them. Then, we showed how, using information available in the Rust compiler, we can automatically combine these ingredients to justify the safety of each Rust operation. In this part of the thesis, we aim to construct the core proof suitable for verifying the safety of unsafe Rust code. In the rest of this part, we use "safe core proof" to refer to the core proof presented in Part I and "unsafe core proof" to refer to the core proof presented in this part of the thesis. In this chapter, we focus on the first step of defining the core proof: defining the ingredients. Similarly to Part I, we can automatically generate the definitions of ingredients based on the information available in the Rust compiler. However, automatically combining these ingredients into a complete core proof for any example of unsafe code is impossible (if a complete and automatic approach for checking memory safety existed, we would not need unsafe code). In this chapter, we assume that the user manually combines the ingredients into a core proof; we discuss to what extent and how we can automate the process in Chapter 20.

In safe Rust, memory is managed automatically, and the compiler guarantees that safe code can read memory when it represents valid instances of types. In unsafe Rust with raw pointers and unsafe functions, programmers can manually manipulate memory, and it is the programmer's responsibility to do that correctly. As a result, we want to ensure that memory accesses performed by Rust operations are safe by constructing a core proof for code containing unsafe operations. For example, we want to guarantee that code only writes to allocated memory. We call such core proof that guarantees memory safety of unsafe, safe, and mixed code *unsafe core proof*. We will refer to the core proof introduced in Part I as *safe core proof*. In a permission logic, we can show that an operation is safe by proving that the context executing the operation has sufficient permissions to the necessary resources. Similarly to Part I, we model capabilities as permission logic resources, which can be transferred and transformed but cannot be fabricated. We create and destroy the resources that model allocation capabilities exactly when the Rust allocation primitives prescribe it. We made the resources that model initialisation capabilities dependent on resources that model allocation capabilities to ensure that only allocated memory can be initialised. The core proof ingredients presented in this chapter enable manually specifying that the resources are passed around appropriately to justify the safety of Rust operations.

Since we target a substantial subset of a realistic programming language, we need to consider many details and edge cases. When designing the ingredients, we considered two possible designs. The first option was to try having as few general primitives as possible. The second option was to

have more but more specialised and easier-to-use primitives. We decided to go with the latter option because having easier-to-use primitives aligns with our goal of enabling programmers without extensive training in verification to verify code. We present the ingredients of the unsafe core proof by going over types from the simplest ones to the more complex ones. In Section 18.1, we present how we change the core proof to also capture allocation by presenting the updated model of primitive types. In Section 18.2, we present the composite types and explain how we change the modelling of fields to accommodate pointer arithmetic. Before we present types with invariants in Section 18.4, we provide a specification language that enables the user to combine the ingredients into a core proof in Section 18.3. In Section 18.5, we present low-level but essential details on how we define addresses to satisfy the requirements of certain constructs we use in our encoding. Similarly to Part I, we use Viper and Prusti to make examples concrete and easier to understand. However, the approach is not limited to Viper and is applicable to a general separation logic.

18.1 Primitive Types

In this section, we show the updated model of primitive types: integers, booleans, and raw pointers (since a raw pointer gives no capabilities to its target, we treat it as a primitive value that stores the target's address). Figure 18.1 shows a simple Rust function that assigns value `42` to variable `a`. The comments in the code indicate the places where the compiler inserts statement `StorageLive(a)` (line 3) that allocates `a` on the stack and statement `StorageDead(a)` (line 5) that deallocates `a`. The overall structure of the core proof is the same as presented in Part I: we model Rust types as Viper predicates and translate Rust functions into Viper methods.

```
1 fn main() {
2     let a: i32;
3     // StorageLive(a)
4     a = 42;
5     // StorageDead(a)
6 }
```

Figure 18.1: A simple Rust program that assigns value 42 to variable `a` of type `i32`.

An important difference between the safe core proof presented in Part I and the unsafe core proof is that the former models only initialisation capabilities while the latter models both initialisation and allocation capabilities. Since having an initialised memory location means that it is also allocated, we model the initialisation capability as allocation capability combined with the knowledge that the memory location contains a valid instance of a type. Similarly to the safe core proof, in the unsafe core proof, we model capabilities using Viper predicates. Figure 18.2 shows an updated predicate `Own<i32>` for modelling an initialised value of `i32` in a Viper-like syntax. We use a different name for the predicate to distinguish between the two versions of the predicate. Also, to make code more readable, we use predicate names formed by using a syntax similar to generics by putting the Rust type between angle brackets "<" and ">" (Viper does not support generic predicates).

```
predicate Own<i32>(address: Address) {
    acc(MemoryBlock(address, size<i32>()), write) &&
    is_valid<i32>(
        from_bytes<i32>(bytes(address, size<i32>()))
    )
}
```

Figure 18.2: A predicate representing an allocated and initialised value of `i32`.

Predicate `Own<i32>(addr)` models an initialisation capability to a valid instance of type `i32` at address `addr`. We model the allocation capability using predicate `MemoryBlock` on line 2 that represents ownership of untyped raw memory. The other conjunct on lines 3–5 captures the knowledge that the memory block represents a valid instance of type `i32`. The `MemoryBlock` predicate has two parameters: address and size. The address matches the address of the wrapping `Own<i32>` predicate while `size<i32>()` is a function that returns the size of `i32`, which is 4 bytes. `bytes<i32>` is a Viper heap-dependent function that returns the bytes stored in the memory block. Function `from_bytes<i32>` is a total function that maps the bytes to a snapshot representing a mathematical value of a `i32` integer. Unlike in Part I where snapshots represent only valid values, in our new model, a snapshot represents a valid value only if function `is_valid<i32>` returns true. We discuss snapshots in more detail in Section 20.2.

With the most important predicates covered, we can show how we translate a Rust function into Viper. Figure 18.3 shows a simplified encoding of function `main` from Figure 18.1. Viper variable `a_address` (line-2) models the address of Rust variable `a`. If a Viper variable is not assigned, it has an unconstraint symbolic value of its type. We use this property to encode the fact that the address of variable `a` is unknown. Unsurprisingly, we model allocation and deallocation of `a` by inhaling (line 5) and exhaling (line 13) predicate `MemoryBlock`. Rust requires a memory block to be deallocated with the same parameters as it was allocated. We ensure this property by following the RustBelt [71] approach and introducing a predicate that models a capability to deallocate a memory block. This predicate tracks the parameters with which the memory block was allocated, guaranteeing that it is deallocated in the same way it was allocated. Unlike RustBelt that models all memory on the heap, we distinguish between heap and stack allocated memory using `MemoryBlockDropHeap` and `MemoryBlockDropStack` predicates, respectively. Encoding of `StorageLive(a)` statement inhales an instance of `MemoryBlockDropStack`, and that instance is exhaled by the encoding of matching `StorageDead(a)`.

[71]: Jung et al. (2018), 'RustBelt: securing the foundations of the Rust programming language'

One advantage of our encoding for safe Rust is that we could use Viper assignments to model Rust assignments. This approach no longer works for the new encoding because Viper does not allow assigning to predicates. Instead, we model assignments by using helper methods such as `assign_constant<i32>`, which is called on line 9. These helper methods are implemented using more primitive operations. For example, method `assign_constant<i32>` (lines 17–23) uses abstract method `write_bytes` to write number 42 to the memory block and then folds an instance of the `Own<i32>` predicate to indicate that this memory now holds a valid value of type `i32`. To deallocate the memory for `a`, we need to convert `Own<i32>` back into `MemoryBlock`. We do this conversion by calling another helper method `forget_initialisation<i32>` on line 12.

```
method main() {
    var a_address: Address

    // StorageLive(a)
    inhale acc(MemoryBlock(a_address, size<i32>()), write) &&
        acc(MemoryBlockDropStack(a_address, size<i32>()), write)

    // a = 42;
    assign_constant<i32>(a_address, snap_constructor<i32>(42))

    // StorageDead(a)
    forget_initialisation<i32>(a_address)
    exhale acc(MemoryBlock(a_address, size<i32>()), write) &&
        acc(MemoryBlockDropStack(a_address, size<i32>()), write)
}

method assign_constant<i32>(address: Address, constant: Snap<i32>)
    requires acc(MemoryBlock(address, size<i32>()), write) && is_valid<i32>(constant)
    ensures acc(Own<i32>(address), write) && snap<i32>(address) == constant
{
    write_bytes(address, size<i32>(), to_bytes<i32>(constant))
    fold acc(Own<i32>(address), write)
}

method forget_initialisation<i32>(address: Address)
    requires acc(Own<i32>(address), write)
    ensures acc(MemoryBlock(address, size<i32>()), write)
{
    unfold acc(Own<i32>(address), write)
}
```

Figure 18.3: A simplified encoding of function `main` from Figure 18.1 into Viper.

18.2 Composite Types

Figure 18.4 shows an example similar to Figure 18.1, but with a composite type `Pair` that is a pair of integers instead of the primitive type `i32`. Predicate `Own<Pair>`, which models an allocated and initialised value of the `Pair` is shown in Figure 18.5. Similarly to the encoding presented in Chapter 2, the predicate's body contains an `Own<i32>` predicate for each field to indicate that the field is allocated and initialised. The main difference between the two encodings is that in Chapter 2, we used object identities instead of addresses to identify objects and had an additional indirection via Viper `Ref`-typed fields, which enabled us to model Rust assignments as Viper assignments. Unfortunately, this model is unsuitable for modelling unsafe Rust code that may perform pointer arithmetic for two reasons. First, the object identity changes when, for example, a field is reassigned while the address stays the same. This difference does not matter for safe code that works on the level of places but is crucial for unsafe code that may manipulate addresses directly. Second, the indirection via `Ref`-typed fields requires providing permissions to these fields, while in Rust, pointer arithmetic itself does not require any capabilities. Therefore, we use abstract addresses instead of object identities for our unsafe core proof.

```
1  struct Pair {
2      first: i32,
3      second: i32,
4  }
5  fn main() {
6      let a: Pair;
7      // StorageLive(a)
8      a = Pair { first: 41, second: 43, };
9      // StorageDead(a)
10 }
```

Figure 18.4: A simple Rust program that creates a variable of type `Pair`.

```
1  predicate Own<Pair>(address: Address) {
2      acc(Own<i32>(field_address_Pair_first(address)), write) &&
3      acc(Own<i32>(field_address_Pair_second(address)), write)
4  }
```

Figure 18.5: A simplified predicate representing an allocated and initialised value of `Pair` from Figure 18.4. To simplify explanation, we omitted the `MemoryBlock` used to represent padding.

We model addresses using a newly-defined Viper type `Address`[1], which we describe in more detail in Section 18.5. As we already showed in Figure 18.2, an address of a primitive type matches the address of the underlying memory block. For composite types like `Pair`, we compute the addresses of the fields by using uninterpreted mathematical functions. Function `field_address_Pair_first` takes the address of the pair and computes the address of field `first`. Function `field_address_Pair_second` does the same for field `second`. Our model expresses that Rust by default gives no guarantees about the layout. For example, it does not guarantee whether field `first` or field `second` will be first in memory[2].

1: We could have continued using the `Ref` type but decided that using a new type makes the encoding clearer.

2: Rust provides attributes that control what guarantees the compiler gives about the layout. By default, the compiler gives no guarantees, but, for example, adding the annotation `#[repr(C)]` guarantees that the compiler will preserve the field order during the compilation. We could support such annotations by adding axioms that constrain the values the field address computation functions return.

Another challenge for supporting composite types is reshaping `MemoryBlock` predicates. Figure 18.6 shows a simplified encoding of function `main` from Figure 18.4 into Viper. As can be seen on lines 5 and 14, `StorageLive` and `StorageDead` statements operate on memory blocks of size `size<Pair>()` while the `Own<i32>` predicate contains a memory block of size `size<i32>()`. Therefore, we introduce two operations for splitting and joining memory blocks. Helper method `assign_composite<Pair>` (lines 18–30), which encodes the assignment to variable `a`, uses method `split_memory_block` (on line 26) to split the memory block for `Pair` into memory blocks for its fields. Then, it calls for each field the helper method `assign_constant`, which consumes the memory block of size `size<i32>()` and produces an instance of `Own<i32>` predicate. Lastly, `assign_composite<Pair>` folds the predicates for fields into a predicate for the entire `Pair`. Similarly, helper method `forget_initialisation<Pair>` (lines 32–40) unfolds `Own<Pair>` into its fields, calls helper methods to convert fields into the underlying memory blocks, and uses `join_memory_block<Pair>` (line 39) to reconstruct the memory block for the entire struct. Many helper methods for composite types have a similar structure: they divide the predicate either by unfolding `Own` or splitting the memory block, recursively call the helper methods for fields, and recombine the new predicate either by folding `Own` or joining the memory blocks.

In our encoding, we axiomatise the split and join operations for each

type. We made this choice because we wanted to assume about layouts of Rust types as little as possible. However, if we specified the layout of a type, we could define and verify split and join operations for that type by changing predicate `MemoryBlock` from abstract to the one shown in the following snippet in Viper-like syntax.

```
predicate MemoryBlock(address: Address, size: Snap<usize>) {
    forall i: Snap<usize> :: 0 <= i && i < size ==>
        acc(MemoryBlockByte(byte_offset(address, i)), write)
}
```

Intuitively, the body of this predicate is a separating conjunction of a contiguous sequence of `size` bytes starting at `address`. As we mentioned in the introduction of the thesis, such potentially unbounded sets of permissions in permission logics are modelled using *iterated separating conjunctions*. In Viper, the iterated separating conjunction is written using a quantifier, as shown in our snippet. By using this definition of `MemoryBlock`, we could define `split_memory_block<Pair>` as an unfolding of the `MemoryBlock` for `Pair` followed by folding of memory blocks for its fields as shown in the following snippet.

```
method split_memory_block<Pair>(address: Address)
    requires acc(MemoryBlock(address, size<Pair>), write)
    ensures acc(MemoryBlock(field_address_Pair_first(address), size<i32>), write)
    ensures acc(MemoryBlock(field_address_Pair_second(address), size<i32>), write)
{
    unfold acc(MemoryBlock(address, size<Pair>), write)
    fold acc(MemoryBlock(field_address_Pair_first(address), size<i32>), write)
    fold acc(MemoryBlock(field_address_Pair_second(address), size<i32>), write)
}
```

While this way, we can reduce the number of trusted methods, we decided that keeping assumptions about the layout to the minimum is more important.

```
method main() {
    var a_address: Address

    // StorageLive(a)
    inhale acc(MemoryBlock(a_address, size<Pair>()), write) &&
        acc(MemoryBlockDropStack(a_address, size<Pair>()), write)

    // a = Pair { first: 41, second: 43, };
    assign_composite<Pair>(
        a_address, snap_constructor<i32>(41), snap_constructor<i32>(43))

    // StorageDead(a)
    forget_initialisation<Pair>(a_address)
    exhale acc(MemoryBlock(a_address, size<Pair>()), write) &&
        acc(MemoryBlockDropStack(a_address, size<Pair>()), write)
}

method assign_composite<Pair>(
    address: Address,
    constant_first: Snap<i32>,
    constant_second: Snap<i32>
)
    requires acc(MemoryBlock(address, size<Pair>()), write) && /* ... */
    ensures acc(Own<Pair>(address), write) && /* ... */
{
    split_memory_block<Pair>(address)
    assign_constant(field_address_Pair_first(address), constant_first)
    assign_constant(field_address_Pair_second(address), constant_second)
    fold acc(Own<Pair>(address), write)
}

method forget_initialisation<Pair>(address: Address)
    requires acc(Own<Pair>(address), write)
    ensures acc(MemoryBlock(address, size<Pair>()), write)
{
    unfold acc(Own<i32>(address), write)
    forget_initialisation<i32>(field_address_Pair_first(address))
    forget_initialisation<i32>(field_address_Pair_second(address))
    join_memory_block<Pair>(address)
}
```

Figure 18.6: A simplified encoding of function `main` from Figure 18.4 into Viper.

18.3 Specification Language for Memory Safety

The previous two sections presented how we encode safe types in the updated core proof suitable for verifying mixed safe-unsafe code. Since unsafe code is used in cases where safe code is insufficiently expressive, it is inevitable that, in some cases, the user will have to manually fill in parts of the core proof when verifying an unsafe program. In particular, the users have to be able to specify three things: the conditions necessary to ensure memory safety, the capabilities that are transferred from one function to another during a function call, and the ghost operations used to transform capabilities. Users can specify the conditions necessary to ensure memory safety using *core* specifications such as core preconditions and core postconditions. Core preconditions express the conditions required to ensure memory safety but do not imply properties related to functional correctness, such as the absence of panics. Only unsafe functions are allowed to have core preconditions since only unsafe functions are allowed to rely on preconditions for their memory safety. The key use of core specifications is for expressing capability transfer. For example, by writing *capability predicate* `own!(*p)` in the core precondition, the user expresses that the initialisation capability to the memory referenced by raw pointer `p` is transferred from the caller to the callee. We encode capability predicates into Viper predicates or iterated separating conjunctions of Viper predicates. Users can invoke a ghost operation by calling a corresponding Rust macro.

Table 18.1: Variants of predicates for specifying initialisation and allocation capabilities.

Variant \ Capability	Allocation	Initialisation
Single	`raw!(...)`	`own!(...)`
Range	`raw_range!(...)`	`own_range!(...)`
Set	`raw_set!(...)`	`own_set!(...)`

This section presents the capability predicates and ghost operations supported by our approach. As mentioned earlier, we decided to provide the users with potentially less general but easier-to-use constructs. Therefore, we have three main variants of predicates for specifying capabilities shown in Table 18.1, each with its own ghost operations. In Subsection 18.3.1, we present the simplest variant that enables the user to specify a capability to a *single* memory location. In Subsection 18.3.2, we go from a single memory location to a group of memory locations and present two variants of capability predicates. One variant enables us to specify capabilities for a contiguous *range* of memory locations. The other variant enables specifying capabilities to a *set* of memory locations. Important examples require us to convert from capabilities to a contiguous range of memory blocks to a capability to a memory block encompassing all these smaller blocks and back. We present the necessary ghost operations in Subsection 18.3.3. In Subsection 18.3.4, we present an additional variant needed for specifying rare but important cases that require capabilities to a range of memory locations with holes. In the last subsection, we summarise the presented specification constructs. In this chapter, we assume functions cannot panic; we show how to verify panic safety in Chapter 19.

18.3.1 Allocating a Memory Block

```
fn main() {
    unsafe {
        let layout = Layout::new::<i32>();
        let p_u8: *mut u8 = alloc(layout);
        if !p_u8.is_null() {
            let p_i32: *mut i32 = p_u8 as *mut i32;
            assign(p_i32);
            forget_initialisation!(*p_i32);
            dealloc(p_u8, layout);
        }
    }
}

#[core_requires(raw!(*p, size_of::<i32>()))]
#[core_ensures(own!(*old(p)))]
unsafe fn assign(p: *mut i32) {
    unsafe {
        *p = 42;
    }
}
```

Figure 18.7: A simple Rust program that assigns value 42 to memory allocated on the heap. In this example, we focus on the heap memory and do not show `StorageLive` and `StorageDead` statements.

```
#[core_requires(layout.size() > 0)]
#[core_ensures(
    !result.is_null() ==> (
        raw!(*result, layout.size()) &&
        raw_dealloc!(
            *result, layout.size(), layout.align()
        )
    )
)]
pub unsafe fn alloc(layout: Layout) -> *mut u8;
```

Figure 18.8: Specification of function `alloc`.

In this subsection, we focus on a variant of capability predicates that enables specifying a capability to a single memory location. Our approach supports three kinds of capability predicates that express a capability for a single memory location. Predicate `own!(addr)` expresses the initialisation capability to memory location `addr`. An initialisation capability to a struct can be unpacked into initialisation capabilities to its fields using ghost operation `unpack!(...)` while `pack!(...)` allows packing back the capability. Predicate `raw!(addr, size)` expresses an allocation capability of `size` bytes to a memory location with address `addr`. The user can split and join allocation capabilities using ghost operations `split!(...)` and `join!(...)` respectively. The user can also convert an initialisation capability into an allocation capability using ghost operation `forget_initialisation!(...)`. As mentioned above, we must ensure that memory is deallocated the same way it was allocated; for example, we need to prevent splitting a memory block into two and deallocating the parts separately. Therefore, predicate `raw_dealloc!(addr, size, align)` expresses a capability to deallocate a heap-allocated memory block of a given size and alignment. We illustrate the use of these capability predicates with an example.

Figure 18.7 shows function `main` similar to the one in Figure 18.1, but

```
1 #[core_requires(
2     raw!(*pointer, layout.size()) &&
3     raw_dealloc!(
4         *pointer, layout.size(), layout.align()
5     )
6 )]
7 pub unsafe fn dealloc(pointer: *mut u8, layout: Layout);
```

Figure 18.9: Specification of function `dealloc`.

with the key difference that it allocates the integer manually on the heap instead of allocating it on the stack. Unsafe function `alloc` called on line 7 returns a pointer to a freshly allocated memory block or a null pointer to indicate that the allocation failed. The struct `Layout` passed as an argument to `alloc` is a pair of size and alignment. We specify core preconditions using annotation `#[core_requires(...)]` and core postconditions using annotation `#[core_ensures(...)]`. We express that `alloc` returns the allocation capability to a memory block using predicate `raw!(*result, size)` in its core postcondition as shown on line 2 in Figure 18.8. Function `dealloc` consumes this predicate via its core precondition (line 1 in Figure 18.9) when deallocating the memory block. In addition to the allocation capability, function `alloc` produces and function `dealloc` consumes a deallocation capability expressed by predicate `raw_dealloc!(...)`.

Compared to Figure 18.1, we moved the assignment to the integer into function `assign`, called on line 4. This function consumes an allocation capability to a memory block as specified by predicate `raw!(...)` on line 14. Assigning through pointer `p` on line 18 initialises this memory block and turns it into a valid integer. The function returns the initialisation capability of this value to the caller as specified by predicate `own!(...)` in the core postcondition on line 15. Similarly to memory allocated on the stack, function `dealloc` expects an allocation capability of a memory block. Therefore, on line 8, we use `forget_initialisation!(...)` to manually forget that the value referenced by the pointer is initialised. After converting the initialisation capability to an allocation capability, we can verify the call to function `dealloc` on line 9.

We encode capability predicate `own!(addr)` to the Viper predicate `Own<T>(...)` (`T` is determined based on the type of `addr`), predicate `raw!(...)` to `MemoryBlock(...)`, and `raw_dealloc!(...)` to `MemoryBlockDropHeap`. Ghost operations `unpack!(...)` and `pack!(...)` are mapped to Viper's `unfold` and `fold` statements, respectively. We map the other three ghost operations to the corresponding helper methods on the Viper level.

18.3.2 Capability Groups

In this subsection, we introduce the specification primitives needed for specifying and verifying functions that operate on multiple memory locations. Typically, such functions operate on memory blocks divided into smaller blocks intended to store values of the same type. To specify such functions, we need a capability predicate that expresses capabilities to a contiguous range of memory locations. Therefore, we introduce *range* variants of initialisation and allocation capability predicates. Capability

predicate `own_range!(addr, count)` expresses initialisation capabilities to `count` memory locations starting at `addr`. Capability predicate `raw_range!(addr, count, element_size)` expresses allocation capabilities of size `element_size` to `count` memory locations starting at `addr`. A less common case than functions operating on contiguous ranges are functions that operate on a group of memory locations that do not belong to the same contiguous range. An example of such a function would be a function operating on a doubly linked list implemented with raw pointers. For this case, we introduce *set* variants of initialisation and allocation capability predicates. Capability predicate `own_set!(pset)` expresses initialisation capabilities to all memory locations pointed by pointers stored in mathematical set `pset` (we support mathematical sets by encoding them into Viper sets [183]). Similarly, `raw_set!(pset, element_size)` expresses a set of allocation capabilities.

[183]: Maier (2022), 'Towards Verifying Real-World Rust Programs'

```
#[core_requires(layout.size() > 0)]
#[core_ensures(
    !result.is_null() ==> (
        own_range!(*result, layout.size()) &&
        raw_dealloc!(
            *result, layout.size(), layout.align()
        )
    )
)]
pub unsafe fn alloc_zeroed(layout: Layout) -> *mut u8;
```

Figure 18.10: Specification of function `alloc_zeroed`.

In the previous subsection, we showed how to specify functions that operate on a bounded number of memory locations, which enabled us to specify two out of four allocation primitives exposed by the Rust allocation API. From the remaining two, allocation primitive `alloc_zeroed` allocates a memory block already initialised with zeros. Range capability predicates enable us to specify this function as shown in Figure 18.10. The specification of this function is very similar to the one of function `alloc` with the key difference that instead of predicate `raw!(...)`, we use range capability predicate `own_range!(...)`. `own_range!(...)` in the core postcondition expresses that if the returned pointer is non-null, it points to a sequence of `layout.size()` valid `u8`s (we will present the specification constructs that can be used to express that the memory blocks contain zeros in Chapter 20). We present the ghost operations necessary to verify the clients of such functions in the following subsection.

We encode range and set capability predicates using Viper's iterated separating conjunctions. The encoding is complicated; we discuss it in Section 18.5. We decided to provide range and set capability predicates to the users instead of exposing the general iterated separating conjunction from Viper because the general version requires a deep understanding of the feature, and by providing customised primitives, we can shield the user from some of this complexity.

18.3.3 Reallocating a Memory Block

In the previous subsection, we introduced capability predicates for specifying capabilities to groups of memory locations. In this subsection, we introduce the ghost operations and specification constructs necessary for

```
#[core_requires(old_layout.align() == new_layout.align())]
#[core_requires(old_layout.size() > 0 && new_layout.size() > 0)]
#[core_requires(
    raw!(*ptr, old_layout.size()) &&
    raw_dealloc!(*ptr, old_layout.size(), old_layout.align())
)]
#[core_ensures(
    !result.is_null() ==> (
        raw!(*result, new_layout.size()) &&
        raw_dealloc!(*result, new_layout.size(), new_layout.align()) &&
        forall(|i: usize| i < old_layout.size() && i < new_layout.size() ==>
            old(bytes(*ptr, old_layout.size())[i]) == bytes(*result, new_layout.size())[i]
        )
    )
)]
#[core_ensures(
    result.is_null() ==> (
        // Specification that the original block is returned.
    )
)]
pub unsafe fn realloc(
    ptr: *mut u8,
    old_layout: Layout,
    new_layout: Layout,
) -> *mut u8;
```

Figure 18.11: Specification of function `realloc`.

verifying important patterns such as clients of the fourth Rust allocation primitive: `realloc`. We introduce four new ghost operations. Operation `split_range!(...)` is similar to `split!(...)`; it enables the user to split a single allocation capability into a range of allocation capabilities. Operation `join_range!(...)` is the opposite of `split_range!(...)`; it enables to join a range of allocation capabilities into a single one. Operation `stash_range!(...)` is similar to operation `forget_initialisation!(...)`. `forget_initialisation!(...)` enables to permanently forget that some memory is initialised while `stash_range!(...)` enables to **temporary** convert a range of initialisation capabilities to underlying allocation capabilities. The initialisation capabilities can be recovered using operation `restore_stash_range!(...)` that requires proving that the bytes stored in memory are still the same. We enable specifying that bytes of a memory block are still the same by introducing function `bytes` to access the raw bytes stored in a memory block. We illustrate the newly introduced features using an example based on allocation primitive `realloc`.

Function `realloc` resizes a memory block by deallocating the existing one and allocating a new one of the requested size. The specification of this function is shown in Figure 18.11. The specification mixes `alloc` and `dealloc` specifications. Similarly to `dealloc`, `realloc` consumes `raw!(...)` and `raw_dealloc!(...)` predicates by requiring them in its core precondition. Similarly to `alloc`, `realloc` returns `raw!(...)` and `raw_dealloc!(...)` predicates with `new_layout.size()` if allocation succeeds and `result` is not null. However, differently from `alloc`, if the allocation fails, `realloc` returns the original memory block. A more important difference between `realloc` and the other two functions is that `realloc` has to preserve

```
#[core_requires(cap > 0)]
#[core_requires(
    own_range!(*ptr, cap) &&
    raw_dealloc!(*ptr, cap * size_of::<i32>(), align_of::<i32>())
)]
#[core_ensures(
    !result.is_null() ==> (
        own_range!(*result, cap) &&
        raw_range!(*result.add(cap), 2 * cap, size_of::<i32>()) &&
        raw_dealloc!(*result, 2 * cap * size_of::<i32>(), align_of::<i32>())
    )
)]
#[core_ensures(/* specification for the failure case */)]
pub unsafe fn extend(
    ptr: *mut i32,
    cap: usize,
) -> *mut i32 {
    unsafe {
        stash_range!(*ptr, cap, stash);
        join_range!(*ptr, cap);
        let old_layout = Layout::array<i32>(cap);
        let new_layout = Layout::array<i32>(2 * cap);
        let new_ptr = realloc(
            ptr,
            old_layout,
            new_layout,
        );
        if new_ptr.is_null() {
            // Handle the failure case.
        } else {
            let new_ptr_i32 = new_ptr as *mut i32;
            split_range!(*new_ptr_i32, 2 * cap);
            restore_stash_range!(*new_ptr_i32, stash1);
            new_ptr_i32
        }
    }
}
```

Figure 18.12: Function `extend` that doubles the size of a memory block while preserving the initialisation of existing elements.

values while `alloc` and `dealloc` do not. The quantifier on lines 11–14 of the `realloc` core postcondition expresses that the beginning of the new memory block contains the same values as the old memory block. This property is expressed using function `bytes` that returns a mathematical sequence of bytes stored in the corresponding memory block.

It is crucial for `realloc` to preserve the values of the reallocated memory block because it is typically used for resizing memory that contains information. For example, function `extend` in Figure 18.12 demonstrates the essence of extending the underlying memory block of a data structure such as `ArrayList`. The implementation of `extend` is straightforward: it doubles the size of the given memory block by calling `realloc`. However, `realloc` requires an allocation capability to the memory block while `extend` in its precondition takes a range of initialisation capabilities as shown on line 3. Therefore, we convert the initialisation capabilities into allocation capabilities using ghost operation `stash_range!(...)` on

line 19. The parameters of this operation match the ones of predicate `own_range(...)` except the last one, which is the name that can be used when restoring the stash. Since `realloc` expects a single allocation capability, we additionally join the range of allocation capabilities into a single capability using operation `join_range!(...)` on line 20. After the call to `realloc(...)`, we split the allocation capability into a range of allocation capabilities and restore the knowledge about initialisation to the **new** memory location. Since `realloc` guarantees that the prefix of the newly returned memory block contains the same bytes, restoring initialisation capabilities succeeds.

We encode operations `split_range!(...)` and `join_range!(...)` into Viper using the `split_memory_block_range` and `join_memory_block_range` helper methods. We encode operations `stash_range!(...)` and `restore_stash_range!(...)` using magic wands. Operation `stash_range!(...)` splits an initialisation capability into an allocation capability and a magic wand. The magic wand requires an allocation capability together with a proof that the bytes are still the same and ensures the original initialisation capability. Operation `restore_stash_range!(...)` applies this magic wand to the new allocation capability. We use the name of the stash to identify the magic wand that needs to be applied. We encode the `bytes` function to function `bytes` on the Viper level.

18.3.4 Overlapping Memory Ranges

```
1  #[core_requires(
2      raw_range!(*src, count, size_of::<T>()) &&
3      raw_range!(*dst, count, size_of::<T>())
4  )]
5  #[core_ensures(
6      raw_range!(*src, count, size_of::<T>()) &&
7      raw_range!(*dst, count, size_of::<T>()) &&
8      /* quantifier expressing that values are the same */
9  )]
10 pub const unsafe fn copy_nonoverlapping<T>(
11     src: *const T,
12     dst: *mut T,
13     count: usize
14 );
```

Figure 18.13: Specification of function `std::ptr::copy_nonoverlapping`.

In the previous subsection, we presented the `own_range(...)` and `raw_range(...)` predicates that are well-suited for specifying non-overlapping memory ranges. For example, using them, it is straightforward to specify functions such as `std::ptr::copy_nonoverlapping` from the Rust standard library whose specification is shown in Figure 18.13. However, Rust programmers also use functions such as `std::ptr::copy` that allow copying between *overlapping* ranges. This function is, for example, used for implementing the `Vec::dedup_by` method we mentioned earlier. To support specifying such functions, we also introduce a generalised version of the range predicate that takes a boolean filter instead of the start and end points of the range as shown on lines 4–9 in Figure 18.14. The filter uses function `range_contains(start_pointer, count, checked_pointer)` to check whether `checked_pointer` is between pointers `start_pointer` and

`start_pointer.add(count)`. Similarly to our design choice related to range predicates, we decided to give users the function `range_contains` instead of the complex primitives that can be used to implement it.

```
#[core_requires(raw_range!(*src, count, size_of::<T>()))]
#[core_requires(raw_range!(
    *dst,
    |index| {
        index < count &&
        !range_contains(
            src, count, unsafe { dst.add(index) }
        )
    },
    size_of::<T>()
))]
#[core_ensures(raw_range!(*dst, count, size_of::<T>()))]
#[core_ensures(raw_range!(
    *src,
    |index| {
        index < count &&
        !range_contains(
            dst, count, unsafe { src.add(index) }
        )
    },
    size_of::<T>()
))]
#[core_ensures(/* values are preserved and copied */)]
pub unsafe fn copy<T>(
    src: *const T,
    dst: *mut T,
    count: usize,
);
```

Figure 18.14: Specification of function `std::ptr::copy`.

18.3.5 Summary

Table 18.2 lists all predicates supported by our approach that can be used to specify allocation and initialisation capabilities. Table 18.3 lists all ghost operations for manipulating allocation and initialisation capabilities.

Table 18.2: Predicates for specifying initialisation and allocation capabilities.

Predicate	Explanation	Encoding
`own!(addr)`	Initialisation capabiltiy at `addr`.	Predicate `Own`.
`own_range!(addr, count)`	A sequence starting at `addr` of `count` initialisation capabilities.	Iterated separating conjunction of `Own` predicates.
`own_range!(addr, filter)`	A sequence starting at `addr` of initialisation capabilities for elements for which `filter` returns true.	Iterated separating conjunction of `Own` predicates.
`own_set!(pset)`	A set of initialisation capabilities at locations pointed by pointers stored in set `pset`.	Iterated separating conjunction of `Own` predicates.
`raw!(addr, size)`	Allocation capability of `addr` of size `size`.	Predicate `MemoryBlock`
`raw_range!(addr, count, size)`	A sequence starting at `addr` of `count` allocation capabilities of size `size`.	Iterated separating conjunction of `MemoryBlock` predicates.
`raw_range!(addr, filter, size)`	A sequence starting at `addr` of allocation capabilities for elements for which `filter` returns true.	Iterated separating conjunction of `MemoryBlock` predicates.
`raw_set!(pset, size)`	A set of allocation capabilities at locations pointed by pointers stored in set `pset`.	Iterated separating conjunction of `MemoryBlock` predicates.
`raw_dealloc(addr, size, align)`	A capability to deallocate a heap-allocated memory block.	Predicate `MemoryBlockDropHeap`.

Table 18.3: Ghost operations for manually managing initialisation and allocation capabilities.

Ghost Operation	Explanation	Encoding
`unpack!(addr)`	Unpacks the initialisation capability at `addr`.	Viper `unfold` statement to unfold the corresponding `Own` predicate.
`pack!(addr)`	Packs the initialisation capability at `addr`.	Viper `fold` statement to fold the corresponding `Own` predicate.
`forget_initialisation!(addr)`	Forgets that the memory block is initialised by converting initialisation capability into allocation capability.	`forget_initialisation` helper method to convert the `Own` predicate into the `MemoryBlock` predicate.
`stash!(addr, name)`	Stash the initilisation capability under name `name` revealing the allocation capability.	Package a magic wand from `MemoryBlock` to `Own` predicate.
`restore_stash!(addr, name)`	Restore the stash `name`.	Apply the corresponding magic wand.
`stash_range!(addr, count, name)`	Stash the range of initialisation capabilities under name `name`.	Package a magic wand.
`restore_stash_range!(addr, name)`	Restore the stash `name`.	Apply the corresponding magic wand.
`split!(addr, T)`	Splits the allocation capability at `addr` of size `size_of::<T>()` into allocation capabilites for `T` fields.	`split_memory_block` helper method to split the `MemoryBlock` predicate into `MemoryBlock` predicates for its fields.
`join!(addr, T)`	Joins allocation capabilities of `T` fields into an allocation capability of size `size_of::<T>()`.	`join_memory_block` helper method to join `MemoryBlock` predicates for fields into the `MemoryBlock` predicate for `T`.
`split_range!(addr, count, T)`	Splits the allocation capability of size `count * size_of::<T>()` into a range of allocation capabilities of size `size_of::<T>()`.	`split_memory_block_range` helper method to split the `MemoryBlock` predicate into a range of `count` `MemoryBlock` predicates.
`join_range!(addr, count, T)`	Joins the range of allocation capabilities of size `size_of::<T>()` into the allocation capability of size `count * size_of::<T>()`	`join_memory_block_range` helper method to join the range of `count` `MemoryBlock` predicates into a `MemoryBlock` predicate.

18.4 Types With Invariants

In Section 18.1 and Section 18.2, we explained how our core proof for unsafe code differs from the one presented in Part I for types that are supported by both. In the previous section, we showed specification language extensions that enable us to prove memory safety manually. In this section, we demonstrate how we combine the two to enable verifying code that uses types with invariants as our `ArrayList`. As we mentioned in the introduction of this part, we support types with invariants by extending the automatically generated Viper predicate for that type with the user-written invariant.

```
#[core_invariant(
    size_of::<T>() * self.cap <= (isize::MAX as usize)
    && !self.ptr.is_null()
    && self.len <= self.cap
    && (
        if size_of::<T>() != 0 {
            (self.cap != 0 ==> (
                own_range!(self.ptr, self.len)
                && raw_range!(
                    unsafe { self.ptr.add(self.len) },
                    self.cap-self.len,
                    size_of::<T>()
                )
                && raw_dealloc!(
                    *self.ptr,
                    size_of::<T>() * self.cap,
                    align_of::<T>())
            ))
        } else {
            self.cap == (isize::MAX as usize)
            && own_range!(self.ptr, self.len)
            && raw_range!(
                unsafe { self.ptr.add(self.len) },
                self.cap-self.len,
                size_of::<T>()
            )
        }
    )
)]
pub struct ArrayList<T> {
    ptr: *mut T,
    cap: usize,
    len: usize,
}
```

Figure 18.15: Invariant of `ArrayList`.

Figure 18.15 shows the definition of the `ArrayList` with its invariant written in a `#[core_invariant(...)]` annotation. The invariant stated in Figure 18.15 is more precise than the informal one we gave in Chapter 17. For example, it states that `ArrayList` *owns* the memory block referenced by `ptr` and explicitly handles the case when `T` is a zero-sized type. The invariant also includes `raw_dealloc!(...)` predicate, which enables the implementation of `ArrayList` to call `dealloc` and `realloc` functions needed to implement, for example, `extend` and `drop` methods. Figure 18.16

shows the predicate for `ArrayList`. As discussed in Section 18.2, the predicate includes a predicate for each field of `ArrayList`. Predicate `Own<*mut T>(...)` gives permission only to the pointer itself since raw pointers, unlike references, do not give capabilities to their targets. Using implicit dynamic frames that support heap-dependent expressions enables us to conjoin the invariant to the automatically generated conjuncts. We discuss how we encode functional specifications and specifications that, like our invariant, mix capabilities and functional properties in Chapter 20.

```
predicate Own<ArrayList>(address: Address) {
    acc(Own<*mut T>(field_address_ArrayList_ptr(address)), write) &&
    acc(Own<usize>(field_address_ArrayList_cap(address)), write) &&
    acc(Own<usize>(field_address_ArrayList_len(address)), write) &&
    /* encoded invariant */
}
```

Figure 18.16: Predicate of `ArrayList`.

Figure 18.17 shows an implementation of method `ArrayList::push` that pushes a new element at the end of the list. This method calls method `ensure_sufficient_capacity` on line 3 that ensures that the capacity of the list is sufficient to store the new value. On line 9, the length of the list is increased by one, which breaks the type invariant. The invariant is restored on the following line by writing the value through the pointer referencing the memory block after the last element. Since accessing fields `ptr` and `len` requires having capabilities to them, we unpack `*self` on line 7. Unpacking `*self` also allows us to temporarily break its invariant until we pack `*self` on line 11. In this example, we have omitted the ghost operations required for reasoning about references; we discuss them in Chapter 21.

```
pub fn push(&mut self, value: T) {
    // The following call ensures: self.len < self.cap
    self.ensure_sufficient_capacity();

    // Ghost operations for handling reference self.

    unpack!(*self);
    let element_ptr = unsafe { self.ptr.add(self.len) };
    self.len += 1;
    unsafe { element_ptr.write(value); }
    pack!(*self);

    // Ghost operations for handling reference self.
}
```

Figure 18.17: An implementation of method `ArrayList::push` that pushes a new element at the end of the list with ghost operations necessary to prove memory safety. Method `ensure_sufficient_capacity` ensures that the capacity is sufficient to store an additional element. We have omitted ghost operations related to reference handling; we present them in Chapter 21.

18.5 Memory Addresses

We mentioned above that we use a newly-defined Viper type `Address` for modelling addresses. To fully support raw pointers, we need to choose a model of addresses that supports pointer arithmetic and works well with range and set capabilities. We encode range and set capabilities using Viper iterated separating conjunctions. Reliably supporting range

capabilities requires solving three challenges. We present the challenges and our solution to them in the first three subsections of this section. We believe that the low-level challenges we discuss in this section motivate our design decision to hide the general form of iterated separated conjunction from users well. In the last subsection, we present how we encode set capabilities.

18.5.1 Injectivity Requirement of the Iterated Separating Conjunction

The following snippet shows a general form of iterated separated conjunction supported by Viper.

```
forall vs :: guard(vs) ==>
    acc(P(arguments(vs)), pamount(vs))
```

Here, `vs` is a set of bound variables, `guard(vs)` is a boolean expression, `arguments(vs)` are the expressions that compute the arguments of predicate `P`, and `pamount(vs)` is an expression that computes the permission amount. For this iterated separating conjunction to be well-formed, Viper requires `arguments(vs)` to be **injective** [30][3]. In this subsection, we iteratively derive the model of addresses and encoding of range capability predicates that satisfy the injectivity requirement. In the following subsections, we refine the solution to address two other challenges.

[30]: Müller et al. (2016), 'Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution'

3: If resources are exclusive (like for almost all resources in our case), the injectivity requirement prevents writing iterated separating conjunctions that are trivially false.

First Attempt: Naïve Encoding. The first attempt to model the range capability predicate `raw_range!(pointer, count, size_of::<T>())` could be as shown in the following snippet (to make examples easier to understand, in this section, we omit some type conversions such as converting the value of a pointer to an address).

```
forall index: Int ::
    0 <= index && index < count ==>
    acc(MemoryBlock(
        address + index * size<T>(),
        size<T>()
    ), write)
```

In this snippet, we compute the address of an element by taking the base address and adding an index multiplied by the size of an element. Unfortunately, the element address is not injective with respect to `index` when `T` is a zero-sized type because all element addresses would be equal to the base address.

Second Attempt: Wrapper Predicate Trick. A standard Viper trick used when a location is computed by an expression that is not injective is to hide the computation inside a wrapper predicate. For example, the

following snippet shows a wrapper predicate that depends on the base address and index, and computes the actual address in its body.

```
predicate MemoryBlockWrapper(
    address: Address,
    index: Int,
    size: Snap<usize>
) {
    MemoryBlock(address + index * size, size)
}
```

This predicate can now be used in the iterated separating conjunction instead of the original one, as shown below.

```
forall index: Int :: 0 <= index && index < count ==>
    acc(MemoryBlockWrapper(
        address, index, size<T>()
    ), write)
```

This solution satisfies the injectivity requirements (`index` is trivially injective). However, with this encoding we could only match ranges that start at the same address. For example, using this approach, we could not verify the function shown in the following snippet, which takes a capability to the range of size `count` and splits it at `split_at` into two capabilities for each subrange.

```
#[core_requires(...)]
#[core_requires(raw_range!(*p, count))]
#[core_ensures(raw_range!(*p, split_at))]
#[core_ensures(raw_range!(*result, count-split_at))]
unsafe fn split<T>(
    p: *mut T,
    count: usize,
    split_at: usize,
) -> *mut T {
    p.add(split_at)
}
```

The problem with verifying this function is that the `raw_range!(*result, count - split_at)` predicate uses a different base pointer and, therefore, Viper cannot automatically match the predicate instances. While this function is artificial, the same pattern occurs, for example, in quick sort and merge sort algorithms, and, therefore, it is crucial to support it.

Third Attempt: Injective Addresses. Instead of using a predicate wrapper approach to keep the index explicit, which is necessary for injectivity, we take inspiration from CompCert's memory model [184] and define the address as a triple `(allocation, is_zst, byte_index)`[4]. Here, `allocation` is a unique identifier of the entire allocated block, `byte_index` is an offset into a byte within that block, and `is_zst` is a boolean flag that enables us to special case zero-sized types. Using this interpretation of addresses,

[184]: Leroy et al. (2012), *The CompCert Memory Model, Version 2*

4: We started the work on the model of addresses based on ideas from CompCert together with Olivia Furrer during her bachelor thesis project ([185]: Furrer (2023), 'Verifying Vulnerability Fixes in a Rust Verifier').

we can define offsetting a pointer as the partial function defined by the pseudo-code match statement in the following snippet.

```
function offset_byte(addr: Address, o: Int, s: Snap<usize>): Address {
    match (addr, s) {
        (Address { allocation, is_zst: true, byte_index }, 0) =>
            Address { allocation, is_zst: true, byte_index: byte_index + o }
        (Address { allocation, false, byte_index }, size) if size != 0 =>
            Address { allocation, is_zst: false, byte_index: byte_index + o * s }
    }
}
```

This function takes address `addr`, offset `o`, and the size of the pointer target `s` and computes the offset address. The result of `offset_byte` is unspecified if the function is called with arguments whose zero-sizedness differs (`address` is ZST but `s > 0` or vice-versa). In other words, we treat addresses of non-zero sized values and addresses of zero sized values as separate groups of addresses and do not allow mixing them. We made this design decision because mixing these groups of addresses is likely to be a mistake.

Using function `offset_byte` we can define the range capability predicate `raw_range!(pointer, count, size_of::<T>())` as shown in the following snippet.

```
forall index: Int :: 0 <= index && index < count ==>
    acc(MemoryBlock(
        offset_byte(address, index, size<T>()),
        size<T>()
    ), write)
```

This version is injective with respect to `index` and enables matching capability ranges that start at different offsets.

18.5.2 Non-Linear Arithmetic

The definition of `Address` and function `offset_byte` given at the end of the previous subsection enable us to define `raw_range!(...)` in a way that satisfies the injectivity requirement. However, multiplication of offset and size `o * s` in function `offset_byte` is likely to lead to non-linear arithmetic when the size is statically unknown, which is, for example, the case for type parameters. Non-linear arithmetic is poorly supported by SMT solvers and is likely to lead to unstable verifier behaviour, which is very frustrating for the users. We observed that in many examples, pointer arithmetic is performed on pointers of the same type, meaning their targets are the same size. Therefore, we generalise our approach of treating addresses to zero-sized values and addresses to non-zero-sized values separately to having a group of addresses per size. This way we can mitigate the non-linear arithmetic problem by hiding the multiplication from size behind an additional abstraction. More specifically, we introduce an alternative constructor `new` and partial function `get_index` for `Address` type that are conceptually defined as shown in the following snippet containing pseudo-code. However, in the

actual encoding, we encode them as opaque functions so that the SMT solver does not see the multiplication at all.

```
function Address::new(allocation: Allocation, index: Int, size: Snap<usize>): Address
{
    if size == 0 {
        Address { allocation, is_zst: true, byte_index: index }
    } else {
        Address { allocation, is_zst: false, byte_index: index * size }
    }
}
function Address::get_index(address: Address, size: Snap<usize>): Int
{
    if size == 0 {
        address.byte_index
    } else {
        address.byte_index / size
    }
}
```

Using this wrapper, we can define a partial function `offset` that enables us to compute the offset of a pointer to a type of unknown size without exposing non-linear arithmetic to the SMT solver in many cases. The following snippet shows the definition of `offset` in pseudo-code.

```
function offset(
    addr: Address, o: Int, s: Snap<usize>
): Address {
    Address::new(
        addr.allocation,
        Address::get_index(addr, s) + o,
        s,
    )
}
```

The following snippet shows the updated definition of the range capability predicate `raw_range!(pointer, count, size_of::<T>())` that uses function `offset`.

```
forall index: Int :: 0 <= index && index < count ==>
    acc(MemoryBlock(
        offset(address, index, size<T>()),
        size<T>()
    ), write)
```

This version still satisfies the injectivity requirement but additionally avoids exposing the SMT solver to non-linear arithmetic. We use function `offset` also for defining the meaning of pointer manipulating Rust functions such as `add` we used in our examples.

Unfortunately, avoiding multiplication in the definition of the range capability predicate does not solve the problem completely because users may write multiplication themselves. The type invariant of `ArrayList`

(shown in Figure 18.15) contains conditions as the one shown in the following snippet.

```
size_of::<T>() * self.cap <= (isize::MAX as usize)
```

This condition expresses that the total size in bytes of the memory block used by `ArrayList` is not larger than the largest value that can be stored in an integer of type `isize`. To mitigate this scenario and enable users to express their intent more directly, we add a specification function `array<T>(count)` inspired by the `Layout::array` method that is used by the implementation to compute the actual size. `array<T>(count)` represents the size of an array that contains `count` elements of type `T`. Under the hood, this function maps to an uninterpreted multiplication with axioms tailored for proving scenarios related to memory management. This approach is similar to how verifiers like Dafny [43] deal with non-linear arithmetic. We leave integrating the more general approach used by Dafny to future work.

[43]: Leino (2010), 'Dafny: An Automatic Program Verifier for Functional Correctness'

18.5.3 Quantifier Triggers

Our definitions of iterated separating conjunctions up to now omitted one aspect that is crucial for automation: triggers [186]. A *trigger* is a syntactic pattern that tells the SMT solver when to instantiate the quantifier during the proof search. Viper uses triggers to guide an SMT solver also when reasoning about iterated separated conjunctions [30]. The following snippet shows a definition of predicate `raw_range!(...)` with the trigger shown between the curly braces.

[186]: Detlefs et al. (2005), 'Simplify: a theorem prover for program checking'

[30]: Müller et al. (2016), 'Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution'

```
forall index: Int ::
    { offset(address, index, size<T>()) }
    0 <= index && index < count ==>
    acc(MemoryBlock(
        offset(address, index, size<T>()), size<T>()
    ), write)
```

The trigger `offset(address, index, size_of::<T>())` tells the SMT solver to instantiate the iterated separating conjunction every time it finds a term matching `offset(address, ?, size_of::<T>())`. The subterm that matches `?` is used as a value of `index` when instantiating the iterated separating conjunction. For example, the following snippet shows a Rust function that writes to the fifth element using pointer arithmetic.

```
#[core_requires(raw_range!(ptr, 10, size_of::<i32>()))]
fn write_to_five(ptr: *mut i32) {
    *(ptr.add(5)) = 42;
}
```

Call `ptr.add(5)` gets encoded to `offset(address, 5, size_of::<T>())`, which matches the trigger with `index = 5` and, therefore, the verifier can prove that the assignment is justified. However, if we try to write to an element without calling the `add` method, as shown in the following

snippet, the verification fails because there is no suitable term to trigger the iterated separated conjunction.

```
#[core_requires(raw_range!(ptr, 10, size_of::<i32>()))]
fn write_to_five(ptr: *mut i32) {
    *ptr = 42;
}
```

This snippet illustrates why choosing triggers is crucial for SMT-based verification. Since we use the `raw_range!(...)` predicate to specify that we have permissions to a `MemoryBlock` predicate at some address, the most flexible trigger would be `MemoryBlock(element_address)`. However, SMT triggers are required to mention all bound variables. Therefore, we have to write our iterated separating conjunction defining `raw_range!(...)` in a way that quantifies over addresses instead of indices. We can achieve this goal using the function `range_contains` we used before when discussing the `raw_range!(...)` variant with a filter in Subsection 18.3.4. The following example shows our final encoding of predicate `raw_range!(...)`.

```
forall element_address: Address ::
    { MemoryBlock(element_address, size<T>()) }
    range_contains(address, count, element_address) ==>
        acc(MemoryBlock(
            element_address, size<T>()
        ), write)
```

In this iterated separating conjunction, we are quantifying over addresses of elements and use function `range_contains` to check that they are within the required range. We define `range_contains` as shown in the following snippet.

```
function range_contains(address, count, size, checked_address): Bool
{
    address.allocation == checked_address.allocation &&
    address.is_zst == checked_address.is_zst &&
    Address::get_index(address, element_size) <=
        Address::get_index(checked_address, element_size) &&
    Address::get_index(checked_address, element_size) <
        Address::get_index(address, element_size) + count
}
```

`own_range!(...)` is defined analogously with predicate `Own<T>(...)` instead of `MemoryBlock(...)`. We define the variants of `raw_range!(...)` and `own_range!(...)` with a boolean filter in the same way as regular ones, but with the guard replaced with the filter. Since the filter takes the index and not the address of the element, when encoding the filter, we replace the index with an expression that computes it from the element address.

18.5.4 Set Predicates

Up to now, we focused on range capability predicates. As we mentioned above, the set capability predicate `raw_set!(pset, size)` expresses allocation capabilities to all memory blocks referenced by pointers stored

in mathematical set `pset`. Supporting set capability predicates is much easier than range capability predicates. The following snippet shows how we encode `raw_set!(pset, size_of::<T>())` using iterated separating conjunction.

```
forall address: Address ::
    { MemoryBlock(address, size<T>()) }
    address in pset ==>
        acc(MemoryBlock(address, size<T>()), write)
```

The injectivity requirement is automatically satisfied since we use the quantified variable `address` directly as the argument. We encode `own_set!(pset)` in an analogous way using predicate `Own<T>` instead of `MemoryBlock`.

In this chapter, we presented the new core proof together with specification constructs suitable for verifying mixed safe and unsafe code. In the next chapter, we show how we use our new core proof to verify the safety of abstractions.

Verifying Memory Safety and Functional Correctness of Safe Abstractions

19

In the introduction of this part, we mentioned that we aim to support two verification goals. First, we want to enable verifying functional correctness of mixed safe-unsafe code. Second, we want to enable verifying that safe abstractions guarantee memory safety when used by unverified safe clients. As we mentioned in Chapter 17, these two goals require us to model, for example, Rust assert statements differently, which we address by verifying each function in two different modes. We explain the two modes in Section 19.1. The mode for addressing the first goal of verifying functional correctness is very similar to what we used in Part I. Therefore, in the remainder of this chapter, we focus on the challenges related to our second goal of ensuring that safe abstractions are memory safe. In Chapter 17, we discussed the key techniques programmers use to ensure that safe abstractions maintain their invariants. In Section 19.2, we show how we verify that these techniques were used correctly. We cover the additional challenges caused by panic safety and how we support drop handlers used for fixing invariants in Section 19.3. We finish this chapter by discussing how we support verifying drop handlers in Section 19.4.

19.1 Two Verification Modes

As we mentioned above, we support our two verification goals (functional correctness and memory safety) by verifying each function in two different modes: *functional-correctness* and *memory-safety*. In these two modes, we differently handle the following four properties: memory safety, functional correctness, absence of panics, and absence of arithmetic errors such as overflows. We first discuss how the two modes differ when verifying safe functions and then extend the explanation to also include unsafe functions and core specifications (we call a function safe if it is declared as safe even when it contains unsafe blocks inside it).

For safe functions, the functional-correctness mode is effectively the same what we used in Part I: we verify that assuming a precondition, the function does not panic, does not cause arithmetic errors, and if it terminates it ensures its postcondition. Verifying in memory-safety mode is more interesting. Since our goal is to verify that a safe function is memory safe against unverified safe clients, we can rely only on properties that are ensured by the Rust compiler: that function parameters are valid instances of their types. In particular, we cannot rely on functional preconditions when verifying memory safety of safe functions. For example, method `ArrayList::index` is declared as a safe function. Therefore, it must guarantee memory safety even when called with an out-of-bounds index. Since postconditions can typically be established only when preconditions hold, we do not verify functional postconditions in memory-safety mode by default. When verifying a function in memory-safety mode, we do not check that the preconditions of called safe functions are upheld because of

two reasons. First, a safe function cannot cause a memory safety violation even if its precondition is violated. Second, to establish the precondition of the called function, it is often necessary to assume the precondition of the caller, which we do only when verifying in functional-correctness mode. Since without establishing the precondition of the called function, its postcondition is not guaranteed to hold, in memory-safety mode we do not assume the postcondition. To summarize, when verifying a function declared as safe in memory-safety mode, we by default ignore all functional specifications: both the preconditions and postconditions of the function being verified, as well as preconditions and postconditions of the called safe functions. One consequence of ignoring functional specifications is that the verified and called functions may panic. As we discussed in Chapter 17, safe abstractions use runtime assertions such as bounds checks to prevent memory errors. Therefore, when verifying in memory-safety mode, we treat panics as early returns instead of verifying that they do not occur. Arithmetic errors pose a unique challenge when verifying in memory safety mode. We typically cannot prove that they do not occur without relying on the functional precondition and we do not know what exactly happens if an error occurs. In Rust, an arithmetic error such as overflow can either cause a panic or produce an unexpected value depending on the compiler settings. Therefore, when verifying a function in memory-safety mode, we require that the function guarantees memory safety in both cases: when an arithmetic error leads to a panic and when it leads to an incorrect value. We model the incorrect value case by storing into the result of the arithmetic operation an unknown value that is a valid instance of the type.

Unsafe functions, unlike safe functions, *are* allowed to rely on clients ensuring certain properties for their memory safety. In Chapter 18, we presented core specifications that programmers can use to express the conditions necessary to ensure memory safety of unsafe functions. When verifying an unsafe function in memory-safety mode, we assume its core precondition and require it to ensure its core postcondition when the function does not panic. When verifying a call to an unsafe function (from either safe or unsafe function), we check its core precondition and assume its core postcondition. When verifying an unsafe function in functional-correctness mode, we verify it assuming both core and functional preconditions and require it to establish both core and functional postconditions. While distinguishing between core and functional specifications for unsafe function sounds complicated, the users need to worry about the distinction only if the verified unsafe function distinguishes the two cases. If the distinction for a specific function is not relevant, the user can simply mark all specifications as core.

There are two important exceptions to the overall approach we described so far. The first exception are functions that fully check their preconditions with runtime assertions. If such a function does not panic, it guarantees its postcondition. To support this case, we introduce an annotation `#[no_panic_ensures_postcondition]`. When verifying a function annotated with this annotation, we verify its functional postcondition even in the memory-safety mode. As a result, the caller of such a function can also assume its functional postcondition in the memory-safety mode. The second exception are unsafe functions that guarantee properties even when they panic. For example, if an unsafe function received a

capability to some memory from the caller, it often needs to return that capability even if it panics. To support this case, we provide a special *panic core* postcondition `#[panic_core_ensures(...)]` (inspired by exceptional postconditions in [187] and [188]) that the unsafe function ensures when it panics.

[187]: Leino et al. (2004), 'Exception Safety for C#'

[188]: Leavens et al. (2011), *JML Reference Manual*

19.2 Maintaining Type Invariants

In Chapter 17, we discussed that safe abstractions like our `ArrayList` rely on type invariants to ensure their memory safety. For example, method `ArrayList::index` in Figure 17.2 relies on field `ptr` pointing to a sequence of `len` initialised values to ensure that the reference it returns is valid. Therefore, it is crucial to guarantee that these type invariants hold. As we mentioned in Chapter 17, the general approach programmers use to maintain a type invariant of a struct like `ArrayList` is ensuring that client code cannot break the invariant and that the implementation of the abstraction maintains it. The former is ensured by making the fields of the struct private while the latter is ensured by checking that all methods that can modify the fields preserve the invariant.

In our approach, we make this intuitive method used by programmers more precise. As described in Chapter 18, users can specify a type invariant using `#[core_invariant(...)]` annotation. The type invariant is then conjoined to the automatically generated predicate for that type. Predicates in permission logics like implicit dynamic frames are required to be *self-framing*. Intuitively, this means that the predicate (and thus the invariant written by the user) cannot constrain memory locations, which it does not own. This property guarantees that the predicate cannot be invalidated without unfolding it. As a result, we need to only verify code that could directly modify the memory locations owned by the predicate. The predicate owns two kinds of locations: fields of the struct and locations pointed at by raw pointers. The former can be directly modified by all safe and unsafe code to which access is granted by the Rust visibility rules. This reason is why we require all fields of a struct with a user-specified invariant to be private and consider the code that has direct access to them as part of the safe abstraction that needs to be verified. In Rust, all functions that are declared in the same module as the struct can read its private fields and, therefore, should be considered part of the safe abstraction. The locations pointed at by raw pointers can be modified only by unsafe code because dereferencing a raw pointer is an unsafe operation. As a result, safe clients cannot violate the invariant constraining memory locations referenced by raw pointers (as we noted before, ensuring memory safety against unverified unsafe clients is a non-meaningful goal because such a client can in general break type system properties in arbitrary ways; luckily, most client code is written in safe Rust).

We require the type invariant to hold each time a value is moved, copied, or borrowed. This requirement implies that function definitions are allowed to assume that all parameters are valid instances of their types (because arguments are moved into a function when it is called), but are also required to ensure this property when calling other functions or when returning values (also when a panic occurs). Requiring values

to be valid when they are moved, copied, or borrowed is stricter than, for example, the visible state semantics [189] used in other verification approaches based on invariants. However, we chose this design because it is consistent with how Rust treats primitive values: moving a value of type `bool` that does not store a valid bit-pattern for `bool` is undefined behaviour [70]. We enforce the requirement by encoding move, copy, and borrow operations to require a packed version of a capability (a folded version of a predicate), which implies that the invariant holds.

[189]: Müller et al. (2006), 'Modular invariants for layered object structures'

[70]: Developers (2023), *The Rustonomicon: What Unsafe Rust Can Do*

19.3 Ensuring Panic Safety

In Chapter 17, we presented two ways a panic can lead to a memory safety error if it is raised when a type invariant is temporarily violated. First, a panic causes drop handlers to be executed, which can lead to a drop handler observing the broken invariant. Second, if the type whose invariant is broken was obtained from the caller via a mutable reference, the caller may observe the broken invariant through the original reference. The general approach for ensuring type invariants presented in the previous section prevents both of these scenarios. However, with the approach we presented so far, we cannot verify some correct code that uses the three techniques to deal with panics we mentioned in Chapter 17. In the following, we explain how we enable verifying code that uses each of the techniques.

19.3.1 Avoiding Panics

Panics can be avoided in two ways: if the called function promises to never panic or if the called function's functional precondition is ensured, which implies that it will not panic. Below we discuss each of these two cases.

Never Panicking Functions. Examples of functions that never panic would be many pointer manipulating functions in the standard library such `std::ptr::read` and `std::ptr::write` that are called by the `std::mem::swap` implementation we showed in Section 17.3. We repeat the example below.

```
// SAFETY: exclusive references are always valid to
// read/write, including being aligned, and nothing
// here panics so it's drop-safe.
unsafe {
    let a = ptr::read(x);
    let b = ptr::read(y);
    ptr::write(x, b);
    ptr::write(y, a);
}
```

In our technique, a user can specify that a function never panics by annotating it with `#[no_panic]`[1]. When verifying the implementation of such a function in memory-safety mode, we check that all panic branches are unreachable. When verifying a call to such a function, we

1: Annotation `#[no_panic]` is equivalent to `#[panic_core_ensures(false)]` but expresses the intent more clearly.

correspondingly assume that the function does not panic even if its functional precondition does not hold.

Non-Panicking Function Calls. The annotation `#[no_panic]` enables us to express that a specific function never panics. However, we sometimes need to show that a specific function call is guaranteed not to panic. We can prove that a called function is guaranteed not to panic by proving that its functional precondition holds since the functional precondition implies the absence of panics. We enable the user to temporarily enable checking of preconditions in memory safety mode by wrapping code in `checked!` block as shown in the following snippet.

```
1  if list.len() > 0 {
2      let first_element = checked! { list.index(0) };
3      // ...
4  }
```

The called method `ArrayList::index` panics if the index is out-of-bounds. Since the if condition guarantees that index `0` is in bounds, we can prove the precondition of `ArrayList::index` even in memory-safety mode and, therefore, be sure that it never panics.

Within `checked!{...}` block, function calls are always checked in functional-correctness mode: function preconditions are checked, postconditions are assumed, and panic branches are omitted.

19.3.2 Preserving Invariants

Figure 19.1 shows method `override_from_input` from Section 17.3 we used to motive the technique. Before calling `read_into`, which potentially panics, the method sets the `len` field of `*self` to zero. This change ensures that in the case of a panic, the invariant is already in a valid state. The call to the unsafe method `read_into` requires capability to the memory block pointed at by `*self.ptr`. This capability is stored in `*self` invariant and to access it, we need to `unpack!(*self)` as shown on line 7. (We also need to perform some ghost operations related to references that are indicated by `...` on line 6; we discuss them in Chapter 21.) When the control flow exits the method, our approach requires `*self` to be packed. We can easily do that when no panic occurs by adding a ghost statement `pack!(...)` at the end of the method. However, there is no place where we could add statement `pack!(...)` when a panic occurs. We enable performing ghost operations on panic branches by providing two new constructs: block `before_drop!{v => ...}` and block `after_drop!{v => ...}`. Block `before_drop!{v => ...}` enables executing ghost code before the drop handler of `v` is called. Block `after_drop!{v => ...}` enables executing ghost code after the drop handler is called. In our example, we put the `pack!(...)` operation in `before_drop!{g => ...}` as shown on line 9. Since drop handlers are executed even when panics occur, this approach enables us to pack `*self` on a panic.

The example also shows how to deal with a situation when the example has no value with a drop handler that is called in the right moment. We define our own ghost value (line 8) using empty struct `GhostDrop` (line 18)

with an empty drop handler (line 19) and attach the ghost operations to it.

```
1  pub fn override_from_input<T: Input>(
2      &mut self,
3      input: &mut T,
4  ) {
5      unsafe {
6          // ...
7          unpack!(*self);
8          let g = GhostDrop;
9          before_drop! { g =>
10             pack!(*self);
11             // ...
12         }
13         self.len = 0;
14         let len = read_into(input, self.ptr, self.cap);
15         self.len = len;
16     }
17 }
18 struct GhostDrop;
19 impl Drop for GhostDrop {}
```

Figure 19.1: Function `override_from_input` from Section 17.3 that we used to motivate the preserving the invariant technique with specifications necessary to verify memory safety.

19.3.3 Fixing Invariants

To support the previous technique, we used a drop handler to execute ghost code when a panic occurs. The last of the three techniques is similar in that it uses a drop handler to execute code that fixes the invariant when code panics. Figure 19.2 shows a fragment of method `Vec::from_iter` from the Rust standard library. In this fragment, a drop guard is created on line 4 and then potentially some panicking code is executed (line 5). If the code does not panic, the drop handler is cancelled by forgetting the drop guard (line 6) and an instance of the vector is constructed using unsafe constructor `Vec::from_raw_parts` (line 7). If the code panics, the drop handler is called and uses the unsafe constructor (lines 17–19) to create an instance of the vector that is then safely dropped. The unsafe constructor is an unsafe function that consumes capabilities to the memory referenced by raw pointer `ptr`. These capabilities must be provided by the drop handler, which must take them from the caller. In other words, the implementation of the drop handler behaves like an unsafe function: it depends on the caller ensuring its requirements to guarantee memory safety.

As we discussed in Section 17.3, treating the drop handler as an unsafe function violates behavioural subtyping. A drop handler is declared as a safe method on trait `Drop` whose precondition is `true` (it requires only a valid instance of a type). We resolve this conflict by special-casing drop handlers defined on private structs that are never used as arguments to generic functions, which ensures that they are never called via the `Drop` trait without knowing the concrete implementation that is going to be executed. We treat drop handlers on such structs as unsafe functions, which, for example, enables users to write core preconditions.

```
unsafe fn from_iter(
    /* ... */
) -> Vec<T> {
    let dst_guard = InPlaceDstBufDrop { ptr, len, cap };
    /* ... */
    std::mem::forget(dst_guard);
    unsafe { Vec::from_raw_parts(ptr, len, cap) }
}
struct InPlaceDstBufDrop {
    ptr: *mut T,
    len: usize,
    cap: usize,
}
impl Drop for InPlaceDstBufDrop {
    fn drop(&mut self) {
        unsafe {
            Vec::from_raw_parts(
                self.ptr, self.len, self.cap
            )
        };
    }
}
```

Figure 19.2: A simplified fragment of method `std::vec::Vec::from_iter` from the standard library illustrating a drop handler that consumes capabilities from the context in which it is called.

19.4 Drop Handlers

As was mentioned in Section 17.3, the signature of a drop handler in Rust is a lie: it says that the value pointed at by `self` should be valid when the drop handler returns while that is clearly not the intention because the drop handler could not deallocate memory in that case. The precise rules for what should and should not be allowed in a drop handler are still under discussion. For example, it is not clear whether the fields of a dropped struct must be valid instances of their types when the drop handler returns. However, it is clear that the drop handlers must be able to deallocate heap-allocated memory. Therefore, we aim for a conservative option that requires the fields to be valid instances of their types but allows the core invariant of the type to be violated when the drop handler returns. We implement this conservative option in our approach by special-casing the encoding of drop handlers. The drop handler starts execution with an initialisation capability to `*self` and is required to ensure initialisation capabilities of each field in `*self` when it returns.

Making Verification Lightweight | 20

Up to now, we have focused on defining an updated version of core proofs suitable for verifying mixed safe-unsafe code and showing how it can be used for verifying safe abstractions. One of the goals of this part of the thesis is to make verification lightweight. In Part I, we significantly reduced the verification effort by automatically generating the core proof and enabling users to use a simple specification language based on Rust expressions. We achieved this by leveraging the fact that Rust type system forbids unrestricted mutable aliasing in safe Rust. However, raw pointers enable unrestricted mutable aliasing and, therefore, the approach based on place capabilities we used in Part I does not apply to unsafe code. In this chapter, we show how we can regain lightweight verification even in the presence of raw pointers. In Section 20.1, we focus on automatically generating the core proof of functions that contain both safe and unsafe code. As we mentioned in the introduction of this part, our solution is based on the observation that non-aliasing requirements intended to enable the Rust compiler to optimise code enable us to slice a function into managed and non-managed parts. Therefore, we are able to generate the core proof for the managed part automatically. In Section 20.2, we present how we change the encoding of functional specifications to maintain the simple specification language from Part I for safe code while enabling the new specification features needed for specifying mixed safe and unsafe code. This section also shows how we use snapshots to integrate specifications with the unsafe core proof.

20.1 Place Capabilities for Safe-Unsafe Code

As we discussed in Chapter 18, a crucial part of a core proof is the ghost operations that transform the capabilities to justify the actions the code performs. In Part I, we used the PCS elaboration algorithm to compute all necessary such operations. Since raw pointers enable mutable aliases, we cannot expect to design an algorithm that can automatically compute all needed ghost operations for all Rust code that may use raw pointers. One option is to let the user write all necessary ghost operations if a function contains any unsafe code. This option would greatly increase the required verification effort and, therefore, is undesirable. Another option is to try designing heuristic for inferring ghost operations that would be as complete as possible. However, debugging why a complex heuristic failed can be extremely frustrating to the user. Therefore, we decided to aim for a middle ground: we use a few simple rules to slice each function into managed and non-managed parts. For the managed part, we use a modified version of the PCS algorithm from Chapter 2 to automatically compute the necessary ghost operations. For the non-managed part, we require the user to write the ghost operations. We designed the rules in such a way that we over-approximate the non-managed part assuming

[126]: Jung et al. (2020), 'Stacked borrows: an aliasing model for Rust'

[127]: Villani (2023), 'Tree Borrows'

[128]: Villani (2023), *Tree Borrows: A new aliasing model for Rust*

1: Currently, we do not check that the code adheres to Stacked Borrows or Tree Borrows. In case code does not adhere to one of these models, we may slice the function incorrectly. However, incorrect slicing may only need to a confusing verification error about a missing permission due to the PCS elaboration algorithm inserting an unexpected operation.

the code adheres to either Stacked Borrows [126] or Tree Borrows [127, 128][1]. In Subsection 20.1.1, we show two scenarios where using the PCS elaboration algorithm on an entire function could lead to surprising verification errors. In Subsection 20.1.2 and Subsection 20.1.3, we present how our technique addresses each of the scenarios. In Subsection 20.1.4, we discuss the changes we made to the PCS elaboration algorithm from Chapter 2 to support the unsafe core proof.

20.1.1 Challenges Caused by Aliasing

A key property of place capabilities is that they are identified by *syntactic* places. For example, after executing the following snippet, exclusive capabilities are associated with syntactic places `a.first` and `a.second`.

```
let mut a = Pair { first: 1, second: 2 };
PCS: {a → E}
unpack a
PCS: {a.first → E, a.second → E}
```

This property enabled us to create the PCS elaboration algorithm (presented in Chapter 2) we used for computing PCS operations. However, once we have raw pointers, we cannot anymore identify capabilities using syntactic places as shown by the following snippet.

```
let mut a = Pair { first: 1, second: 2 };
PCS: {a → E}
let x = addr_of_mut!(a);
PCS: {???}
unpack ???
```

Raw pointer `x` aliases `a` and both can be used to mutate the underlying memory location. Therefore, we could say that both `a` and `*x` share the same capability (which we cannot call anymore "place capability" because it is not associated with any single place). In permission logics such as implicit dynamic frames and separation logic, capabilities (more specifically, access permissions) are typically associated not with a syntactic place, but with an address. Since both `*x` and `a` have the same address, this model can naturally capture code that uses raw pointers. However, since the same capability can be modified via multiple syntactic names, we cannot anymore use our simple syntactic approach to track its state. Since in the example shown above `x` is directly assigned to alias `a`, we could still track the state of the capability by remembering which syntactic places alias each other. However, with pointer arithmetic, tracking of aliases quickly becomes complicated. For example, either of `y` or `z` in the following snippet could be aliases of `b`.

```
let mut a = Pair { first: 1, second: 2 };
let mut b = Pair { first: 3, second: 4 };
let x = addr_mut!{a};
let y = x.add(1);
let z = 0x42 as *mut Pair;
```

Without being able to track aliases, we also cannot track the state of capabilities. If we tried to apply the PCS elaboration algorithm when

it cannot track precise state of capabilities, we could get surprising verification errors. There are two scenarios in which this problem could occur, which we illustrate with examples below.

Local Aliases. In the following snippet, we initialise local variable `a` of type `Pair`, create raw pointer `x` pointing into it, manually unpack the capability referenced by `x`, and finally try to assign `a.first` to `b`.

```
let mut a = Pair { first: 1, second: 2 };
let x = addr_of_mut!(a);
unpack!(*x);
let b = a.first;      // Verification error.
```

If we run the PCS elaboration algorithm on this example and then try to verify it, we get a surprising verification error on the last line saying that the capability to `a` is missing. The verifier reports the error because the PCS elaboration algorithm seeing access to `a.first`, tries to unpack capability `a`, which fails because the capability was already unpacked via alias `x`. Such surprising errors can be very frustrating to the user and, therefore, it is important to avoid them.

Hidden Capabilities. In the following snippet, we create an instance of our safe abstraction `ArrayList`, store it in variable `list`, unpack it, create raw pointer `element_ptr` to an element of `list`, pack `list`, and try to dereference `element_ptr`.

```
let list: ArrayList = // ...
unpack!(list);
let element_ptr = list.ptr.add(index);
pack!(list);
let a = unsafe { *element_ptr };    // Verification error.
```

Since `list` is packed before `element_ptr` is dereferenced, the verifier does not see the capability that justifies dereferencing and reports a verification error. In this snippet, the user can see operation `pack!(list)` in the code and conclude that it is causing the error. However, if operation `pack!(list)` were inserted automatically, understanding the problem could be hard and frustrating for the user.

20.1.2 Local Aliases

In the previous subsection, we saw that we do not have a reliable way to track the state of capabilities potentially aliased by raw pointers. Therefore, we require the user to manually write ghost operations for all capabilities that could potentially be aliased by raw pointers. As a result, we need to distinguish which capabilities can be aliased by raw pointers and which cannot. The PCS elaboration algorithm described in Chapter 2 manages capabilities identified by *safe* places, that is, variables[2] (for example, `a`), field accesses (for example, `a.f.g`), and dereferences of references (for example, `*x` and `(*a.g).f`). We use the following rule to determine whether a safe place could be aliased by a raw pointer: a safe place is considered as potentially aliased by a raw pointer only if the

2: Function parameters are also variables.

function contains an expression that creates a raw pointer aliasing that place.

To make it easier to explain why we picked such a rule, we split it into two parts:

1. A local variable or a field access `p` is considered potentially aliased by a raw pointer only if the function contains an expression (either `addr_of!(p)` or `addr_of_mut!(p)`) that creates a raw pointer of it.
2. A mutable reference `r` is considered aliased by a raw pointer only if the function contains an expression (either `r as *const _` or `r as *mut _`) that casts the reference to a raw pointer. (Since shared borrow capabilities are duplicable, we avoid the challenges caused by aliasing by associating a copy of the capability with each syntactic place.)

The first part of the rule is justified by the fact that the Tree Borrows and Stacked Borrows models forbid using raw pointers to access memory to which a raw pointer was never created. The goal of the Tree Borrows and Stacked Borrows models is to enable the compiler to use non-aliasing information for optimisations. Since unsafe code can break the types and the Rust compiler cannot reliably determine the boundaries of unsafe code, it cannot use type information for optimisations. Therefore, Jung et al. [126] proposed that, instead of relying on the type system, the compiler should rely on operational semantics for optimisations. More specifically, Jung et al. proposed to define an operational semantics that would clearly state which behaviours are allowed and which ones lead to *undefined behaviour* (UB). When optimising, the compiler can assume that UB does not occur. For example, if a compiler observes that index `i` is used for an unchecked access of an array element, it can assume that the value of `i` is between 0 and the array's length.

[126]: Jung et al. (2020), 'Stacked borrows: an aliasing model for Rust'

Stacked Borrows and the later model Tree Borrows require the use of raw pointers to adhere to *provenance* rules. Simply speaking, a raw pointer can be used to access a location only if it was somehow derived from it (using `addr_of_mut!(...)`, casting a reference to a raw pointer, from another valid pointer to that location, or some other way). For example, it is illegal to use `y` to access `b` in the following snippet even though we know that `y` is valid inside the if statement.

```
let mut a = 1;
let mut b = 2;
let x = addr_of_mut!{a};
let y = x.add(1);
if y == addr_of_mut!{b} {
    *y = 42;    // Undefined Behaviour!
}
```

Requiring that a pointer was somehow derived from a location is not sufficient to ensure that the pointer was created in the function we are verifying, which is the property we want to have. There are two possible ways how this property could be violated. First, a value and a pointer pointing into it could be passed to a function as arguments as shown in the following snippet.

```
call_aliased(addr_of!(a), a);
```

Second, a called function could derive a pointer from a passed reference and return to the caller as shown in the following snippet.

```
let x = &mut a;
let p: *mut Pair = into_raw_pointer(x);
```

Stacked Borrows and Tree Borrows ensure that the first part of the rule works even in these two cases. The first case is prevented because in Stacked Borrows and Tree Borrows, moving a value invalidates all raw pointers pointing to it as shown in the following snippet. In this snippet, we create raw pointer `x` to local variable `a`, move out `a` into `b`, restore `a` by moving back from `b`, and finally dereference `x` that references now valid again `a`. However, the dereference of `x` still causes undefined behaviour because moving out `a` invalidated the pointer.

```
let mut a = Pair { first: 1, y: 2 };
let x = addr_of!(a);
let b = a;  // Moving out a.
a = b;      // Moving back.
let z = unsafe { (*x).first };  // Undefined behaviour!
```

Since function arguments are moved from the caller to the callee, and, similarly, the result is moved from the callee to the caller, the values obtained from other functions cannot have any valid pointers referencing them.

The case in which a called function derives a raw pointer from a passed reference and returns it to the caller is allowed in Stacked Borrows and Tree Borrows. However, in these models, expiring a reference invalidates all the pointers derived from it. For example, in the following snippet, assignment `a = 42` invalidates not only mutable reference `x`, but also the raw pointer `y` that was derived from it, making the assignment on the last line illegal.

```
let mut a: i32 = 1;
let mut x = &mut a;
let y = x as *mut i32;
a = 42;
unsafe { *y = 2; } // Undefined Behaviour!
```

As a result, when variable `a` is used again, we are sure that no valid aliases exist anymore.

The second part of the rule is justified by Stacked Borrows and Tree Borrows for the common case of non-nested references. These models allow using raw pointers only when the mutable borrow from which they were directly derived is active. For example, in the following snippet, we create reference `x` to variable `a`, cast reference `x` into raw pointer `p`, reborrow reference `x` with reference `y`, dereference raw pointer `p`, and dereference reference `y`.

```
let x = &mut a;
let p = x as *mut Pair;
let y = &mut *x;
let b = unsafe { (*p).first };  // Expires y.
let c = y.first;    // Undefined behaviour!
```

Since `p` was created from `x`, reborrowing `x` makes it unusable as long as reborrowing reference `y` is alive. Therefore, when we dereference `p`, we automatically expire `y` and, as a result, later dereferencing `y` causes undefined behaviour.

When a function is called all non-nested references in its parameters are implicitly reborrowed guaranteeing that there are no valid aliases for the duration of the function[3]. Similarly, returning a reference from a function also reborrows.

3: There are exceptions related to two-phase borrows. However, our model of references does not support two-phase borrows yet and the semantics of two-phase borrows are still one of the more active areas of discussion, which makes it hard to predict the final rules.

Stacked Borrows and Tree Borrows do not guarantee non-aliasing for nested borrows because these models aim to allow unsafe code to redefine the meaning of types, which is required in some cases. For example, the code shown in the following snippet is allowed by these models.

```
unsafe fn modify(x: & &mut Pair) {
    unsafe {
        let p: *mut *mut Pair = std::mem::transmute(x);
        (**p).first = 42;
    }
}
```

The surprising part of this example is that it mutates a value behind a shared reference. Our current verification approach does not allow verifying such examples. We leave extending our approach from both core proof design and automation points of view to support such examples to future work.

20.1.3 Safe Abstractions

The second scenario we showed in Subsection 20.1.1, in which automatically inferred ghost operations can lead to surprising verification errors, is hiding a capability inside a core invariant by packing. We avoid this scenario using the following rule: if a type has a core invariant, then the user is required to unpack and pack capabilities to values of this type manually.

It is important to note that values of types that contain fields of types with core invariants are still managed automatically as long as the field with invariant is not unpacked. For example, the following snippet shows struct `TreeNode` that stores children nodes in `ArrayList` and a method that replaces children nodes with new ones.

```
struct TreeNode<T> {
    value: T,
    children: ArrayList<Node<T>>,
}
impl<T> TreeNode<T> {
    fn replace_children(&mut self, new_children) {
        self.children = new_children;
    }
}
```

As expected for a completely safe method, all necessary ghost operations can be computed automatically even though `children` is of type `ArrayList` that has a core invariant. The presented rule enables us to compute the capability operations automatically for clients that use only safe APIs of values with core invariants even when the same function contains (unrelated) unsafe code.

20.1.4 Supporting Mixed Safe-Unsafe Code in the PCS Elaboration Algorithm

The PCS elaboration algorithm we presented in Part I exploits the fact that place capabilities are identified by syntactic places to compute the PCS operations automatically. Figure 20.1 on the following page shows a safe Rust code snippet with elaborated place capabilities and PCS operations, which were elaborated using the algorithm from Part I. As we mentioned in Part I, the algorithm does the forward pass over all statements, computing the capabilities before and after each statement. The algorithm ensures that a processed statement has the required capabilities by inserting the necessary PCS operations that transform the current capabilities into the required ones. For example, the assignment on line 8 requires *exclusive* capability to place `pair2.second`. Therefore, the algorithm inserts the `unpack pair2` operation on line 10 to unpack the capability `pair2` into capabilities `pair2.first` and `pair2.second`. We must make three important changes to adapt this PCS elaboration algorithm to support mixed safe-unsafe code. First, we need to integrate the rules we discussed in previous subsections to distinguish which capabilities should be managed by the algorithm and which should be left to be managed by the user. Second, we need to adapt the algorithm to generate the unsafe core proof we presented in Chapter 18 because, even for entirely safe code, the unsafe core proof is more detailed than the one presented in Part I. Importantly, we need to change the algorithm to distinguish between allocation and initialisation capabilities because we had only initialisation capabilities in Part I. Third, in safe code, the algorithm has precise knowledge of available capabilities, while in mixed safe-unsafe code, it needs to be changed to work with only partial knowledge. To avoid presenting too many details at once, we exploit the fact that the changes are orthogonal and discuss each change in its subsubsection. More specifically, in Subsubsection 20.1.4.1, we present how we integrate the rules using the capabilities from Part I. Then, in Subsubsection 20.1.4.2, we show how we handle the new capability types in our algorithm. Finally, in Subsubsection 20.1.4.3, we discuss how we address the challenge of having only partial information.

20.1.4.1 Managed and Non-Managed Capabilities

In the previous two subsections, we motivated the rules that enable us to decide for which capabilities we can use the PCS elaboration algorithm. To summarise, we require the user to manually write ghost operations that transform capabilities in the following cases:

1. For capabilities that are identified by dereferencing raw pointers.

```
1  PCS: {}
2  let mut pair1 = Pair { ... };
3  PCS: {pair1 → E}
4  let mut pair2 = Pair { ... };
5  PCS: {pair1 → E, pair2 → E}
6  unpack pair2
7  PCS: {pair1 → E, pair2.first → E, pair2.second → E}
8  pair2.second = 1;
9  PCS: {pair1 → E, pair2.first → E, pair2.second → E}
10 unpack pair1
11 PCS: {pair1.first → E, pair1.second → E, pair2.first → E,
   pair2.second → E}
12 pair1.second = 2;
13 PCS: {pair1.first → E, pair1.second → E, pair2.first → E,
   pair2.second → E}
14 pack pair1
15 PCS: {pair1 → E, pair2.first → E, pair2.second → E}
16 let pair3 = pair1;
17 PCS: {pair3 → E, pair2.first → E, pair2.second → E}
```

Figure 20.1: A safe Rust example with elaborated PCSs and PCS operations.

2. For safe places to which a raw pointer was created in the same function.
3. For places whose types have core invariants.

The capabilities to which these rules apply we call *non-managed* capabilities. The algorithm uses rule-specific approaches to determine whether a rule applies to a specific capability.

To determine whether rule (1) applies, we only need to check whether the capability is identified by a raw pointer. This check does not require keeping any state in the algorithm. Therefore, we omit such capabilities from the algorithm's state. Figure 20.2 illustrates our approach. The example shown in Figure 20.2 is a variation of the example from Figure 20.1. Compared to Figure 20.1, the example in Figure 20.2 has an additional statement on line 6 that creates raw pointer `x` referencing `pair1`. After this statement, the capability can be modified via both `*x` and `pair1`[4]. Since `*x` is a dereferencing of raw pointer `x`, we omit it from the algorithm's state as can be seen on line 7, which shows that the state contains only the capability to the pointer `x.pointer`[5], but not to its target `x.*`.

Rule (2), unlike rule (1), is path sensitive as Figure 20.3 illustrates. In this example, a raw pointer to place `a` is only created if the value of `b` is `true`. As a result, the capability identified by location `a` should be managed by the algorithm only if `b` was false. Therefore, we need to track whether the capability is managed in the algorithm's state. For this reason, we introduce a *raw-borrowed* (RB) capability to track locations that are known to have potentially raw pointers pointing to them. As shown on line 7 in Figure 20.2, after creating a raw pointer to `pair1`, its capability is changed to RB. In Tree Borrows, once a raw pointer is created pointing to a place, the raw pointer can be used to modify that place until the place gets deallocated. Therefore, `pair1` has the capability RB until the end of the method to indicate that it is not managed (moving `pair1` to `pair3` on line 18 does not deallocate the memory location of `pair1`). As shown

4: The Tree Borrows model allows modifying via both `pair1` and `*x` while Stacked Borrows allows only via `*x`. In our approach, we rely only on guarantees that both models provide.

5: For clarity, we use the same notation for raw pointers as we used for references.

```
1  PCS: {}
2  let mut pair1 = Pair { ... };
3  PCS: {pair1 → E}
4  let mut pair2 = Pair { ... };
5  PCS: {pair1 → E, pair2 → E}
6  let x = addr_of_mut!(pair1);
7  PCS: {x.pointer → E, pair1 → RB, pair2 → E}
8  unpack pair2
9  PCS: {x.pointer → E, pair1 → RB, pair2.first → E,
   pair2.second → E}
10 pair2.second = 1;
11 PCS: {x.pointer → E, pair1 → RB, pair2.first → E,
   pair2.second → E}
12 unpack!(pair1);
13 PCS: {x.pointer → E, pair1 → RB, pair2.first → E,
   pair2.second → E}
14 pair1.second = 2;
15 PCS: {x.pointer → E, pair1 → RB, pair2.first → E,
   pair2.second → E}
16 pack!(*x);
17 PCS: {x.pointer → E, pair1 → RB, pair2.first → E,
   pair2.second → E}
18 let pair3 = pair1;
19 PCS: {x.pointer → E, pair1 → RB, pair3 → E,
   pair2.first → E, pair2.second → E}
```

Figure 20.2: A mixed safe-unsafe Rust example with elaborated PCSs and PCS operations.

```
1  if b {
2      let x = addr_of_mut!(a);
3      // ...
4  }
5  // a is managed capability depending on whether b is true
```

Figure 20.3: An example illustrating that sometimes whether a capability is managed can depend on the path.

on line 12, user-written pack and unpack operations do not affect the RB capabilities. In the PCS elaboration algorithm presented in Part I, we always merged the state of incoming branches. In the version for mixed code, we merge the state only if that can be done without loss of information (otherwise, we remember what state we had when coming from a specific branch). This way, we can provide a better user experience because we require fewer annotations from the user.

Rule (3) is also path sensitive like rule (2). Therefore, we handle it similarly. Figure 20.4 shows a snippet in which a field of variable `list` is read. Since `list` is of a type that has a core invariant, according to rule (3), we require the user to manually unpack the capability as shown on line 3. Instead of producing the nested capabilities, unpacking converts the capability of `list` to *unpacked* (U) as shown on line 4. The U capability informs the PCS elaboration algorithm that when the user packs `list` back, it should restore the E capability. This way, the PCS elaboration algorithm can manage the data structures containing safe abstractions as fields.

```
1  let mut list: ArrayList = ...
2  PCS: {list → E}
3  unpack!(list);
4  PCS: {list → U}
5  let len = list.len;
6  PCS: {list → U, len → E}
7  pack!(list);
8  PCS: {list → E, len → E}
```

Figure 20.4: An example illustrating rule (3).

20.1.4.2 Allocation and Initialisation Capabilities

In the previous subsubsection, we showed how to distinguish managed capabilities from non-managed. This subsubsection explains what kind of managed capabilities and PCS operations we must handle to generate the unsafe core proof. In Chapter 18, we discussed capabilities and operations we need to add to support mixed safe-unsafe code. Many of these capabilities and operations are also needed for constructing the unsafe core proof of completely safe Rust programs (because the unsafe core proof is more detailed than the safe core proof). Therefore, the PCS elaboration algorithm has to compute them. The capabilities that our PCS elaboration algorithm does not handle because we cannot automatically determine whether they are aliased or not are capabilities to heap-allocated memory, range capabilities, and set capabilities.

In Part I, for programs without references, we needed only one kind of capability: *exclusive*. This capability expresses that a place is allocated and initialised. As we mentioned earlier, in unsafe core proof, we need to model allocation and initialisation separately. Therefore, in Chapter 18, in addition to *initialisation* (I) capability that expresses that a memory location is allocated and initialised, we also introduced *allocation* (A) and *drop-stack* (DS) capabilities. The *allocation* capability indicates that the place is allocated and is mapped to a `MemoryBlock` predicate. The *drop-stack* (DS) capability expresses that the corresponding place is allocated on the stack and can be deallocated. These capabilities and the operations that manipulate them are computed analogically to how the PCS elaboration algorithm for safe code computes *exclusive* capabilities and corresponding PCS operations.

Figure 20.5 on the next page demonstrates how the algorithm works on a simple example. In this example, a variable `pair` is created, its field `first` is moved out into variable `a`, and then both variables are deallocated. Before a variable is initialised, it gets allocated by `StorageLive` (lines 1 and 5), which results in *allocation* capabilities (A) added to the state as shown on lines 2 and 6. Initialising variable `pair` converts its *allocation* capability (A) into *initialisation* capability (I) as shown on line 4. Unpacking and packing of *initialisation* capabilities works in the same way as in the safe version (lines 6–8). Like in the safe version, unpacking is encoded into Viper's `unfold` and packing to Viper's `fold`. `StorageDead(a)` on line 13 requires an *allocation* capability to `a` but we have *initialisation* capability. Therefore, the PCS elaboration algorithm inserts `forget-initialisation a` operation on line 11, which gets encoded into calling the corresponding helper method. `StorageDead(pair)` on 19 also requires *allocation*. To satisfy this requirement, the PCS elaboration algorithm needs to insert not only `forget-initialisation pair`, but also

`memory-block-join pair` operation (line 17). As mentioned in Chapter 18, split and join operations are similar to unfold and fold operations but are for allocation capabilities.

```
1  // StorageLive(pair)
2  PCS: {pair → A}
3  let pair: Pair = ...
4  PCS: {pair → I}
5  // StorageLive(a)
6  PCS: {a → A, pair → I}
7  unpack pair
8  PCS: {a → A, pair.first → I, pair.second → I}
9  let a: BigInt = pair.first;
10 PCS: {a → I, pair.first → A, pair.second → I}
11 forget-initialisation a
12 PCS: {a → A, pair.first → A, pair.second → I}
13 // StorageDead(a)
14 PCS: {pair.first → A, pair.second → I}
15 forget-initialisation pair.second
16 PCS: {pair.first → A, pair.second → A}
17 memory-block-join pair
18 PCS: {pair → A}
19 // StorageDead(pair)
20 PCS: {}
```

Figure 20.5: An example showing how the PCS elaboration algorithm elaborates allocation and initialisation capabilities.

20.1.4.3 Handling Partial Knowledge

Since the algorithm presented in Part I is executed on a type-checked safe Rust program, it is guaranteed to always succeed in finding a way to produce the capabilities of the desired shape. This property no longer holds when we run the algorithm on mixed safe-unsafe code. For example, in the following snippet, the user manually unpacks `a`, which changes its capability to *unpacked*. As a result, the PCS elaboration algorithm cannot satisfy the requirement to provide the capability to `a.f`, which is needed for the assignment to `b`.

```
1  unpack!(a);
2  let b = a.f;
```

Therefore, if the PCS elaboration algorithm cannot satisfy the requirements, it proceeds under the assumption that it succeeded in satisfying them. This behaviour is sound because the verifier tracks capabilities precisely and will report a verification error if some capability is missing.

20.2 Functional Specification of Safe-Unsafe Code

In Part I, we presented how we can exploit the non-aliasing guarantees available in safe Rust to provide a simple specification language based on Rust expressions. We want to maintain this lightweight specification

while enabling the specification of mixed safe-unsafe code. This goal requires us to solve two challenges. First, we need a new way of linking the user-written specifications to the core proof. In Part I, we modelled Rust primitive types using primitive Viper types. For example, we modelled a value of type `i32` as a field of type `Int` wrapped in a predicate. This choice allowed us to translate specifications directly into heap-dependent expressions that can be conjoined to encoded type capabilities. In our new unsafe core proof, we model primitive types using untyped memory blocks and, therefore, need a different approach for linking functional specifications to capabilities. We present our solution to this challenge in Subsection 20.2.1. Second, for functional specification to be well-defined, each memory access in the specification has to be justified with a corresponding capability. In Part I, we used the PCS elaboration algorithm to find the capabilities that justify each memory access in the functional specification. Since the PCS elaboration algorithm cannot be used with raw pointers, we need an alternative technique for finding the capabilities that justify dereferencing them. We present in Subsection 20.2.2 a lightweight extension to our specification language from Part I that enables showing that dereferencing a raw pointer in the functional specification is justified.

20.2.1 Encoding of Functional Specification

In this section, we show how we use snapshots to link functional specifications to the unsafe core proof. As we mentioned in Section 5.2, a snapshot is a mathematical value that fully captures the set of values stored in a group of heap locations. We decided to base our specifications on snapshots because (as we discussed in Section 5.2), they enable quantifying over non-primitive types and returning non-primitive types from functions. However, when designing the snapshots we have to solve two challenges. First, when handling unsafe code, we have to consider invalid values. Therefore, unlike in Part I, we have to consider snapshots that can be invalid instances of their types. As we mentioned in Chapter 18, we address this challenge by introducing a validity function `is_valid` for each type, that returns true if a snapshot is a valid instance for that type. For example, snapshot `s` is a valid instance of `i32` if it is an integer whose value is between `i32::MIN` and `i32::MAX`. Second, we have to link snapshots to predicates. For this purpose, we use `snap` functions, which we showed in Section 5.2, that take a permission to a predicate and return a snapshot that represents the value stored in the predicate. The following snippet shows the `snap` function for Rust type `i32`.

```
function snap<i32>(address: Address): Snap<i32>
    requires acc(Own<i32>(address), wildcard)
    ensures is_valid<i32>(result)
{
    unfolding acc(Own<i32>(address), write) in
        from_bytes<i32>(bytes(address, size<i32>()))
}
```

As discussed in Section 18.1, `bytes` is a heap-dependent function that returns the bytes of a memory block, and function `from_bytes` is a mathematical function that converts bytes to a (potentially invalid) snapshot of

an `i32` value. Since having an instance of predicate `Own` signifies that the value is valid, `snap` functions can ensure that the snapshot they return is a valid instance of the type. The keyword `wildcard` used as a permission amount signifies that this function needs any positive permission amount to the predicate.

The following snippet shows a simple function with a precondition that expresses that the values stored at the targets of pointer `p` and parameter `a` are positive.

```
#[core_requires(own!(*p) && *p > 0 && a > 0)]
unsafe fn require_positive(p: *mut i32, a: i32) { }
```

On the definition side, this precondition is encoded as the two `inhale` statements shown in Figure 20.6. The `inhale` statement on line 1 inhales the permissions generated from the parameter types. The `inhale` statement on line 2 inhales the core precondtion. The function call `snap<*mut i32>(p)` obtains the snapshot of predicate instance `Own<*mut i32>(p)`. The function call to function `value<*mut i32>` obtains the actual address out of the snapshot. This address is passed to function call `snap<i32>`, which obtains the snapshot of `i32`. Finally, the mathematical value stored in this snapshot is extracted by calling `value<i32>` and compared to `0`. The encoding of `a > 0` is analogous.

```
1 inhale acc(Own<*mut i32>(p), write) && acc(Own<i32>(a), write)
2 inhale acc(Own<i32>(value<*mut i32>(snap<*mut i32>(p))), write) &&
3         value<i32>(snap<i32>(value<*mut i32>(snap<*mut i32>(p)))) > 0 &&
4         value<i32>(snap<i32>(a)) > 0
```

Figure 20.6: An inhale of permissions to parameters and core precondition.

In the rest of this section, we describe for each kind of type how we define their snapshots and validity functions. To make examples clearer we present snapshots using pseudo-Viper syntax with algebraic data types (ADTs); in our actual implementation, we manually axiomatise them to have a more optimised encoding.

Primitive Types. We define snapshots of primitive types such as `bool`, `i32`, `char`, and `*mut T` by using an ADT that is either a valid value or uninit as shown in the following snippet for <i32>.

```
adt Snap<i32> {
    Value(Int)
    Uninit
}
```

As was mentioned above, a destructor `value<i32>` allows extracting the Viper value out of the snapshot (`Bool` for `bool`, `Int` for integer types and `char`, `Address` for raw pointers).

`is_valid<bool>` returns true if snapshot is an ADT variant `Value`. `is_valid` for integer types and `char` requires the snapshot to be `Value(v)` where `v` is required to be within the valid range for that type. `is_valid` for raw pointers is an uninterpreted function, which means that a user cannot prove that some sequence of bytes is a valid pointer; valid raw pointers can be obtained only through built-in Rust operations.

Generic Type Parameters. Since generic type parameters are modelled using abstract predicates, their snapshot types, and validity functions, and `snap` functions are also abstract.

Structs. We define the snapshot of a struct similarly to a snapshot of a primitive type, just with a parameter for each field. The following snippet shows the definition of a snapshot for type `Pair`.

```
adt Snap<Pair> {
    Value(first: Snap<i32>, second: Snap<i32>)
    Uninit
}
```

A snapshot of a struct is valid if it has ADT variant `Value` and all of its fields are also valid. The `snap` function obtains the snapshots of all fields and combines them to obtain the snapshot of the struct. The following snippet shows a `snap` function for `Pair`.

```
function snap<Pair>(address: Address): Snap<Pair>
    requires acc(Own<Pair>(address), wildcard)
    ensures is_valid<Pair>(result)
{
    unfolding acc(Own<Pair>(address), write) in
        Snap<Pair>::Value(
            snap<i32>(...first...),
            snap<i32>(...second...),
        )
}
```

Structs with Core Invariants. When defining a snapshot of a struct with an invariant, we treat each synctactically mentioned initialisation predicate as if it were an additional field. For example, the following snippet shows a definition of `MyBox<T>` with its core invariant.

```
#[core_invariant(!self.ptr.is_null() ==> own!(*self.ptr))]
struct MyBox<T> {
    ptr: *mut T,
}
```

The snapshot for this type is shown in the following snippet.

```
adt Snap<MyBox<T>> {
    Value(ptr: Snap<*mut T>, ptr_deref: Snap<T>),
    Uninit,
}
```

The additional parameter `ptr_deref` captures the value of the predicate referenced by pointer `ptr`. The validity function requires such fields to contain valid snapshots only when the corresponding capability is present in the invariant. For our example, `ptr_deref` is ADT variant `Uninit` when field `ptr` is null. The `snap` function, similarly to structs without invariants, obtains snapshots of each real or fictional field and creates the snapshot of the entire struct. However, its implementation becomes significantly more complicated because it has to consider all possible

initialisation variants as shown in the following snippet containing the `snap` function for `MyBox<T>`.

```
function snap<MyBox<T>>(address: Address): Snap<MyBox<T>>
    requires acc(Own<MyBox<T>>(address), wildcard)
    ensures is_valid<MyBox<T>>(result)
{
    unfolding acc(Own<MyBox<T>>(address), write) in
        (...ptr.is_null() ?
            Snap<MyBox<T>>::Value(
                snap<*mut T>(...ptr...),
                Snap<T>::Uninit,
            )
            :
            Snap<MyBox<T>>::Value(
                snap<*mut T>(...ptr...),
                snap<T>(...),
            )
        )
}
```

Since core invariants can also contain `own_range(...)` and `own_set(...)` predicates, we also need to define their snapshots. For `own_range(...)`, we use Viper sequences and for `own_set(...)` we use Viper sets.

Enums. We define a snapshot of an enum by having a variant for each variant of the enum and an additional variant to represent an invalid enum. For example, the `Option<T>` type in Rust is shown in the following snippet.

```
enum Option<T> {
    Some(T),
    None,
}
```

We define the snapshot of `Option<T>` as shown in the following snippet.

```
adt Snap<Option<T>> {
    ValueSome(Snap<T>),
    ValueNone,
    Uninit,
}
```

A snapshot of an enum is valid if it is one of the enum variants and the arguments of the variant are all valid. The `snap` function for enums is defined analogously to the one for structs with the key difference that we need to consider each variant separately as shown in the following

snippet.

```
function snap<Option<T>>(address: Address): Snap<Option<T>>
    requires acc(Own<Option<T>>(address), wildcard)
    ensures is_valid<Option<T>>(result)
{
    unfolding acc(Own<Option<T>>(address), write) in
        (... discriminant == 0 ?
            Snap<Option<T>>::ValueSome(snap<T>(...))
            :
            Snap<Option<T>>::ValueNone
        )
}
```

In this subsection, we presented snapshots, which we use to link memory accesses in specifications to capabilities that justify them. In the following subsection, we discuss how can we justify raw pointer dereferences in specifications.

20.2.2 Specifying Memory Referenced by Raw Pointers

The specification technique presented in Part I enabled us to use a specification language based on Rust boolean expressions as shown in the following snippet.

```
#[requires(r1.first == r2.first)]
unsafe fn test(r1: &Pair, r2: &Pair) { }
```

Such specifications are enabled by the PCS elaboration algorithm that for each memory access in the specification finds a capability that justifies it. The following snippet shows an alternative version of the example that uses raw pointers instead of shared references.

```
#[core_requires(own!(*p1))]
#[core_requires(p1 != p2 ==> own!(*p2))]
#[requires((*p1).first == (*p2).first)]
unsafe fn test(p1: *mut Pair, p2: *mut Pair) { }
```

In this version, we have two memory accesses `(*p1).first` and `(*p2).first` that need to be justified. However, we cannot use the PCS elaboration algorithm anymore since it cannot handle raw pointers.

For statements, we solved this challenge by giving to the user ghost operations such as `unpack!(...)` that enable them to manually transform the capabilities into the required form in the cases when the PCS elaboration algorithm cannot do that. We could approach the challenge for specifications in the same way by providing a ghost operation `unpacking!(...)` that enables the user to manually unpack the capability inside a specification. However, this can lead to verbose and tedious to write specifications in cases a user needs to unpack a capability multiple times; for example, to unpack capability `*p` into `*p.f.g.h`, they would need three nested `unpacking!(...)` operations. Therefore, we decided to enable the PCS elaboration algorithm to track capabilities to memory locations referenced by raw pointers instead.

In the previous section, we showed several examples how changing a capability through one alias causes a verification error when using another alias. The key problem in all these examples is that all aliases are using the same exclusive capability to a memory location and, therefore, changes through one alias affect all other aliases. Rust's shared references also allow aliasing, but each shared reference is associated with a duplicable read-only capability that is limited in time (the capability expires when the lifetime of the corresponding reference expires). This property enabled us to treat the capability of each shared reference separately in Part I. We exploit this observation to enable using the PCS elaboration algorithm for specifications that contain raw pointer dereferences.

The key difference between capabilities used to justify specifications and capabilities used to justify user-written code is that the former never consumes the capabilities: it only requires proving that a capability is held and that it can be transformed into the required form. Therefore, when justifying specifications, we can pretend that all capabilities are duplicable, which enables us to treat each capability separately. We introduce a new specification construct `eval_using!(own!(p) => assertion)` that tells the PCS elaboration algorithm that when elaborating operations in `assertion`, it can assume that syntactic place `p` has an initialisation capability associated with it. The following snippet shows the updated version of the snippet from before that uses `eval_using!(...)` to tell the PCS elaboration algorithm that `*p1` and `*p2` have initialisation capabilities associated with them.

```
#[core_requires(own!(*p1))]
#[core_requires(p1 != p2 ==> own!(*p2))]
#[requires(
    eval_using(own!(*p1) ==>
        eval_using(own!(*p2) ==>
            (*p1).first == (*p2).first
        )
    )
)]
unsafe fn test(p1: *mut Pair, p2: *mut Pair) { }
```

As can be seen from the snippet, the fact that `eval_using(...)` does not consume the capability enables us to avoid branching on whether `p1` and `p2` alias when writing the functional part of the specification.

While `eval_using(...)` enables us to specify functional properties of functions that use raw pointers, we can often simplify these specifications. Often functional specifications are written immediately after predicates that provide the capabilities that justify these specifications as shown in the following example.

```
#[core_requires(
    !p.is_null() ==> own!(*p) && (*p).f.g == 5
)]
```

We interpret such examples as if `eval_using(...)` is following the predicate as shown in the following snippet.

```
#[core_requires(!p.is_null() ==>
    own!(*p) && eval_using(own!(*p) => (*p).f.g == 5))]
```

This simple syntactic sugar avoids the need to use `eval_using(...)` in most cases making most functional specifications for code that uses raw pointers not much more complex than the ones for safe code. With this syntactic sugar, expressions like `(*p).f.g` look similar to Viper's heap dependent expressions. However, there are two important differences. First, in our specification language, places are matched syntactically while Viper determines the corresponding capability based on the actual value identifying the resource. Second, our PCS algorithm automatically ensures that capabilities are transformed into the right shape while Viper requires the user to do that manually by providing the necessary `unfolding` expressions.

It is important to note that `eval_using(...)` cannot be used to specify permissions that are hidden inside type invariants. For example, the following snippet shows struct `MyBox` that owns the memory location referenced by field `p`.

```
#[core_invariant(!self.p.is_null() ==> own!(*self.p))]
pub struct MyBox<T> {
    p: *mut T,
}
```

If we have a function that takes a reference to this box, `eval_using!(...)` unfortunately cannot be be used to specify the value stored in the memory location referenced by field `p` because the syntax does not allow specifying that predicate `own!(...)` is inside the core invariant of the box. We considered extending the syntax to allow specifying (potentially arbitrary long) chain of places whose invariants contain the permissions. However, we decided against this extension because it leads to hard-to-read specifications and the desired specifications can be expressed by calling a pure function on a type with invariant.

21 Mutable and Shared Borrows

Borrowing is one of the key features of Rust and is used constantly by programmers. Therefore, all verification techniques developed for Rust have to support borrowing to be useful. While the focus of this part of the thesis is on the verification of mixed safe-unsafe code, to enable evaluating our approach on interesting examples, this chapter presents our technique for supporting references. Based on our discussion in Chapter 7 of approaches for modelling references, we have two options we could use as a base for developing a model of references suitable for SMT-based verification of unsafe code. The first option would be to build on our model based on magic wands we presented in Part I. The key challenge we would have to address to enable verifying unsafe code in this model is adding a model of lifetimes, which are extensively used by safe abstractions to ensure memory safety. The second option would be to base our work on the models of references presented in RustBelt [71] and RustHornBelt [133]. The advantage of these models is that they were designed for verifying unsafe code. However, as we discussed in the introduction of this part, RustBelt and RustHornBelt were designed for manual verification in a proof assistant and their approaches would have to be adapted to enable SMT-based verification. We believe that both options are viable. However, we decided to base our approach on RustBelt and RustHornBelt because they are more different from our work and we expect that working on them can give us a different view and deeper understanding of Rust references and borrowing.

As we discussed in Chapter 7, the tracking of borrows in RustBelt is done by the lifetime logic, the library that implements the "bank". An important property of RustBelt's design is that the lifetime logic is designed as a separable component. This property enables us to integrate the unchanged lifetime logic with our approach, which we investigated with Pascal Huber in his Master's thesis [190]. Since the focus of RustBelt is verifying the soundness of safe abstractions, the lifetime logic does not provide any mechanism for specifying the values of borrows, which is necessary for verifying functional correctness. Verification of functional correctness is enabled by RustHornBelt. However, RustHornBelt enables functional verification by tracking prophecies outside of the lifetime logic, which exposes the complexity of the mechanism for tracking prophecies in the core proof making it challenging to reliably automate in an SMT-based verifier[1]. Therefore, instead of trying to automate RustHornBelt directly, we took a different path: we extended the interface of the lifetime logic in a way that enables RustHornBelt-like reasoning. The key advantage of our approach is that the resulting logic is significantly more suited for automating in an SMT-based verifier. The design of our lifetime logic enables us to completely automatically derive a proof in our new lifetime logic for programs written in safe Rust, giving the users exactly the same experience as in Part I. For unsafe code, we provide annotations that the user can use to guide the generation of the proof.

[71]: Jung et al. (2018), 'RustBelt: securing the foundations of the Rust programming language'

[133]: Matsushita et al. (2022), 'RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code'

[190]: Huber (2022), 'Automatically Generating Memory Safety Certificates for Rust Programs'

1: Creusot, an SMT-based verifier based on the prophecies approach, does not model RustHornBelt resources and does not support verifying unsafe code.

We mentioned that we decided to base our approach on the lifetime logic because this path is more different from the work we did in the rest of the thesis and can give us a deeper understanding of Rust references and borrowing. Up to now, almost all[2] operations for transferring and transforming resources were either Viper's primitive operations (for example, Viper's `unfold` statement) or operations that could be decomposed into them (for example, the `forget_initialisation` ghost operation that can be decomposed into recursively unfolding all predicates and joining all memory blocks). Therefore, the soundness of these operations depends on the soundness of Viper. Our lifetime logic introduces new operations for transferring and transforming resources, which are axiomatised without relating them to primitive Viper operations. Therefore, to gain the same level of trust, we need to prove that the lifetime logic is sound and that we are modelling it correctly in Viper. We are convinced that our lifetime logic is sound because it is based on RustBelt's lifetime logic with extensions guided by RustHornBelt's design, both of which were proven sound in Iris [191] (and mechanised in Coq [58]). We are also convinced that our modelling of the logic in Viper is correct because we took the standard approach of modelling lifetime logic resources as opaque Viper resources and lifetime logic rules as opaque Viper methods. However, we do not yet have the formal proofs and, therefore, consider the work presented in this chapter incomplete.

2: The main exception is the modelling of shared references in Part I. Other exceptions are operations for type parameters (because to decompose some operations we need to know a concrete type) and operations for splitting and joining memory blocks (because to decompose these operations we would need to choose a concrete memory layout).

[191]: Jung et al. (2018), 'Iris from the ground up: A modular foundation for higher-order concurrent separation logic'

[58]: Barras et al. (1997), *The Coq proof assistant reference manual: Version 6.1*

This chapter is structured as follows. In Section 21.1, we present how we model RustBelt's lifetime logic in Viper and how we integrate it with our core proof presented in Chapter 18. In Section 21.2, we show how we use information from Polonius and our PCS elaboration algorithm to automatically derive a core proof for safe Rust functions using references. In Section 21.3, we present our extensions of the lifetime logic and show how we use them to enable verification of functional correctness in programs that use references. In Section 21.4, we summarise.

21.1 Modelling RustBelt's Lifetime Logic in Viper

As we discussed in Part I, when reference `x` expires in Rust, the capability that the reference is borrowing (`x.*`) is returned to the lender. In our model of borrows based on magic wands we presented in Part I, we reconstruct the **explicit** backward flow that returns the borrowed capabilities to the lenders. As we discussed in Subsection 7.5.1 (Borrowing via Library in RustBelt) on page 87, in RustBelt, the capabilities are returned **implicitly**, similarly to the Rust compiler.

This section shows how we can model RustBelt's lifetime logic, which is defined in the higher-order framework Iris [191], in Viper, which uses a first-order logic. In this section, we focus on modelling the elements of RustBelt's lifetime logic in Viper. Since the same elements of RustBelt's lifetime logic are used for verifying safe and unsafe code, we use only safe Rust examples in this section. In the following section, we show how the information available in the compiler can be used to compute all the necessary operations for safe code examples automatically and how the user can guide the generation of the proof for examples using unsafe code.

[191]: Jung et al. (2018), 'Iris from the ground up: A modular foundation for higher-order concurrent separation logic'

We start this section by repeating the intuition of how the lifetime logic works in Subsection 21.1.1. In Subsection 21.1.2, we present the general approach we use to model the lifetime logic in Viper. In Subsection 21.1.3, we illustrate our encoding by walking through a simple example. We finish this section by discussing in Subsection 21.1.4 important challenges related to modelling RustBelt borrows in Viper.

21.1.1 Library-Based Modelling of Borrows

As we discussed in Subsection 7.5.1, the lifetime logic can be understood as a library written in a linear functional programming language that implements a bank-like entity. *Creating a borrow* stores the borrowed capability in the bank for the lifetime for which the borrow was created. When the borrow is created, the bank issues a borrow capability and inheritance. The *borrow capability* can be used to access the borrowed capability as long as the corresponding lifetime is alive, which is tracked using the *lifetime token*. The *inheritance* can be used to take out the borrowed capability from the bank once the lifetime is dead, which is tracked using the duplicable *dead lifetime token*. For mutable borrows, the borrow capability is unique while, for shared borrows, the borrow capability is duplicable. To access the borrowed capability, the user has to *open* the borrow. Opening the borrow consumes the borrow capability and a fraction of the lifetime token. The user can regain the borrow capability and the consumed fraction of the lifetime token by *closing* the borrow. *Creating a lifetime* generates a full lifetime token for that lifetime. The user can *end the lifetime* by giving up the full lifetime token, which produces a dead lifetime token.

The full lifetime logic supports significantly more operations to enable verifying realistic Rust code. However, in this section, we focus on describing how we model these operations. We model the remaining operations in a similar way.

21.1.2 Encoding Overview

The intuitive behaviour of the lifetime logic we described is formally captured using *rules* that manipulate Iris *resources*. The following equation shows the lifetime logic rule for creating a borrow[3].

$$\triangleright P \Rrightarrow \underbrace{\&^{\kappa}_{\text{full}} P}_{\text{borrow capability}} * \underbrace{([\dagger\kappa] \Rrightarrow \triangleright P)}_{\text{inheritance}} \tag{21.1}$$

3: Rule *LftL-borrow* on page 142 in [129]: Jung (2020), 'Understanding and evolving the Rust programming language'.

In this rule, P[4] is a resource that models the capability that is stored in the bank, $\&^{\kappa}_{\text{full}} P$ is a resource that models the mutable borrow capability, and $[\dagger\kappa] \Rrightarrow \triangleright P$ is a resource that models the inheritance.

4: $\triangleright$ is a later modality related to step indexing, which Iris requires for dealing with recursive types. We do not model the later modality because our approach does not support types for which it is necessary to use step indexing in RustBelt (for example, `&dyn Trait`). While we are confident that not modelling step indexing is justified in our case, it would be important future work to formally prove that this is indeed the case.

Our general approach is to model Iris resources using (typically abstract) predicates in Viper. For example, if the borrowed capability is an initialisation capability to place `a` of type `Pair`, which we model in Viper using predicate instance `Own<T>(a)`, we model the mutable borrow capability for lifetime `lft` as an instance of predicate `MutBorrow<Pair>(lft, a)`. Similarly to `Own<Pair>`, "`<Pair>`" in `MutBorrow<Pair>` is part of the

name of the predicate: since Viper is first-order, we model higher-order resources by monomorphising them. We model the inheritance produced by borrowing `a` using an abstract Viper predicate `Inheritance<Pair>(lft, a)`. The inheritance in Iris is defined as a *view shift*[5]. Intuitively, a view shift $A \Rrightarrow\!\!\!\!\ast\ B$ can be understood as a resource that similarly to a magic wand can be *applied* to transform resource A into resource B. We model applying a view shift using an abstract Viper method that consumes the view shift resource together with the resources on its left-hand side and produces the resources on its right-hand side. The following snippet shows the method we use to model the application of the inheritance for `Pair`.

5: In Iris both $\Rrightarrow\!\!\!\!\ast$ and $\Rrightarrow$ connectives are called view shifts as explained on page 138 in [129]: Jung (2020), 'Understanding and evolving the Rust programming language'. However, the rules we model contain only $\Rrightarrow\!\!\!\!\ast$.

```
method apply_inheritance<Pair>(
    lft: Lifetime,
    addr: Address
)
    requires acc(Inheritance<Pair>(lft, addr), write)
    requires acc(DeadLifetimeToken(lft), write)
    ensures acc(DeadLifetimeToken(lft), write)
    ensures acc(Own<Pair>(addr), write)
```

The fact that method `apply_inheritance` consumes the predicate instance modelling the inheritance ensures that the inheritance is applied only once. This snippet also shows that we model the dead lifetime token using an abstract Viper predicate `DeadLifetimeToken(...)`. Unlike the inheritance, the dead lifetime token is a duplicable resource. Therefore, we encode `apply_inheritance` to return the consumed dead lifetime token[6]. We model lifetimes using uninterpreted Viper type `Lifetime`.

6: Alternatively, we could use the so-called *wildcard* permission amount for the dead lifetime token. When consuming a wildcard permission amount of a resource, Viper consumes a positive amount that is strictly smaller than currently available. As a result, if a wildcard permission amount is used for a resource, we know that we always have some positive amount to it, which is suitable for modelling duplicable resources. However, wildcard permissions are more complex and Viper is sometimes slower handling them than the constant ones. Therefore, we decided to model duplicable permissions by using constant permission amounts.

The borrow creation rule shown in Equation 21.1 is itself defined as a view shift. Therefore, we model applying the rule as a Viper method. However, unlike the view shift used in the definition of the inheritance, this view shift can be always proven without consuming any resources. Therefore, in our Viper encoding we omit the corresponding predicate; the method that encodes the applying of the rule requires only resource P in its precondition. In general, we model lifetime logic rules as Viper methods that check additional requirements and require the consumed resources in their preconditions and ensure the produced resources in their postconditions. If a lifetime logic rule is bidirectional, we model it as two Viper methods; one for each direction.

The last concept we need to explain is how we model references. In Rust, a reference is a pointer to the borrowed memory location. Therefore, we model references as pointers whose initialisation capabilities include the borrow capabilities. More specifically, we encode this initialisation capability of a mutable reference to `Pair` using the Viper predicate shown in the following snippet.

```
predicate Own<&mut Pair>(addr: Address, lft: Lifetime) {
    acc(Own<*mut T>(addr), write) &&
    acc(MutBorrow<T>(
        lft, value<*mut T>(snap<*mut T(addr))
    ), write)
}
```

Predicate `Own<*mut Pair>(addr)` provides a capability to the raw pointer

while `MutBorrow<T>(...)` is the mutable borrow capability of the target. We make the lifetime of the reference an explicit argument of the `Own` predicate representing the reference. We model a shared reference to `T` in the same way as a mutable reference with the only difference that we use predicate `ShrBorrow` instead of `MutBorrow`. Predicate `ShrBorrow` expresses shared borrow capability of the target.

In the next subsection, we show how we generate the core proof for a simple borrowing example.

21.1.3 Example Core Proof Walkthrough

Figure 21.1 shows a simple borrowing example. In this example, variable `a` is mutably borrowed by reference `x` for lifetime `'l0` (we use illegal syntax to make the example clearer; Rust does not allow specifying lifetimes in borrow expressions). Then, reference `x` is used to mutate field `first`. Since this is the only use of `x`, the reference expires after the assignment and `a` becomes unblocked. Figure 21.2 shows a slightly simplified version of the Viper encoding of this example, automatically generated by our approach. In Figure 21.2, before each statement we show the resources consumed by that statement (if any) and after the statement we show the resources produced by the statement (if any).

```
1  let mut a = Pair { first: 1, second: 2 };
2  let x = &'l0 mut a;
3  x.first = 42;
```

Figure 21.1: A simple borrow.

Encoding of `let mut a = ...`. The first statement in Figure 21.1 initialises variable `a`. We encode it as shown on line 4 in Figure 21.2 by calling the helper Viper method `assign<Pair>(a)`. As can be seen on line 5, this statement produces an initialisation capability to `a`, which we model with predicate instance `acc(Own<Pair>(a),write)`.

Encoding of `let x = &'l0 mut a`. The second statement in Figure 21.1 borrows variable `a` for lifetime `'l0` with reference `x`. Borrowing in RustBelt does not automatically create the lifetime; the user needs to create the lifetime using a ghost statement `newlft`. We automatically insert this statement before creating the borrow to create the lifetime `'l0` as shown in Figure 21.2 on line 6. This statement returns a fresh lifetime and a full lifetime token that signifies that the lifetime is alive. We model the `newlft` statement in Viper as a helper method that produces a lifetime token as shown on line 7. Similarly to the dead lifetime token, we model the lifetime token as an abstract Viper predicate `LifetimeToken(lifetime)`.

With lifetime created, we can execute the borrow statement itself. Borrowing variable `a` consumes the initialisation capability for `a` and produces an initialisation capability for reference `x` and an inheritance to recover `a` (shown on line 10). The unique borrow capability of the target is hidden inside the initialisation capability of the reference as we can see after the reference predicate is unfolded on line 12. We model the creation of a borrow using an abstract Viper method `borrow_mut` as shown on line 9.

```
1  var l0: Lifetime
2  var a: Address
3  var x: Address
4  assign<Pair>(a);
5  Produce: { acc(Own<Pair>(a),write) }
6  l0 := newlft();
7  Produce: { acc(LifetimeToken(l0),write) }
8  Consume: { acc(Own<Pair>(a),write) }
9  borrow_mut(x, l0, a);
10 Produce: { acc(Own<&mut Pair>(x, l0),write), acc(Inheritance<Pair>(l0, a), write) }
11 Consume: { acc(Own<&mut Pair>(x, l0),write) }
12 unfold acc(Own<&mut Pair>(x, l0),write)
13 Produce: { acc(Own<*mut Pair>(x),write),
              acc(MutBorrow<Pair>(l0, value<*mut Pair>(snap<*mut Pair>(x))), write) }
14 Consume: { acc(MutBorrow<Pair>(l0, value<*mut Pair>(snap<*mut Pair>(x))), write) }
15 unfold_mut_ref<Pair>(l0, value<*mut Pair>(snap<*mut Pair>(x)))
16 Produce: { acc(MutBorrow<i32>(l0, ..._first(...(...(x)))), write),
              acc(MutBorrow<i32>(l0, ..._second(...(...(x)))), write) }
17 Consume: { acc(MutBorrow<i32>(l0, ..._first(...(...(x)))), write),
              acc(LifetimeToken(l0),token_perm) }
18 token_perm = open_mut_ref<i32>(l0, field_address_Pair_first(value<...>(snap<...>(x))))
19 Produce: { acc(Own<i32>(..._first(...(...(x)))), write),
              acc(MutBorrowClose(l0, ..._Pair_first(...(...(x))), token_perm), write) }
20 Consume: { acc(Own<i32>(..._first(...(...(x)))), write) }
21 assign<i32>(field_address_Pair_first(value<...>(snap<...>(x))), ...42...)
22 Produce: { acc(Own<i32>(..._first(...(...(x)))), write) }
23 Consume: { acc(Own<i32>(..._first(...(...(x)))), write),
              acc(MutBorrowClose(l0, ..._first(...(...(x))), token_perm), write) }
24 close_mut_ref<i32>(l0, field_address_Pair_first(value<...>(snap<...>(x))), token_perm)
25 Produce: { acc(MutBorrow<i32>(l0, ..._first(...(...(x)))), write),
              acc(LifetimeToken(l0),token_perm) }
26 Consume: { acc(LifetimeToken(l0),write) }
27 endlft(l0);
28 Produce: { acc(DeadLifetimeToken(l0),write) }
29 Consume: { acc(Inheritance<Pair>(l0, a), write), acc(DeadLifetimeToken(l0),write) }
30 apply_inheritance(l0, a);
31 Produce: { acc(Own<Pair>(a),write), acc(DeadLifetimeToken(l0),write) }
```

Figure 21.2: A slightly simplified version of the encoding of simple borrow from Figure 21.1 using RustBelt lifetime logic, generated automatically by our approach. Before each statement we show resources consumed by the statement (if any) and after each statement we show the resources produces by the statement (if any).

Encoding of `x.first = 42`. The last statement in Figure 21.1 assigns to a field of the borrowed pair via reference `x`. To justify this assignment, we need to obtain the initialisation capability to place `x.*.first`. On a high-level, this requires unpacking capability `x` twice to obtain the borrow capability for `x.*.first` and then open the borrow to obtain the initialisation capability for `x.*.first`. Since we have a regular initialisation capability for `x`, we can unpack it using a Viper `unfold` statement (line 12) as we presented in Chapter 18. Unpacking `x` consumes the initialisation capability for `x` (shown on line 11) and produces the initialisation capability for `x.pointer` and a mutable borrow capability (shown on line 13). To obtain the borrow capability for `x.*.first`, we need to unpack the borrow capability for `x.*`. Unpacking mutable borrow capabilities for

complex types requires care as we explain in Subsection 21.1.4. However, mutable borrow capabilities to simple structs like `Pair` can be unpacked in the same way as initialisation capabilities. We encode the unpacking of the capability using a helper method `unfold_mut_ref<Pair>(...)` (line 15), which for `Pair` has the same effect as a Viper `unfold` statement: it consumes the capability for the pair (shown on line 14) and produces the capabilities for its fields (shown on line 16). Finally, we have the mutable borrow capability `x.*`, which we need for opening the borrow to obtain the initialisation capability for `x.*`.

As we mentioned earlier, the RustBelt rule for opening the mutable borrow [7] enables exchanging a mutable borrow capability and a fraction of a lifetime token for the corresponding initialisation capability. We model this rule using method `open_mut_ref<i32>` as shown on line 18. As shown on line 17, this method consumes the mutable borrow capability for `x.*.first` and some positive permission amount of the lifetime token. Since the permission amount this method consumes is under-specified (it is known to be positive and smaller than the currently held permission amount), it returns the actually consumed permission amount, which we store in the variable `token_perm`. The method gives to the caller the initialisation capability to `x.*.first`, which we model with predicate `Own<i32>`, and a view shift for closing the borrow, which we model using an abstract Viper predicate `MutBorrowClose` as shown on line 19. After the assignment, which we model with `assign<i32>` helper method, we apply the view shift by calling `close_mut_ref<i32>` method as shown on line 24. This method consumes the initialisation capability to `x.*.first` together with the resource for the view shift itself and restores the capability state to the one we had before opening the borrow.

7: Rule *LftL-bor-acc* on page 142 in [129]: Jung (2020), 'Understanding and evolving the Rust programming language'.

Encoding of Expiration. Since the assignment of `x.first` is the last use of reference `x`, we can end lifetime `'l0`. In RustBelt, the lifetime is ended by executing statement `endlft` that consumes the full permission to the lifetime token and produces the dead lifetime token. We model this statement using an abstract Viper method as shown on line 27. Once we obtained the dead lifetime, we can use it to apply the inheritance view shift to recover that borrowed initialisation capability `a`. We model applying the inheritance view shift by calling method `apply_inheritance` as shown on line 30. This method consumes the inheritance resource together with the dead lifetime token and produces the borrowed capability. The dead lifetime token is a duplicable resource. Since Viper does not support duplicable resources, `apply_inheritance` returns the predicate instance modelling the dead lifetime token back to the caller.

In this example, we showed how we model the most important parts of the lifetime logic. The full lifetime logic has significantly more rules, which we model in a similar way using the general principles we presented in Subsection 21.1.2 and, therefore, we do not describe them here. Table 21.1 summarises the predicates we use to model RustBelt resources.

21.1.4 Modelling Borrow Capabilities

We mentioned that we model mutable borrow capabilities using Viper predicates `MutBorrow(...)` and shared borrow capabilities using Viper

Table 21.1: Viper predicates used for modelling lifetime logic resources.

Predicate	Explanation
`LifetimeToken(lifetime)`	Any positive permission amount of a lifetime token indicates that the corresponding `lifetime` is alive.
`DeadLifetimeToken(lifetime)`	Dead lifetime token indicates that the corresponding lifetime is dead. This predicate is duplicable.
`MutBorrow<T>(lifetime, address)`	A predicate representing a mutable borrow capability.
`MutBorrowClose<T>(lifetime, address, tperm)`	A predicate modelling the viewshift to close a mutable borrow.
`ShrBorrow<T>(lifetime, address)`	A predicate representing a shared borrow capability.
`ShrBorrowClose<T>(lifetime, address, tperm)`	A predicate modelling the viewshift to close a shared borrow.
`Inheritance<T>(lifetime, address)`	A predicate modelling the viewshift to recover the mutably borrowed capability.

predicates `ShrBorrow(...)`. However, we have not described how we model them. We do that in this subsection.

Similarly to `Own` predicates, the borrow predicates `MutBorrow` and `ShrBorrow` express that the target is initialised because Rust references can point only to valid instances of types. However, differently from owned values, borrow values cannot be moved or reallocated. Therefore, while the lifetime logic allows borrowing arbitrary predicates, we decided to not model borrowed variants of allocation predicates such as `MemoryBlock` and `MemoryBlockDropHeap`. Based on this decision, we define `MutBorrow` and `ShrBorrow` for primitive types as abstract Viper predicates. We define `MutBorrow<T>(...)` for composite type `T` similarly to `Own<T>(...)` with all nested `Own` predicates replaced with `MutBorrow` predicates and all allocation predicates removed. We define `ShrBorrow<T>(...)` for composite types in the same way. In the rest of this subsection, we discuss how we model unpacking and packing borrow capabilities.

Shared Borrow Capabilities. Shared borrow capabilities in RustBelt are duplicable resources. This means that if we unpack the shared borrow capability for place `p`, we can still use it. Since Viper does not support duplicable resources, we model unpacking and packing of a shared borrow capability as preserving the predicate instances in a similar way how applying inheritance preserves the dead lifetime token. The following snippet shows the definition of the abstract method that models unpacking of a shared borrow capability for `Pair`.

```
1 method unfold_ref<Pair>(lifetime: Lifetime, address: Address)
2     requires acc(ShrBorrow<Pair>(lifetime, address), write)
3     ensures acc(ShrBorrow<Pair>(lifetime, address), write)
4     ensures acc(ShrBorrow<i32>(lifetime, field_address_Pair_first(address)), write)
5     ensures acc(ShrBorrow<i32>(lifetime, field_address_Pair_second(address)), write)
6     ensures /* Link values of the pair with the values of its fields. */
```

As shown on line 3, we model that the shared borrow capability is duplicable by returning the original predicate instance to the caller. We encode packing of a shared borrow capability in a similar way.

Mutable Borrow Capabilities. The key property that a model of mutable borrows must guarantee is that when a borrow expires, the borrowed

value is in a valid state, which for types with core invariants means that the core invariant must hold. In models that use explicit backward capability flow, like the one we presented in Part I, we can ensure this property by performing the necessary checks during the backward flow. RustBelt's lifetime logic, similarly to the Rust compiler, uses implicit backward capability flow, which does not allow performing the explicit checks during the flow itself. Therefore, the lifetime logic guarantees that the invariant of the borrowed value holds when the borrow expires by ensuring that it holds at any program point at which the backward capability flow can be triggered. The backward capability flow is triggered when the corresponding lifetime is ended, which requires the full lifetime token. Since opening a borrow requires a fraction of a lifetime token, the backward capability flow can be triggered only when the borrow is closed. As a result, the lifetime logic must guarantee that when the borrow is closed, the borrowed value is in a valid state. The lifetime logic maintains this property by enforcing two rules. First, it checks that the value is valid when it is returned to the bank by closing the borrow. Second, it ensures that the invariant can be split only into independent parts[8], which in implicit dynamic frames means that each part needs to be self-framing. For example, this rule does not allow splitting off permission to `self.len` in the Viper assertion shown in the following snippet because that would make the expression that does the comparison not self-framing.

8: Rule *LftL-bor-split* on page 142 in [129]: Jung (2020), 'Understanding and evolving the Rust programming language'.

```
1 acc(self.cap, write) && acc(self.len, write) &&
2     self.len <= self.cap
```

The reason why the lifetime logic disallows splitting off permission to `self.len` is that mutating `self.len` could break assertion `self.len <= self.cap`. Requiring the assertion to be self-framing ensures that the assertion cannot be broken without observing that it was broken.

Figure 21.3 demonstrates what could go wrong if the lifetime logic had a more flexible rule that did not require to guarantee that the parts are independent. The example shows the implementation of method `set_len` on `ArrayList` that just sets the value of field `len` to the user provided value. This method implementation should not verify because at the end of the method, the core invariant of `*self` is potentially broken: for example, the core invariant of `ArrayList` requires that the value of field `len` is not larger than the value of field `cap` while the new value provided by the user does not guarantee this requirement. If `len` was set to a value larger than `cap`, the client could trigger a memory error by calling method `index` with an index larger than `cap`. If we used a more flexible version of the split rule that just splits the invariant into conjuncts, we could use it to obtain a mutable borrow capability to `self.*.len`. Then, we could use this capability to change the value of the field to an arbitrary value. Since the backward flow is implicit, the postcondition of `set_len` does not require to return back any borrow capabilities and we can just leak them. More specifically, there is nothing that would force us to reconstruct the invariant, which would enable us to observe that it was violated.

There is no operation in Viper that would enable us to directly encode lifetime logic's split operation, which we need to model unpacking of a mutable borrow capability. Therefore, we define our own two operations that enable us to soundly unpack mutable borrow capabilities,

```rust
impl<T> ArrayList<T> {
    fn set_len(&mut self, new_len: usize) {
        // unpack self
        // ???
        // open_mut_ref self.len
        self.len = new_len;
        // close_mut_ref self.len
    }
}
```

Figure 21.3: An example that demonstrates why it is necessary to ensure that borrowed capabilities are always in a valid state.

which we encode as Viper methods `unfold_mut_ref(...)` and `unfold_mut_ref_obligation(...)`. Method `unfold_mut_ref(...)` is based on the observation that the invariant can be broken only when a memory location on which it depends is mutated, like field `len` in our example. Therefore, when this method unpacks the mutable borrow capability, it removes the capabilities of memory locations whose values are mentioned in the invariant. For example, unpacking the mutable borrow capability of type `ArrayList` removes capabilities to fields `ptr`, `cap`, and `len` while leaving the capability to the `own_range!(...)` referenced by `ptr`. This option is suitable, for example, for verifying the `ArrayList::index_mut` method that takes a mutable borrow to an entire `ArrayList` and "narrows" it to a mutable borrow to a single element. It is important to note that for structs without invariants, this method behaves the same as Viper's `unfold` statement.

For packing mutable borrow capabilities, we provide method `fold_mut_ref(...)`, but only for structs without type invariants because only mutable borrow capabilities to these types could be successfully packed after unpacking them. Method `unfold_mut_ref_obligation(...)` is based on the observation that the invariant is allowed to be broken while the borrow is open because the backward capability flow cannot be triggered due to a missing fraction of a lifetime token. This method, similarly to opening a borrow, consumes a fraction of a lifetime token, which can be recovered only by packing the capability back using method `fold_mut_ref_obligation(...)`. Since the lender can regain the borrowed capability only by ending the lifetime, which requires the full permission to the lifetime token, we ensure that the mutable borrow is packed back (which checks that the invariant holds) or the lender permanently loses access to the borrowed value.

Enums raise similar challenges to structs with invariants because the variant of the enum depends on the discriminant field. Similarly to our solution for structs with invariants, when unpacking a mutable borrow capability for enum, we remove the capability to the discriminant field. This encoding prevents reading the discriminant while having only the mutable borrow capability to the discriminant itself. We mitigate this potential incompleteness by defining reading of the discriminant to take capability to the entire enum. Nevertheless, our incomplete solution may lead to a verification error when verifying safe code that uses mutable references to enums; we leave addressing this issue to future work.

In this section, we showed how we model RustBelt's lifetime logic in Viper. In the following section, we show how we use the Polonius information

and our PCS elaboration algorithm for automatically deriving the core proof for safe Rust functions containing references.

21.2 Inferring Ghost Operations for Borrows

As we showed in the previous section, the encoding based on the lifetime logic consists of two parts: accounting of lifetimes and automation of operations on references. We start by showing in Subsection 21.2.1 how we use information available in Polonius to derive the lifetime accounting operations. Then, in Subsection 21.2.2, we show how we adapt the PCS elaboration algorithm to infer the ghost operations on references. For both parts, the presented approach can completely automatically handle safe code and we expose annotations that can be used by the user to guide the core proof generation for unsafe code. In this section, we present the key ideas of the approach. For details we refer the reader to Pascal Huber's master thesis [190] with whom we worked on implementing the approach.

[190]: Huber (2022), 'Automatically Generating Memory Safety Certificates for Rust Programs'

21.2.1 Inferring Lifetime Accounting Operations

One advantage of the library approach used by RustBelt is that the abstraction level provided by the library is close to the borrow checker, which makes it relatively straightforward to automatically generate the core proof for code that uses references. As we discussed in Chapter 3, a lifetime in Polonius is a set of loans where a loan corresponds to a program point where a borrow occurred. Figure 21.4 repeats the example from Chapter 3, which we used to illustrate how Polonius works. In this example, reference `x` conditionally borrows either variable `a` or `b` and then is used to mutate the borrowed one. `x` is defined on line 1 and has lifetime `'lx`. On the then-branch, `x` borrows `a`. Polonius marks that `a` is blocked for loan `L1` and that `x` keeps this loan alive by putting `L1` into lifetime `'lx`. Similarly, for the else-branch, Polonius creates loan `L2` and adds it to set `'lx`. After the if statement, Polonius unifies the states of the branches taking a union of `'lx` values resulting in `'lx` containing both `L1` and `L2`. RustBelt's lifetime logic does not distinguish between loans and lifetimes. However, we can express both of them using lifetime logic's lifetimes. To avoid confusion, in this section we will refer to lifetimes used in the context of Polonius as "Polonius lifetimes" and lifetimes used in the context of RustBelt lifetime logic as "RustBelt lifetimes".

```
1  let x: &'lx i32;
2  // 'lx = { }
3  if random_choice() {
4      x = &mut a; // L1
5      // 'lx = { L1 }
6  } else {
7      x = &mut b; // L2
8      // 'lx = { L2 }
9  }
10 // 'lx = { L1, L2 }
11 *x = 4;
```

Figure 21.4: A conditional borrowing example with lifetime information computed by Polonius.

For loans, the key property we want to capture is how long they are alive because this property controls when the borrowed place can be used again. In lifetime logic, whether a RustBelt lifetime is alive is tracked using lifetime tokens: having some positive permission amount to the token for some lifetime witnesses that the lifetime is alive. Figure 21.5 shows an encoding of lifetime accounting of Figure 21.4 in Viper. We model loans using fresh RustBelt lifetimes. When a new loan is created by a borrow statement, we create a new RustBelt lifetime using `newlft` ghost operation as show on line 2. When the loan expires, we end it using `endlft` ghost operation as shown on line 15. As we mentioned in the previous section, the ghost operation `newlft` produces a full permission to `LifetimeToken` that signifies that the lifetime is alive. Ending the lifetime with the ghost operation `endlft` requires full permission to this token and, therefore, if some part of the permission is missing, the loan cannot be ended.

```
1  if random_choice() {
2      l1 := newlft();
3      lx := lft_tok_sep_take_1(l1, 1/2);
4      borrow_mut(x, l1, a);
5
6      lft_tok_sep_return_1(l1, 1/2);
7      old_lx := lx;
8      lx := lft_tok_sep_take_2(l1, l2, 1/2);
9      bor_shorten(lx, old_lx, 1/2, x);
10 } else {
11     // ...
12 }
13 // ...
14 lft_tok_sep_return_2(l1, l2, q)
15 endlft(l1);
16 endlft(l2);
```

Figure 21.5: A simplified encoding of lifetime accounting of Figure 21.4 in Viper.

For Polonius lifetimes, the key property we want to capture is that they do not outlive the loans they contain. We achieve this property using lifetime logic's rule LftL-tok-inter that allows creating a RustBelt lifetime that is an intersction of other RustBelt lifetimes. Applying this rule takes a fraction of the token of each of the intersected RustBelt lifetimes and gives back the fraction of the token to the resulting RustBelt lifetime. The following snippet shows method `lft_tok_sep_take_3` that models this rule for intersecting three RustBelt lifetimes (the number in the method name indicates how many loans it intersects).

```
method lft_tok_sep_take_3(
    l1: Lifetime,
    l2: Lifetime,
    l3: Lifetime,
    q: Perm
) returns (lifetime: Lifetime)
    requires acc(LifetimeToken(l1), q)
    requires acc(LifetimeToken(l2), q)
    requires acc(LifetimeToken(l3), q)
    ensures acc(LifetimeToken(intersect({l1, l2, l3}), q)
```

This method consumes `q` permission amount to lifetime tokens of the

intersected lifetimes, and returns `q` permission amount to the lifetime token of RustBelt lifetime `intersect({l1, l2, l3})` that is an intersection of lifetimes `l1`, `l2`, and `l3`. Function `intersect` is an abstract function that maps a multiset of RustBelt lifetimes into a RustBelt lifetime. Rule LftL-tok-inter is two sided, which means that the fractions of the original tokens can be recovered by giving up the fraction of the token to the intersected RustBelt lifetime. The following snippet shows a method that undoes the effect of `lft_tok_sep_take_3`.

```
1  method lft_tok_sep_return_3(
2      l1: Lifetime,
3      l2: Lifetime,
4      l3: Lifetime,
5      q: Perm
6  ) returns (lifetime: Lifetime)
7      requires acc(LifetimeToken(intersect({l1, l2, l3}), q)
8      ensures acc(LifetimeToken(l1), q)
9      ensures acc(LifetimeToken(l2), q)
10     ensures acc(LifetimeToken(l3), q)
```

We define a Polonius lifetime as a RustBelt lifetime that is an intersection of RustBelt lifetimes representing loans. When `x` borrows `a`, lifetime `'lx` contains only loan `L1`, which we encode by calling `lft_tok_sep_take_1` on line 8. For the permission amount `q`, we use the value $\frac{1}{c+1}$ where c is the largest number of lifetimes containing a specific loan, which is guaranteed to be small enough to satisfy all `lft_tok_sep_take` operations. At the end of the if statement, we have to update `'lx` to contain both `L1` and `L2`. We achieve this by returning the fraction of the lifetime token by calling `lft_tok_sep_return_1` on line 6 and assigning the intersection of loans `L1` and `L2` to `lx` by calling `lft_tok_sep_return_1` on line 8. We can freely recreate lifetime tokens every time Polonius tells us that the set of loans included into the lifetime changed because lifetime tokens carry no value information that we would need to preserve. Since lifetime `lx` is used in the type of `x` and, as a result, the predicate that represents it, when the lifetime changes, the predicate for `x` has to be updated to use the new value of `lx`. We update the predicate by calling method `bor_shorten` on line 9 that models rule LftL-bor-shorten. When the reference is not used anymore and lifetime `'lx` dies, we call `lft_tok_sep_return_2` to return the fractions of the loan lifetime tokens. After this operation, we have full permission to lifetime tokens for RustBelt lifetimes `l1` and `l2` that model loans `L1` and `L2`. Therefore, we can end the two lifetimes using `endlft` statements.

The presented technique can be extended to support function calls in a straightforward way. Figure 21.6 shows function `caller` that takes two references `x` and `y`, reborrows `y` as `z`, and calls function `callee` with `z`. Reference `x` is declared to have lifetime `'lx` while reference `y` has lifetime `'ly`. Lifetime bound `'ly: 'lx` expresses that lifetime `'ly` outlives `'lx`. For simplicity, Polonius treats parameter lifetimes as loans. Therefore, reborrow lifetime `'lz` includes not only reborrow loan `L3`, but also lifetime `'ly`.

Figure 21.7 shows our encoding of lifetime accounting of Figure 21.6 in Viper. Since function `caller` is parametrised with two lifetimes `'lx` and `'ly`, its encoding takes permissions to lifetime tokens that correspond

```
fn caller<'lx, 'ly: 'lx>(
    x: &'lx mut i32,
    y: &'ly mut i32,
) {
    let z: &'lz = &*y;  // L3
    // 'lz = { L3, 'ly }
    callee(z);
}
```

Figure 21.6: A simple function call example with lifetime information computed by Polonius.

[25]: Heule et al. (2013), 'Abstract Read Permissions: Fractional Permissions without the Fractions'

to these lifetimes (lines 8 and 9). We use abstract read permissions [25] to enable calling this method with any positive permission amount: permission amount `call_perm` is a function parameter for which we only require it to be positive (line 7). The method returns the same permission amount to the tokens as it took in the precondition to ensure that the caller can end the lifetimes (lines 23 and 24). We encode the lifetime constraints using the `outlives(ly, lx)` function that expresses that lifetime `ly` outlives lifetime `lx` (line 6). We define this function for lifetimes as a multiset subset relation. If the encoded function has lifetime parameters, for parameter `q` in method `lft_tok_sep_take` we use `call_perm` as the base permission amount as shown on line 12, which prevents verification failing to due `lft_tok_sep_take` taking too much permission. We do not need to change the encoding of calls to `lft_tok_sep_take` when the arguments include not only loans, but also lifetimes because RustBelt does not distinguish between loans and lifetimes. As can be seen from the encoding of a function call to `callee` function on lines 16–18, for parameter `call_perm` we provide the same permission amount as for calls to `lft_tok_sep_take`.

```
method caller(
    x: Address, lx: Lifetime, y: Address, ly: Lifetime,
    call_perm: Perm
) {
    // ...
    inhale  outlives(ly, lx)
    inhale call_perm > none
    inhale  acc(LifetimeToken(lx), call_perm)
    inhale  acc(LifetimeToken(ly), call_perm)

    l3 := newlft();
    lz := lft_tok_sep_take_2(l1, ly, call_perm/2);
    reborrow_mut(z, lz, ...y...);

    // Function call to callee.
    exhale acc(LifetimeToken(lz), call_perm/2)
    // ...
    inhale acc(LifetimeToken(lz), call_perm/2)

    lft_tok_sep_return_2(l1, ly, call_perm/2);
    endlft(l3);

    ensures acc(LifetimeToken(lx), call_perm)
    ensures acc(LifetimeToken(ly), call_perm)
}
```

Figure 21.7: A simplified encoding of lifetime encoding of Figure 21.6 in Viper.

When discussing Polonius information for Figure 21.6, we omitted an

important detail. After the reborrow and before the call to `callee`, Polonius kills the loan `L3` making lifetime `'lz` **longer**. Removing loan `L3` and making lifetime `'l3` longer is sound in this case because `L3` is not borrowing any new memory (it is reborrowing what was already borrowed by lifetime `'lz`). RustBelt supports this scenario via rule F-equalize. However, this RustBelt rule is the one that the authors of `GhostCell` [131] had to weaken to be able to prove `GhostCell` sound in RustBelt. Therefore, we were interested whether we can generate the encoding without using this rule and decided to use heuristic that, in scenarios like this, avoids generating the unnecessary loan. The heuristic seems sufficient for realistic cases, but a proper evaluation is still needed.

[131]: Yanovski et al. (2021), 'GhostCell: separating permissions from data in Rust'

We have presented how we infer lifetime accounting operations for safe code based on information available in Polonius. We cannot automatically infer the lifetime accounting operations for unsafe code because raw pointers do not have lifetime information associated with them; therefore, user input is needed. Instead of exposing ghost operations that enable the user to do the lifetime accounting manually (like we did with operations that manipulate capabilities), we decided to expose higher-level operations that enable the user to change the input of Polonius by adding additional relations between lifetimes. This additional information enables Polonius to compute the lifetimes without the user manually writing the lifetime accounting operations. Currently, we provide two ghost operations for changing how the lifetime accounting part of the core proof is generated.

The first operation enables specifying lifetimes to references produced from raw pointers. The key challenge for lifetime accounting in code that uses raw pointers is that casting a raw pointer into a reference produces a reference with an unconstrained lifetime. For example, in the following snippet that reborrows `y` into `x` via cast to raw pointer `p`, the lifetime of reference `y` is completely unrelated to the lifetime of reference `x`.

```
let x = &mut a;
let p = x as *mut i32;
let y = unsafe { &mut *p };
```

However, to prove that `y` is a valid reference, we need to link it to the borrow capability of reference `x`. The easiest way to do that would be to specify that `x` and `y` have the same lifetime. However, Rust does not have suitable syntax for expressing such properties. Therefore, we introduce ghost operation `set_lifetime_for_raw_pointer_reference_casts!(r)` that tells Polonius for all casts from a raw pointer to a reference to use the lifetime of reference `r`[9]. This operation allows us to link lifetimes of references that are created via raw pointers as shown in the updated version in the following snippet.

9: Ideally, this operation should wrap a block of statements and apply only to casts within the block. However, due to technical reasons related to compiler APIs, we had to make this operation global for the entire function in which it is used.

```
let x = &mut a;
set_lifetime_for_raw_pointer_reference_casts!(x);
let p = x as *mut i32;
let y = unsafe { &mut *p };
```

The second operation enables extending a lifetime until some value is dropped and is needed for verifying code that uses drop handlers to preserve or fix invariants when a panic occurs. Figure 21.8 shows function `override_from_input` from Chapter 19, which we used to illustrate how

drop handlers can be used to attach ghost operations necessary to show that the invariant is preserved when a panic occurs. On a panic branch of function `read_into`, line 15 is the last use of reference `self` and, therefore, Polonius tries to immediately expire it (ghost operations like `pack!(...)` do not affect Polonius). However, expiring `self` fails because it needs to be packed (as shown on line 10), which is done only later when the drop handler of `g` is executed. Therefore, to enable verifying this function, the user needs to be able to extend the lifetime of `self` until `g` is dropped, which they can do by using ghost operation `attach_drop_lifetime!(...)` as shown on line 13. Operation `attach_drop_lifetime!(value, reference)` instructs Polonius that the lifetime of `reference` should extended until `value` is dropped.

```
1  pub fn override_from_input<T: Input>(
2      &mut self,
3      input: &mut T,
4  ) {
5      unsafe {
6          // ...
7          unpack!(*self);
8          let g = GhostDrop;
9          before_drop! { g =>
10             pack!(*self);
11             // ...
12         }
13         attach_drop_lifetime!(g, self);
14         self.len = 0;
15         let len = read_into(input, self.ptr, self.cap);
16         self.len = len;
17     }
18 }
```

Figure 21.8: Function `override_from_input` from Figure 19.1 with ghost operation `attach_drop_lifetime!(...)` necessary to ensure that the lifetime does not end too soon.

21.2.2 Inferring Operations of Borrow Capabilities

For managing borrow capabilities we use the same approach as with initialisation and allocation capabilities, showed in Chapter 18 and Chapter 20. We define ghost operations that enable the user to manually manage borrow capabilities and modify the PCS elaboration algorithm to infer the operations for safe Rust. However, since we currently aim for a minimal support for references, we do not expose yet any predicates for talking about borrowed capabilities pointed at by raw pointers in our specifications. We also do not allow directly borrowing a value referenced by a raw pointer as shown in the following snippet.

```
let p = alloc(...);
unsafe { *p = 42 };
let x = unsafe { &mut *p };
```

It is clear how we could verify such examples in our core proof (the encoding is almost the same as when borrowing values allocated on the stack). However, the main challenge is finding an ergonomic syntax for providing the necessary specification.

As we showed in the previous section, for managing borrow capabilities we need three kinds of ghost operations, which we discuss in the following paragraphs.

Unpacking and packing borrow capabilities. Similarly to initialisation capabilities, we need to unpack and pack borrow capabilities to ensure they are in the right shape to justify accesses. Figure 21.9 repeats the implementation of the `index` method on `ArrayList`. This method takes a shared borrow capability to the entire list and returns a shared borrow capability to one of its elements. To justify this action, we need to unpack the shared reference capability, which we can do using the `unpack_ref!` operation as shown on line 8. Operation `unpack_ref!(lifetime, place)` requires to provide the lifetime because the unpacked capability could be behind a raw pointer and, therefore, with unknown lifetime. Rust allows giving names only to lifetimes that are used as function parameters. Therefore, we provide a ghost operation `take_lifetime!(reference, name)` that allows giving a name to a lifetime used in the type of a reference. We use this operation on line 7 to give name `lft_self` to the lifetime used by reference `self`. We use this name to specify the lifetime in `unpack_ref!` operation. `unpack_ref!(...)` operation is mapped to operation `unfold_ref(...)` on the Viper level. In addition to `unpack_ref!(...)` we also expose `pack_ref!(...)` for packing the capabilities of shared borrows (mapped to `fold_ref(...)`), `unpack_mut_ref!(...)` and `unpack_mut_ref_obligation!(...)` for unpacking capabilities of mutable borrows (mapped to `unfold_mut_ref(...)` and `unfold_mut_ref(...)`, respectively), and `pack_mut_ref!(...)` and `pack_mut_ref_obligation!(...)` for packing capabilities of mutable borrows (mapped to `fold_mut_ref(...)` and `fold_mut_ref_obligation(...)`) operations.

```
1  #[pure]
2  #[requires(index < self.len())]
3  pub fn index(&self, index: usize) -> &T {
4      assert!(index < self.len());
5      let element_ptr = unsafe { self.ptr.add(index) };
6      set_lifetime_for_raw_pointer_reference_casts!(self);
7      take_lifetime!(self, lft_self);
8      unpack_ref!(lft_self, *self);
9      let result = unsafe { &*element_ptr };
10     result
11 }
```

Figure 21.9: Implementation of `ArrayList::index` with ghost operations necessary to verify its memory safety.

For safe code, we infer the unpacking and packing of borrow capabilities using the PCS elaboration algorithm. In the algorithm, we treat borrow and initialisation capabilities uniformly because we can decide which of the operations we need to generate purely based on the type of the place: if the unpacked place is behind a mutable reference, we emit `unpack_mut_ref!`, if behind a shared reference, we emit `unpack_ref!`, and otherwise we emit `unpack!`.

Opening and Closing Borrow Capabilities. As we mentioned above, the lifetime logic does not allow using borrow capabilities directly for reading or writing. Therefore, before reading or writing we have to always open the borrow capability. For example, in the following snippet

`a` is borrowed by reference `x`, which is then cast to raw pointer `*p` used for mutation.

```
let x = &mut a;
take_lifetime!(x, lft);
let p = x as *mut i32;
open_mut_ref!(lft, *x, witness);
let b = unsafe { *p };
close_mut_ref!(*x, witness);
```

To justify the read through the raw pointer `p`, we need to provide the initialisation capability for the target. We obtain this capability by opening the borrow capability using operation `open_mut_ref!(lifetime, place, identifier)`. The `identifier` is a name we use to link opening of the reference with closing it. This name has to be provided to `close_mut_ref!` operation when closing the borrow capability. For safe code, we use a simple heuristic for inferring open and close operations: we wrap the access operation in open and close pair.

Recovering Borrowed Capabilities. Since we do not support yet directly borrowing values referenced by pointers, borrows can be created only in safe code. Therefore, we do not need to expose any operation that would manually restore the borrowed capability by applying the inheritance. For inferring application of inheritance for safe code, we use an approach similar to the one used by λ_{Rust} type system. In the PCS elaboration algorithm, we mark which places are currently blocked and when they are accessed again, we unblock them by emitting the application of the inheritance.

21.3 Lifetime Logic for Functional Specifications

In the previous sections, we showed how we can incorporate RustBelt's lifetime logic into our core proof and automate it using Polonius information. However, RustBelt does not enable verifying functional correctness, which is enabled by RustHornBelt. In Chapter 7, we showed that the prophetic model used by RustHornBelt can be used to encode pledges. Therefore, it would be a natural choice for us to use this model for modelling references in mixed safe-unsafe code. As we mentioned in the introduction of this chapter, automating RustHornBelt in an SMT-based verifier is challenging because it tracks values of borrows outside of the lifetime logic using powerful and complex constructs[10]. Therefore, instead of attempting to automate RustHornBelt, we decided to imagine a variant of the lifetime logic that would encapsulate the complexity of justifying prophetic reasoning inside it. This approach enables us to significantly simplify the encoding making it much easier to automate in an SMT-based verifier. The presented approach is still work-in-progress: for example, we do not have yet a clear argument why the designed interface is sound.

10: For example, linked ghost state such as prophecy controller and value observer.

We use the prophetic approach only in the encoding. For writing specifications, we use the pledges we introduced in Part I. We encode pledges into

prophecies using the relationship between the two models we showed in Chapter 7. Even though we could, we currently do not expose any operator that would allow specifications or ghost code to refer to the final value of a reference. Our encoding of functional specifications relies on snapshots. Therefore, in Subsection 21.3.1, we explain how we define snapshots for references and, in Subsection 21.3.2, we show how we link predicates with snapshots. Finally, in Subsection 21.3.3, we present the ghost operations for resolving mutable references and explain how we infer them for safe code.

21.3.1 Reference Snapshots

In this subsection, we show how we define snapshots for shared and mutable references. As we mentioned above, we define a capability for a reference as a capability for the raw pointer and a capability for the referenced memory location. The following snippet shows `Own` predicate for a shared reference to `T`.

```
predicate Own<&T>(addr: Address, lft: Lifetime) {
    Own<*const T>(addr) &&
    ShrBorrow<T>(lft, addr)
}
```

Based on this definition of the predicate, we define the snapshot of a shared reference similarly to a struct that contains a raw pointer field and a field of type `T`. The following snippet shows the definition of a snapshot for `&T`.

```
adt Snap<&T> {
    Value(pointer: Snap<*const T>, deref: Snap<T>)
    Uninit
}
```

A snapshot is valid if both the pointer and the target value are valid. We define the `snap` function for `&T` as shown in the following snippet.

```
function snap<&T>(addr: Address, lft: Lifetime): Snap<&T>
    requires acc(Own<&T>(addr, lft), wildcard)
    ensures is_valid<&T>(result)
{
    unfolding acc(Own<&T>(addr, lft), wildcard) in
        Snap<&T>::Value(
            snap<*const T>(addr),
            snap_shr_ref<T>(
                lft,
                value<*const T>(snap<*const T>(addr))
            ),
        )
}
```

The definition is similar to `snap` functions for other types with the exception that to obtain a snapshot of shared borrow target we use function `snap_shr_ref`, which we will define below.

The main difference between snapshots of shared references and snapshots of mutable references is that the latter in the prophetic model

include the final value of the target. Therefore, the snapshot for `&mut T` we define as shown in the following snippet.

```
adt Snap<&mut T> {
    Value(
        pointer: Snap<*const T>,
        deref_current: Snap<T>,
        deref_final: Snap<T>
    )
    Uninit
}
```

Similarly, the `snap` function for mutable references obtains the final snapshot of the target as shown in the following snippet.

```
function snap<&mut T>(
    addr: Address, lft: Lifetime
): Snap<&T>
    requires acc(Own<&mut T>(addr, lft), wildcard)
    ensures is_valid<&mut T>(result)
{
    unfolding acc(Own<&mut T>(addr, lft), wildcard) in
        Snap<&mut T>::Value(
            snap<*mut T>(addr),
            snap_mut_ref_current<T>(
                lft, value<*mut T>(snap<*mut T>(address))
            ),
            snap_mut_ref_final<T>(
                lft, value<*mut T>(snap<*mut T>(address))
            ),
        )
}
```

In this snippet, function `snap_mut_ref_current` obtains the current snapshot of the target while function `snap_mut_ref_final` obtains the final snapshot of the target. We define both of these functions below.

21.3.2 Obtaining Snapshots

In the previous subsection, we mentioned two functions `snap_shr_ref` and `snap_mut_ref_current` for obtaining current snapshots of borrows and function `snap_mut_ref_final` for obtaining the final snapshot of the mutable borrow. We first show how we define the functions for obtaining the current snapshot and then discuss how we obtain the final snapshot.

Current Snapshot. A straightforward way to define functions `snap_shr_ref` and `snap_mut_ref_current` would be as taking both a borrow

predicate and a lifetime token as shown in the following snippet.

```
function snap_shr_ref<i32>(
    lft: Lifetime, addr: Address
): Snap<i32>
    requires acc(ShrBorrow<i32>(lft, addr), wildcard)
    requires acc(LifetimeToken(lft), wildcard)
    ensures is_valid<i32>(result)
```

However, if we defined these functions in this way, we would not be able to call them from the `snap` functions of references because these functions do not have access to the lifetime token. Therefore, we define `snap_shr_ref` and `snap_mut_ref_current` to take only a corresponding borrow predicate. As a result, these functions can be used to read values of already expired references. After the expiration of the lifetime, these functions are guaranteed to always return the last snapshot that the reference had before the expiration of the lifetime. For shared references this property holds trivially because they are immutable. For mutable references, this property holds because updating the reference requires to open it, which requires to provide both a fraction of the lifetime token and the full mutable borrow.

As shown in the snippet above, we define `snap_shr_ref` and `snap_mut_ref_current` for primitive types as abstract functions. For composite type `T`, we define `snap_shr_ref` by transforming function `snap<T>` by replacing calls to nested `snap` functions with `snap_shr_ref`. The following snippet shows the definition of `snap_shr_ref` for `Pair`.

```
function snap_shr_ref<Pair>(
    lft: Lifetime, addr: Address
): Snap<Pair>
    requires acc(ShrBorrow<Pair>(lft, addr), wildcard)
    ensures is_valid<Pair>(result)
{
    unfolding acc(ShrBorrow<Pair>(addr), write) in
        Snap<Pair>::Value(
            snap_shr_ref<i32>(...first...),
            snap_shr_ref<i32>(...second...),
        )
}
```

Similarly, we define `snap_mut_ref_current` by replacing nested `snap` calls with `snap_mut_ref_current`.

Final snapshot. A current value of a mutable borrow can change, which is the reason why `snap_mut_ref_current` has to depend on predicate `MutBorrow`. However, the final snapshot is fixed (prophetised) at the borrow creation time and after that does not change. Therefore, we model `snap_mut_ref_final` using a heap-independent function. Besides this difference, we define `snap_mut_ref_final` similarly to `snap_mut_ref_current`. For primitive types, `snap_mut_ref_final` is abstract and for composite types it is defined as `snap` with nested calls replaced with `snap_mut_ref_final`.

21.3.3 Resolving References

In the previous subsections we presented snapshots of references that enable expressing functional specifications and functions that link these snapshots with predicates. In this section, we show how we model the last ingredient of the prophetic approach: operation `resolve` that is used to resolve the prophecy when a reference expires. Intuitively, operation `resolve` tells the verifier that the reference will not be modified anymore and its current value is the final one. For example, the following snippet shows how place `a` is modified via mutable reference `x`.

```
let mut a = 1;
let x = &mut a;
take_lifetime!(x, lft);
*x = 42;
resolve!(lft, *x);
assert!(a == 42);
```

After the assignment to `x`, the reference is resolved using operation `resolve!(lft, *x)`, which enables proving that `a` value is 42. We model operation `resolve!(lifetime, place)` in Viper using an abstract Viper method shown in the following snippet.

```
method resolve<T>(lft: Lifetime, addr: Address, q: Perm)
    requires acc(LifetimeToken(lft), q)
    requires acc(MutBorrow<T>(lft, addr), write)
    ensures old(snap_mut_ref_current<T>(lft, addr)) ==
        snap_mut_ref_final<T>(lft, addr)
    ensures acc(LifetimeToken(lft), q)
```

Similarly to RustHornBelt's rule F-resolve, this method consumes the predicate instance `MutBorrow<T>(...)` disallowing any further mutations. In addition to the `resolve!(...)` operation for resolving a single capability, we also provide `resolve_range!(...)` and `resolve_set!(...)` variants for resolving ranges and sets of capabilities. For example, Figure 21.10 shows the implementation of `index_mut` method of our `ArrayList` in which on lines 10 and 12 we use `resolve_range!(...)` to resolve all the elements of the list, except the one that is returned. In Figure 21.10, we do not explicitly resolve the fields of `ArrayList`. As we discussed earlier, `unpack_mut_ref!(...)` removes permissions for the predicate instances for fields to ensure that each predicate instance is self-framing. Actually, instead of simply removing them, we resolve them so that the verifier knows that values of these fields.

The presented operation `resolve!(...)` is sufficient for manually verifying complex borrow patterns. However, the Rust compiler to accept more code automatically inserts many reborrows that can be challenging to resolve automatically. For example, Figure 21.11 shows function `reborrow` that takes mutable reference `x`, reborrows it as `y`, and returns to the caller. While a programmer is unlikely to write such a function manually, this pattern is generated by the compiler when a programmer writes an identity function that simply returns the parameter. To verify the functional postcondition of function `reborrow`, we need to resolve parameter `x`. However, we cannot use operation `resolve!(...)` for this purpose because its encoding requires predicate instance `MutBorrow(...)`, which was

```
pub fn index_mut(&mut self, index: usize) -> &mut T {
    assert!(index < self.len());
    set_lifetime_for_raw_pointer_reference_casts!(self);
    take_lifetime!(self, lft_self);
    let ptr = self.ptr();
    let len = self.len();
    let element_ptr = unsafe { ptr.add(index) as *mut _ };
    unpack_mut_ref!(lft_self, *self);
    let result = unsafe { &mut *element_ptr };
    resolve_range!(lft_self, ptr, 0, len, 0, index);
    let next_index = index + 1;
    resolve_range!(lft_self, ptr, 0, len, next_index, len);
    result
}
```

Figure 21.10: Implementation of `ArrayList::index_mut` with ghost operations necessary to verify its memory safety.

consumed by reborrowing into `y`. Therefore, we resolve `x` using operation `resolve_reborrowed!(...)` on line 5. This operation instead of consuming predicate `MutBorrow`, consumes the inheritance that allows recovering it. This way, this operation ensures that reference `x` cannot be used anymore after it is resolved.

```
#[after_expiry(*x == before_expiry(*result))]
pub fn reborrow(x: &mut i32) -> &mut i32 {
    take_lifetime!(x, lft);
    let y = &mut *x;
    resolve_reborrowed!(lft, *x);
    y
}
```

Figure 21.11: A simple reborrowing function demonstrating the need of `resolve_reborrowed!(...)` operation.

For inferring `resolve` operations, we use a similar approach to Creusot [134], which resolves the reference when it dies. The key difference between our approach and Creusot is that we use the information available in the PCS elaboration algorithm to decide whether we need to emit `resolve!(...)` or `resolve_reborrowed!(...)`. Creusot can treat both cases uniformly since it does not model capabilities.

[134]: Denis et al. (2022), 'Creusot: A Foundry for the Deductive Verification of Rust Programs'

21.4 Summary

In this chapter, we showed how we extend our approach for verifying mixed safe-unsafe code to Rust programs with references. The extension presented in this chapter maintains all the properties of the approach presented in this part of the thesis: it enables verifying memory safety and functional correctness of mixed safe-unsafe programs, it completely automatically generates the core proof for safe code based on the information available in the compiler, and it provides annotations that the user can use to guide the generation of the core proof for examples that contain unsafe code. An important property of our approach is that the visible behaviour of safe abstractions can be specified using the constructs we introduced in Part I. For example, Figure 21.12 shows the functional specification of method `ArrayList::index_mut`, which is identical to the one of the linked list shown in Figure 17.1 on page 155. As a result, in Figure 21.13, regardless if variable `list` is a safe type `LinkedList` or

internally unsafe type `ArrayList`, the required effort for the user trying to verify the client is exactly the same.

```
#[requires(0 <= index && index < self.len())]
#[ensures(result === old(self.index(index)))]
#[after_expiry(
    self.len() == old(self.len()) &&
    forall(|i: usize|
        0 <= i && i != index && i < self.len() ==>
        self.index(i) === old(self.index(i))
    ) &&
    self.index(index) === before_expiry(&*result)
)]
pub fn index_mut(&mut self, index: usize) -> &mut T {
    // ...
}
```

Figure 21.12: Functional specification of `ArrayList::index_mut`. The implementation with annotations required to verify this method are shown in Figure 21.10 on the preceding page.

```
// Assuming: list.index(0) == 1 && list.index(1) == 1
let r = list.index_mut(1);
*r = 42;
assert!(list.index(0) == 1 && list.index(1) == 42);
```

Figure 21.13: A client using a list. Since both `ArrayList` and `LinkedList` provide the same interface, the effort required for verifying the client code is the same for both data structures.

22 Implementation and Evaluation

In the introduction of this part of the thesis we named two goals we aim to achieve in the context of verifying existing Rust code. The first goal is to enable verifying functional correctness and memory safety of mixed safe-unsafe code that uses unsafe functions and raw pointers. In this goal, we aim to balance two conflicting desires: ease-of-use and expressive power. We aim to enable verifying complex unsafe code while at the same time keeping the approach lightweight for safe parts of the code. In particular, for completely safe code, we wanted to keep the benefits of the verification approach presented in Part I and shield the user from complex permission logics. The second goal is to enable verifying memory safety of safe abstractions used by unverified safe code. As we mentioned in the introduction, one of the main challenges is ensuring panic safety, which programmers typically ensure using error-prone techniques. In this goal, we aim to support ensuring that these techniques are used correctly.

Our original plan for evaluating how the approach presented in this part of the thesis achieves these two goals was to implement our approach as an extension to Prusti, which we presented in Part I, and perform a large case study. For example, collections from the standard library such as `Vec`, `VecDeque`, and `LinkedList` should fall within the fragment supported by our approach, exemplify all three techniques for ensuring panic safety, and had important memory safety bugs. However, in the course of evaluation we ran into a blocking problem: Viper, the verification framework used by Prusti, is extremely slow on examples that use range or set capabilities. In many cases, Viper's symbolic execution backend (the backend we use for evaluation) would not terminate within an hour on an encoding of a Rust function that has a few lines of code. In our investigation of the performance problem, we found that when iterated separating conjunctions are used (to which we encode range capabilities), Viper's symbolic execution backend switches to a more complete algorithm for managing resources. The runtime of this algorithm, at worst case, can be exponential in the number of resources present in a Viper method (we present our investigation in more detail in the appendix, Chapter A). When the Rust compiler lowers a Rust program into MIR, it introduces many temporary variables that become resources in the encoded Viper method. As a result, even for relatively small Rust functions, the generated Viper methods can potentially have many resources. The examples we planned to evaluate are, unfortunately, too large for verification to terminate in reasonable time without fixing the performance problem. Not using iterated separating conjunction is also not an option since this feature is crucial for modelling non-trivial examples with raw pointers.

We considered attempting to fix the performance problem but decided against it for three reasons. First, we have ideas (presented in Chapter A) how we could approach the cause of the performance problem we identified but trying them out takes a significant amount of time and we

do not know whether they work. Second, we are not sure whether the cause of the performance problem we identified is the only one. Third, the investigation is made significantly harder by a bug in Z3, the SMT solver used by Viper, that causes it to crash non-deterministically. To sum up, attempting to fix the performance problem requires a significant amount of time and we are not certain that we are going to succeed. Therefore, we instead implemented the mitigations based on our current understanding of the problem and evaluated our approach on examples on which our encoding to Viper terminates in reasonable amount of time (we chose less than one minute per Rust function).

The rest of this chapter is structured as follows. In Section 22.1, we present the status of our prototype implementation. In Section 22.2, we present how we used our prototype to evaluate our approach and our findings.

22.1 Presentation of Changes Made to the Prusti Tool

We implemented the approach described in this part of the thesis as a prototype extension of Prusti, the verifier we presented in Part I. We made two kinds of changes to Prusti: implemented our approach for reasoning about mixed safe-unsafe Rust code and implemented some mitigations for the performance problem. We give an overview of each kind of change below.

22.1.1 Verifying Mixed Safe-Unsafe Code

We modified Prusti to support our new verification approach without changing the high-level design of the tool. The updated Prusti is still implemented as a compiler plugin that extracts a CFG representation of each function with type and Polonius information and encodes that function in Viper. The verification errors are reported to the user using the Rust compiler error reporting mechanism and are at the level of the Rust program. Even though the core proof generated by our prototype is more complex, we maintain the key property we achieved in Part I: the core proof for safe Rust programs is generated completely automatically and the users interact with the tool at the level of Rust expressions, being completely shielded from the complexity of the permission reasoning.

Our prototype implements the verification approach presented in this part of the thesis and supports all presented specification constructs. Compared to the version of Prusti presented in Part I, the new implementation additionally supports drop handlers, unsafe functions, raw pointers, and pointer arithmetic; we have removed special support for the `Box` type since a similar data structure can be verified using our approach. To support drop handlers, we have to integrate information from two different versions of MIR; our current implementation handles most common cases. The Rust compiler developers expressed that they would be open to provide all information in a single version of MIR. Due to a bug in the implementation, creating a raw pointer either by casting from a reference or by using `addr_of!(...)` currently requires an initialisation capability even though it should not require any. Also, the

implementation currently does not duplicate shared capabilities that are aliased via raw pointers. The implementation performs the discussed well-formedness checks, but some of them are incomplete: the implementation checks that drop handlers with specifications are used only on private structs, but does not enforce yet that these structs are not used as type arguments. Our new model of references requires us to have lifetime information in functional specifications. Unfortunately, the Rust compiler does not provide a reliable way of obtaining lifetime information for trait methods. As a consequence, our prototype implementation does not support trait methods with reference parameters. For some specification constructs, the implementation supports a more restricted syntax than presented due to the technical limitations of the approach we use to parse and type-check specifications. For example, `addr` in predicates such as `own!(addr)` must be a place (a local variable, or a field access) and using a pure function call is not supported. Also, the implementation does not support some combinations of specification constructs yet; most notably, pure functions are not yet allowed in type invariants.

Our implementation encodes panic safety and functional correctness as two different Viper programs. Even though the verifier needs to repeat some work, this design presents three benefits. First, it scales better to larger functions than having a single large Viper program. Second, this split presents optimisation opportunities that could be explored in the future. For example, the two could be cached independently avoiding full verification if a user changed specification relevant only for one part. Third, since panic safety requires a more comprehensive memory safety proof, the memory safety proof could sometimes be omitted when verifying functional correctness. We verify all functions (including the pure ones) by encoding them as Viper methods because it is unclear how to model some of the ghost operations (for example, opening a shared reference) as Viper expressions. To avoid verifying each pure function twice (once as a Viper method and once as a Viper function), we encode Rust pure functions using Viper domain functions and corresponding axioms.

22.1.2 Mitigating the Performance Problem

Our main hypothesis based on the investigation we present in Chapter A is that our issues with the performance are caused by the fact that when iterated separating conjunctions are used, Viper's symbolic execution backend switches to a more complete and slower algorithm for accounting permissions. More specifically, we found that the runtime of the iterated separating conjunctions algorithm could be, at worst case, exponential in the number of predicate instances. In many cases, when attempting to verify Rust programs, the verification time does not simply increase exponentially with each added statement in a Rust function but, instead, at some point suddenly jumps above a reasonable verification time. We think that there could be two reasons for such behaviour. First, a small change on the source level can sometimes result in relatively large changes on the MIR level. For example, calling a function with four parameters can lead to creating five temporary variables on the MIR level (four for the parameters and one for the result), which could increase the runtime of an exponential algorithm by a factor of $2^5 = 32$. The second and more

likely reason is related to heuristics used by Viper's symbolic execution backend. On a conceptual level, one strategy used by this backend to improve performance is to use heuristics to split resources into groups whose permissions can be tracked separately. Potentially, even a small change in a program can lead to two groups being merged. If two groups of sizes n and m get merged, the runtime would change from $2^n + 2^m$ to 2^{n+m}, which could be a noticeable change.

Unfortunately, Viper does not expose a way to control these heuristics and implementing such a mechanism would require major changes to the infrastructure. Since we are not certain that the potential reasons we found are sufficient causes to explain the performance problem, we decided to try mitigating the problem on the Prusti side. The key idea of the mitigation is to do permission accounting of resources that do not belong to iterated separating conjunctions in Prusti. This way we can ensure that these resources do not affect the performance of the iterated separating conjunction algorithm. By default, we manage all resources that are not iterated separating conjunctions in Prusti and provide an annotation `quantified!(c)` that allows the user to specify that the resource corresponding to capability `c` should be managed by Viper. This approach puts additional burden on the user, which is against the goals we aim to achieve, but we found it necessary to expose as few resources to the iterated separating conjunction algorithm as possible. Our implementation of manual permission accounting in Prusti is highly incomplete; we focused on supporting scenarios that are important for the evaluation and that we can hope to support. For example, our support for loops is limited because examples containing loops are typically too large to hope for them to terminate in reasonable time. Since our permission accounting in Prusti supports only Viper predicates, we changed the encoding to avoid magic wands by replacing them with abstract predicates and appropriate assertions. These mitigations enabled us to verify some non-trivial examples, which we present in the following section, but did not solve the problem in general and further work is needed to understand it and potential ways of fixing it.

22.2 Evaluation of the Verification Approach

We evaluated whether our approach is suitable for verifying realistic mixed safe-unsafe Rust code by performing two studies. In these studies, we focus on expressivity of our approach since the performance problem prevents us from meaningfully evaluating verification performance and annotation overhead[1]. In the first study, we evaluate whether our technique is expressive enough to verify safe abstractions that can be used for writing verified safe code. In the manual evaluation of Rust examples we presented in Subsection 6.2.3, we used safe abstraction data structures and functions with trusted specifications. In this first study, we attempt to verify these trusted specifications. We discuss this study in Subsection 22.2.1. In the second study, we evaluate whether our technique is sufficiently expressive for verifying tricky patterns that programmers use to ensure memory safety. This study was mainly done by Olivia Furrer under our supervision as part of her bachelor thesis [185]. In this study, she attempted to verify fixes of bugs found by RUDRA [172].

1: The number of annotations is inflated by the need to write annotations that help to mitigate the performance problem and the need to split larger functions into smaller ones that often have very similar specifications.

[185]: Furrer (2023), 'Verifying Vulnerability Fixes in a Rust Verifier'

[172]: Bae et al. (2021), 'Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale'

Table 22.1: Safe abstractions used in examples that we used in Subsection 6.2.3 to evaluate the suitability of Prusti for verifying functional correctness of safe code. The first half contains examples from Table 6.1. The second half contains examples from Table 6.2. The first three safe abstractions (`Vec`, `Box`, and `String`) are data structures and the last one is function (`std::mem::replace`) that enables the client to swap a value referenced by a mutable reference with an owned one. The subcolumns of `Vec` show how the vector was used. "Generic" indicates that a generic version of the vector was used (`Vec<T>` with `T` as a generic parameter). "Mono." that a monomorphised version of the vector was used; for example, `Vec<i32>`. "Inv." indicates that the wrapper had also a type invariant associated with it; for example, that all values stored in the vector are between -2 and 2. "`Vec<Vec<T>>`" indicates that a two-dimensional vector was used. In all examples two dimensional vectors also contained an invariant that all inner vectors are of the same length.

Example	`Vec`				`Box`	`String`	`replace`
	Generic	Mono.	Inv.	`Vec<Vec<T>>`			
100 doors	✓						
Binary Search (generic)	✓						
Heapsort	✓						
Knight's tour		✓	✓	✓			
Knuth Shuffle	✓						
Langton's Ant			✓	✓			
Selection Sort (no-panic)	✓						
Ackermann Func.							
Binary Search (mono.)		✓					
Fibonacci Seq.							
Knapsack Problem/0-1		✓	✓	✓			
Linked List Stack					✓		✓
Selection Sort (functional)		✓					
Towers of Hanoi							
Borrow First	✓						
Message						✓	
Binary Search	✓						
Selection Sort	✓						
Linked List					✓		

We present this study in Subsection 22.2.2.

22.2.1 Verifying Trusted Safe Abstractions

Table 22.1 lists the examples we verified in Subsection 6.2.3 together with safe abstractions for which we had to give trusted specifications. The examples use three data structures `Vec`, `Box`, and `String`, and method `std::mem::replace`. Table 22.2 gives an overview of the safe abstractions we verified. We could not verify `Vec` because of the performance problem. Therefore, instead of `Vec`, we use our `ArrayList` whose implementation is heavily inspired by `Vec`, but is written in a way that verification terminates within reasonable time (less than one minute per function). The key principle we used when implementing `ArrayList` is to strucure the code in such a way that each function operates on as few range capabilities as possible, which result in many small functions. To check that the specifications of `ArrayList` are suitable, we tried to implement the trusted stubs we used in examples and verify their specifications. Instead of `Box`, we verified our own implementation `SimpleBox` because `Box` in the Rust standard library is partially a compiler built-in with its most important functions such as allocation and deallocation being Rust compiler primitives. We checked whether specifications of `SimpleBox` are sufficient by trying to reverify the examples "Linked List Stack" and "Linked List" with `SimpleBox` instead of `Box`. `String` in example "Message"

Table 22.2: The verified safe abstractions we used intead of the standard library abstractions to verify the wrappers used in the examples. Column "Abstraction" lists the abstraction from the standard library, column "Replacement" lists the verified safe abstraction we used, "LOC" is the number of lines of code in the verified abstraction, "Spec. LOC" is the number of lines of specifications and ghost code needed to verify the abstraction, and "Verification Time" is time it took to verify the specifications averaged over 3 runs. It is important to note that since the implementations were written by us in a style that mitigates the performance problem, the numbers should be taken with caution.

Abstraction	Replacement	LOC	Spec. LOC	Verification Time
`Vec`	`ArrayList`	227	367	14 min. 48 sec.
`Box`	`SimpleBox`	47	31	1 min. 23 sec.
`replace`	n/a	5	6	17 sec.

is used as an opaque type with only an equality method without any specification defined on it; therefore, we omit it from our study. We verified the original implementation of function `std::mem::replace` [192] with a single-line change that makes a cast from a reference to a raw pointer explicit, which makes the compiler generate MIR operations supported by our implementation. In the following, we give an overview of what we achieved and then discuss for each safe abstraction which uses of it we managed to verify and what additional challenges we had to solve.

[192]: Developers (2024), *Implementation of function std::mem::replace*

Overview. The key property of a safe abstraction is that it gives an illusion to a safe client that it is written in safe Rust. More specifically, it cannot cause memory errors and its behaviour can be specified using the simple constructs we presented in Part I. For all three safe abstractions, we verified that they are memory safe and that their public functional specifications written in the specification language from Part I are upheld. For example, even though the `ArrayList` and `SimpleBox` implementations handle zero-sized types differently, this difference is hidden behind the abstraction. Therefore, the client is not required to know whether the type with which they instantiated the collection is zero-sized if it is not relevant for them (as we discuss below, for clients written in the context of embedded software, precise tracking of sizes may be important). As can be seen from Table 22.2, the implementations of `ArrayList` and `SimpleBox` are non-trivial with the `ArrayList` implementation taking 227 lines and the implementation of `SimpleBox` taking 47 lines of code. The original implementation of function `std::mem::replace` contains only 5 lines of code, but requires complex reasoning about raw pointers. The required number of lines of specification and ghost code is around the number of lines of code: `ArrayList` required 367, `SimpleBox` required 31, and `std::mem::replace` required 6 lines of specification. However, it is important to note that since `ArrayList` and `SimpleBox` were written by us in a style that mitigates the performance problem, the verification overhead could be lower significantly if the performance problems were solved. We measured how long it takes to verify the safe abstractions on Lenovo T470p laptop with Intel(R) Core(TM) i7-7700HQ CPU with 4 cores (8 threads), 16 GB of RAM, and Ubuntu 20.04.6 averaged over 3 runs. Verification of `ArrayList` takes almost 15 minutes, `SimpleBox` takes 1 minute 23 seconds, and `std::mem::replace` takes 17 seconds to verify.

`ArrayList`. We managed to implement and verify our own versions of all methods[2] used by `Vec` wrappers that use generic and monomor-

2: The `Vec` methods we reimplemented for `ArrayList` and verified are `new`, `with_capacity`, `from_elem`, `drop`, `len`, `capacity`, `push`, `index`, `index_mut`, and `swap`. All of these metods are part of the public `Vec` API, except `from_elem` that is used internally by macro `vec![...]`.

phised versions of the data structure. When specifying the behaviour of methods that allocate memory such as `with_capacity(n)` that creates a new list with capacity `n`, we had to solve a challenge that allocating can non-deterministically fail and cause a panic. Since the functional precondition is required to imply absence of panics, the only sound precondition we could give to an allocating method is `false`. Verifiers such as Dafny that target verification of application software for servers and personal computers, where running out of memory is an exceptional event, just assume that allocation never fails. However, Rust is also used for embedded software where it is critical to ensure that allocation errors are handled correctly. For example, a version of `Vec` for the Linux kernel [193] provides a method `try_with_capacity` that would return an error instead of panicking if an allocation error occurs. To support both of these scenarios, when verifying `ArrayList`, we introduced a constant `allocation_never_fails`, which we used to specify that allocation functions can fail only if this constant is false. Using this constant, we could give a meaningful functional specification to the `with_capacity` method: instead of precondition `false`, we used precondition `allocation_never_fails`. A user who does not care about allocation failures, could set this constant to `true` and use method `with_capacity`. The user for whom allocation failures are important would set this constant to `false` and would be prevented from accidentally using methods that panic when allocation fails. When verifying the wrappers, we set the constant to `true` since the safe examples were not designed to handle allocation failures. In addition to examples that use monomorphised and generic versions of `Vec`, we also considered examples that use `Vec` with an additional type invariant and examples that use two-dimensional vectors `Vec<Vec<T>>`. Currently, our implementation supports only core invariants on types and, therefore, we could not verify the wrappers of these examples. However, enabling them seems to be a straightforward extension of our approach. Overall, the approach itself seems to be suitable for verifying specifications of safe abstractions such as `ArrayList` that enable using them for building safe clients.

[193]: Linux Project Developers (2023), *Rust for Linux Vec::try_with_capacity*

`SimpleBox`. We implemented and verified the key operations such as creating a new box, destroying it, borrowing the value stored in it, and moving the value out of it. Since creating a new box potentially allocates new memory, which can non-deterministically fail, we used `allocation_never_fails` as its precondition. Implementing and verifying all methods was straightforward with the exception of moving out the value from the box, whose signature is shown in the snippet below.

```
fn into(self) -> T { /* ... */ }
```

This method consumes the box and returns the value stored in it. The challenge with implementing this method is that at the end of this method, the drop handler of the box is automatically called, which deallocates the box and the value stored in it, which is a problem because we want to return the value to the caller. Therefore, we need to cancel the invocation of the drop handler to be able to return a valid value. Rust provides two ways of cancelling the drop handler: function `forget` [181] and struct `ManuallyDrop` [194]. Function `forget` consumes the value without calling its destructor while `ManuallyDrop` allows wrapping the value to indicate to the compiler that the drop handler should not be called. The wrapped

[181]: Developers (2023), *Function std::mem::forget*

[194]: Developers (2023), *Struct core::mem::ManuallyDrop*

value can be accessed using methods that return references pointing to it. `ManuallyDrop` is a newer and preferred way of cancelling the drop handler. Unfortunately, our model of references is too restrictive to model this struct: our model requires that when a mutable borrow expires, the referenced value is in a valid state. For our `SimpleBox`, this means that it still has the capability to the value contained in it. However, the whole point of using `ManuallyDrop` is to be able to move out capabilities from it. When using `forget`, we run into a similar problem that this function consumes the value, which requires the value to be valid. To be able to move the value out of a box, we decided to special case the treatment of function `forget` in a similar way to how we treat the drop handler: when calling `forget`, instead of requiring the initialisation capability to the entire value, we require initialisation capabilities to its fields. This way we are able to keep the capability to the value stored in the box and return it to the caller.

We have successfully used `SimpleBox` to reverify the example "Linked List Stack". However, we could not verify example "Linked List" because the example uses patterns not yet supported by our mitigations for the performance problem.

`std::mem::replace`. As mentioned above, we successfully verified function `replace` with a small change to its implementation. Function `replace`, similarly to `std::mem::swap` discussed in Section 19.3, relies on called functions not panicking for its memory safety. Since the comment in the function claims that none of the called functions could panic, we annotated them with `#[no_panic]` annotations and were able to successfully verify the function. However, it is important to note that the Rust documentation of these functions does not mention that these functions guarantee absence of panics. In general, it seems that in the Rust documentation it is uncommon to mention the behaviour of a function with respect to panics. Since knowing this behaviour is crucial when writing unsafe code, the Rust documentation should be improved to include this information. We have successfully used function `replace` with our verified specifications when reverifying example "Linked List Stack".

22.2.2 Verifying Vulnerability Fixes

In this subsection, we summarise the study done by Olivia Furrer in her bachelor thesis and its key findings. For details, we refer the interested reader to her thesis report [185]. The goal of the study was to identify missing features that are required for verifying real-world unsafe Rust code. During her bachelor thesis project, Olivia tried to verify using our approach fixes of bugs found by RUDRA [172], a static analysis tool capable of scanning all publicly available Rust code for certain problematic patterns. She attempted to verify four complex examples, each representing a different pattern. She successfully verified examples `glium::buffer::Content::read` and `bam::bgzip::Block::load` in Prusti, example `std::vec::Vec::from_iter` verified in Prusti only partially, and for example `std::vec::Vec::dedup_by` verified a manual encoding of our approach in Viper. In the rest of this subsection, we discuss each example and the findings related to it.

[185]: Furrer (2023), 'Verifying Vulnerability Fixes in a Rust Verifier'

[172]: Bae et al. (2021), 'Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale'

`glium::buffer::Content::read`. In this example, `Vec` is initialised with data using a user-provided closure. The key pattern is shown in the following snippet.

```
let value = Vec::with_capacity(size);
unsafe { value.set_len(size); }
let slice = &mut *value;
f(slice);
```

A vector of capacity `size` is created and its length is set to match its capacity. Then, a mutable reference `slice` to its contents is created and passed to the user-provided closure `f` to initialise. Since creating a reference to uninitialised data is unsound according to the operational semantics presented in Stacked Borrows and Tree Borrows, Olivia changed the closure to take a mutable reference to the vector instead (actually, she changed the function that models the closure since our implementation does not support closures). The key challenge in verifying this example is specifying the call to closure `f` that takes a mutable reference to a vector with a weaker invariant: the invariant of `Vec` requires that the first `len` fields are initialised while in this example they are not. It is potentially surprising, but calling functions in a state that satisfies weaker requirements than implied by the type system is allowed as long as the called function explicitly allows that. Ralf Jung named such functions that can be called with weaker requirements than implied the type invariants of their parameters super-safe [195]. Olivia verified this example by introducing a new struct `BrokenVec` that tracks initialisation using a ghost field instead of field `len` and used that struct as a parameter for closure `f` (both `Vec` and `BrokenVec` are based on our `ArrayList`). While creating a new type works, it is clearly preferrable to avoid it. To better support such examples, a possible extension of our approach would be to enable defining multiple core invariants that can be switched when verifying unsafe code. An alternative extension would be to introduce an annotation that enables specifying that a type invariant is broken. However, it is unclear how to elegantly integrate such an extension with our approach because we check that the type invariants hold not only at the function boundaries, but also at each assignment and closing of a borrow.

[195]: Jung (2021), *Comment on Rust issue #60847*

`bam::bgzip::Block::load`. This example also calls a user-provided function to initialise the contents of a vector. However, in this example, the vector is created already initialised with zeroes and then cleared. The following snippet shows the key pattern.

```
// In constructor:
let value: Vec<u8> = vec![0u8; size];
value.clear();
// In method read:
unsafe { value.set_len(size); }
let slice = &mut *value;
read_exact(slice, size);
```

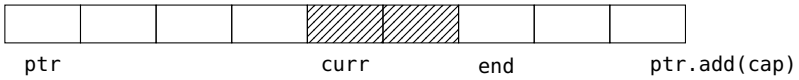
The key difference from the previous example is that reference `slice` pointing into the contents of the vector in this example is a valid one even though the vector was cleared after it was initialised. Clearing the vector

does not affect the initialisation of contents because the vector holds elements of unsigned integers `u8` that implement trait `Copy`. One property of types that implement trait `Copy` is that they do not implement `Drop` and, therefore, clearing the vector is equivalent to just changing field `len` of the vector to 0. Olivia verified this example by adding a ghost field to `Vec<u8>` that tracks the actual number of initialised elements and specifying that the implementation of method `clear` preserves the initialisation. This example shows that it would be beneficial to extend our approach with the ability to make specifications dependent on whether a type parameter implements some trait.

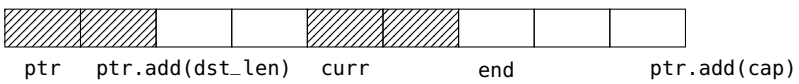
`std::vec::Vec::from_iter`. Rust iterators enable expressing complex patterns with a few lines of code. For instance, in the following snippet we consume a vector into an iterator with `into_iter`, filter some elements out, map the remaining ones, and construct a new vector out of the result.

```
1 let new_vec = vec.into_iter().filter(/*...*/).map(/*...*/)
2                  .collect();
```

To achieve good performance, Rust iterators are heavily optimised. For example, in some cases `new_vec` will reuse the allocation of `vec` thus saving the overhead of doing a large allocation. Function `std::vec::Vec::from_iter` is exactly the function that implements a part of such optimised iteration. Verifying this example is interesting because it employs all three techniques for ensuring panic safety. Since the entire function is too large to verify with our performance issues, Olivia targeted the final fragment of this function that constructs the new iterator because that is the location where RUDRA found the bug. An interesting aspect of function `into_iter` is that it looks into memory from two different viewpoints. One viewpoint is the type invariant of the iterator that at the beginning of the fragment specifies that the memory looks as shown in the following diagram.



Pointer `ptr` points to a memory block of capacity `cap`, pointer `curr` points to the current element, and pointer `end` points one past the last element. Memory between `ptr` and `curr` is uninitialised because these elements are already consumed while memory between `curr` and `end` is initialised and contains the *remaining* elements. The code in the function extends the viewpoint of the invariant and specifies that first `dst_len` elements are initialised and store the newly computed *destination* elements as shown in the following diagram.



The fragment we want to verify drops the remaining elements and turns the allocation with destination elements into a vector. The following snippet shows the key parts of the fragment.

```
1 src.forget_allocation_drop_remaining();
2 unsafe { Vec::from_raw_parts(dst_buf, len, cap) };
```

Variable `src` is the iterator and its method `forget_allocation_drop_remaining` drops the remaining elements. The key challenge with verifying this function call is that this function is a safe function defined on the iterator, which means that it views memory from the viewpoint of the iterator and is not aware about the initialised elements at the beginning of the memory block, which it needs to preserve. We can stash the elements at the beginning of the block to switch from the function viewpoint to the type invariant viewpoint. However, to unstash the elements, we need to prove that the bit-pattern of the block was preserved, which we cannot do because the function can express only that snapshots are preserved and we did not include the values of raw memory into the snapshots. This example shows that for verifying some unsafe code, it is necessary to make values of potentially uninitialised memory a part of snapshots. Besides not being able to specify that the function preserves the values of the uninitialised memory, verifying it is straightforward. The function employs two techniques for ensuring panic safety (calling only functions that do not panic and changing the type invariant to safe one before entering the code that may panic), which pose no challenges for our approach.

A later version of function `from_iter` includes a drop guard that tries to deallocate the destination elements in case dropping the remaining elements panics. The following snippet shows the code of the changed fragment.

```
1 let dst_guard = InPlaceDstBufDrop { /* ... */ };
2 src.forget_allocation_drop_remaining();
3 std::mem::forget(dst_guard);
4 unsafe { Vec::from_raw_parts(dst_buf, len, cap) };
```

`InPlaceDstBufDrop` is a private struct with a drop handler that drops the destination elements in case a panic occurs. If this code does not panic, this drop handler is cancelled by calling function `forget`. Olivia did not attempt to verify this version of the code. However, we did verify the call to the drop handler after her thesis, gaining some evidence that our approach for verifying code with drop guards is suitable for verifying realistic code (we replaced `forget_allocation_drop_remaining` with an unsafe function that takes raw memory instead of the iterator to be able to prove the unstash operation). Unfortunately, we were not able to verify the body of the drop handler because verification does not terminate within an hour, even though it should trivially verify since its precondition matches exactly the precondition of called function `Vec::from_raw_parts(dst_buf, len, cap)`[3].

3: This example is one of the reasons why we think that the cause of slow performance we identified in Chapter A could be not the only one.

This example shows the importance of supporting all three techniques for ensuring panic safety and splitting and merging memory ranges. It also demonstrates that unsafe code may care that safe functions preserve the values of potentially uninitialised memory, which requires snapshots to include the values of raw memory. Making values of raw memory part of snapshots would be a straightforward change to our approach.

`std::vec::Vec::dedup_by`. We used `dedup_by` in Chapter 17 to motivate the need to support drop guards and complex memory access patterns. Olivia did her thesis before the implementation of Prusti was complete; more specifically, Prusti did not have yet support for the `raw_range!(...)`

variant needed to specify function `std::ptr::copy`. Therefore, Olivia verified the example in Viper using hand-written encoding. Prusti now does support the features necessary to verify this example, but unfortunately the example cannot be verified due to the performance problem.

22.2.3 Discussion

This section presented two studies we used to evaluate our approach. While the performance problem prevents us from evaluating verification performance and annotation overhead, the studies confirm three key advantages of our approach compared to prior work. First, our work enables verifying memory safety and functional correctness of safe, unsafe, and mixed safe-unsafe Rust code using an SMT-based verifier. As a result, using our approach, the entire project can be verified with a single tool, even if the project contains code that uses patterns whose memory safety cannot be proven by the Rust compiler. Second, our work enables verifying the memory safety of code with implicit control flow such as panics. As we saw from our studies, this feature is crucial when verifying the memory safety of safe abstractions. Third, our studies show that with our approach, we can specify the interfaces of safe abstractions using a specification language based on Rust expressions we introduced in Part I. This result shows that our approach enables incremental verification: the programmers can specify and verify their safe code using the simple specification language from Part I, even if that safe code uses safe abstractions whose implementations were verified using more complex features introduced in this part of the thesis.

Our evaluation also revealed that to verify some of the examples, we need to extend our approach with additional features. We identified three extensions of our approach that are important for verifying real-world code. First, we found several examples that temporarily use a different core invariant for a specific value. In our evaluation, we rewrote such examples to use an additional type with the temporary invariant; however, allowing the user to switch temporarily to a different invariant for a specific value would be much better. Second, we found that sometimes the value of uninitialised memory is important, and, therefore, we should include it in the snapshot so that knowledge about it is preserved. Third, `ManuallyDrop` allows moving out values through a mutable reference into it, which our approach does not support. Since this type is one of the core building blocks for writing unsafe code, we need to find a way to support it. We are convinced that the first two extensions can be supported with minor modifications to our approach. Since `ManuallyDrop` is a special type in Rust, we could support it by treating it in a special way in the verifier. Supporting the general pattern is an open question we leave to future work.

Related Work 23

In the earlier parts of the thesis, we discussed the relevant related work on unbounded verification of safe and unsafe Rust. In this chapter, we discuss work related to the other aspects of the work presented in this part of the thesis.

23.1 Bug Hunting Tools for Rust

There are many tools developed for Rust that enable their users to efficiently find bugs, including in unsafe Rust code. These tools can be grouped into three categories: bounded verifiers, static analysers, and sanitisers. The bounded verifiers CRUST [99], Kani [196], Rust SMACK [101], and [103]'s and [197]'s work enable *bounded* verification of Rust programs containing unsafe code against specifications written as boolean Rust expressions. The static analysis tools RUDRA [172], SafeDrop [198], Rupair [199], MIRAI [200], and its extension MirChecker [201], and [158]'s and [202]'s work enable efficiently analysing large codebases with little to no input from the user and finding bugs for which the analyser was designed. Currently, these static analysers can find memory bugs, including ones caused by panics, wrong type declarations, as well as simple functional property violations such as out-of-bounds accesses. Sanitisers such as Valgrind [203] and Miri [204] execute a Rust program in instrumented mode and check for violations of various properties such as dangling pointers, memory leaks, and even aliasing rules such as Stacked Borrows [126] or Tree Borrows [127]. All bug hunting tools are effective at finding various classes of bugs, but they also share one key limitation compared to our work: they cannot be used to prove the absence of bugs.

23.2 Handling Untrusted Code

One of our goals was to enable verifying that a safe abstraction guarantees memory safety even when used by unverified safe clients. As we discussed in the introduction, this property is important because it gives peace of mind to the programmers writing safe clients because they can be sure that their code will not cause memory errors. More generally, the question is how can we ensure that the guarantees provided by checked code (either verified, or checked by some type system) are not invalidated by unchecked code. Related work on protecting checked code from unchecked code could be grouped into three categories: work that, like ours, uses verification to ensure that checked (verified) code guards properly itself from unchecked (unverified) code using runtime assertions and private state, work that uses runtime isolation to separate the two, and hybrid approaches.

[99]: Toman et al. (2015), 'Crust: A Bounded Verifier for Rust (N)'

[196]: VanHattum et al. (2022), 'Verifying Dynamic Trait Objects in Rust'

[101]: Baranowski et al. (2018), 'Verifying Rust Programs with SMACK'

[103]: Lindner et al. (2018), 'No Panic! Verification of Rust Programs by Symbolic Execution'

[197]: Kan et al. (2020), 'An Executable Operational Semantics for Rust with the Formalization of Ownership and Borrowing'

[172]: Bae et al. (2021), 'Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale'

[198]: Cui et al. (2021), 'SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-Flow Analysis'

[199]: Hua et al. (2021), 'Rupair: Towards Automatic Buffer Overflow Detection and Rectification for Rust'

[200]: Venter (2023), *MIRAI: Rust mid-level IR Abstract Interpreter*

[201]: Li et al. (2021), 'MirChecker: Detecting Bugs in Rust Programs via Static Analysis'

[158]: Qin et al. (2020), 'Understanding memory and thread safety practices and issues in real-world Rust programs'

[202]: Li et al. (2022), 'Detecting Cross-language Memory Management Issues in Rust'

[203]: Nethercote et al. (2007), 'Valgrind: a framework for heavyweight dynamic binary instrumentation'

[204]: Team (2023), *Miri: An interpreter for Rust's mid-level intermediate representation*

[126]: Jung et al. (2020), 'Stacked borrows: an aliasing model for Rust'

[127]: Villani (2023), 'Tree Borrows'

The most similar approach to ours is probably the one by David Cok and Rustan Leino [205], who developed an approach for specifiying a boundary between verified and unverified code for JML and Dafny. The two approaches have many similarities on the technical level, but have important differences on the meta level due to different goals. The goal of Cok and Leino's approach is to ensure that a verified function validates its input and "fails fast" with an appropriate error in case input is invalid. Their core idea is to replace in the boundary functions between verified and non-verified code the `requires Pre` construct used for regular preconditions with a `recommends Pre else Exc` construct. Similarly to `requires`, `recommends` expresses a condition that should hold when the function is called. However, the function is not allowed to assume that the condition holds: instead, it needs to check it and if it does not hold, throw an exception specified in the `else` clause. A fundamental difference between the two approaches is that Cok and Leino aim to guard verified code from invalid inputs while we aim to ensure that verified code is memory safe when called with untrusted inputs and when making callbacks to unverified clients. For example, if a client violates a condition specified in `recommends`, the implementation is required to throw an exception and leave the state unchanged as if the call did not happen while violating the functional precondition does not give any guarantees about the behaviour of the function, except that it does not violate memory safety. Therefore, the two approaches are complementary and could be combined for functional correctness as described in the paper. Besides fundamental differences, Cok and Leino's approach has many technical similarities to our approach. In some cases, our functional preconditions specify validity conditions of the input and act similarly to `recommends`. For example, the functional precondition of method `ArrayList::index` specifies that the index parameter must be in bounds and violating this requirement leads to a panic. Our `checked!` annotation is the exact opposite of their `//@ allow`, which they use to express that the annotated operation may throw an exception.

[205]: Cok et al. (2022), 'Specifying the Boundary Between Unverified and Verified Code'

The approaches that use runtime isolation to protect safe from unsafe Rust are Galeed [206], Sandcrust [207], and XRust [208]. These approaches create runtime isolation between memory accessed by safe and unsafe Rust and check the data on the boundary so that unsafe Rust (and C code called by it) cannot corrupt the memory used by safe Rust. A similar path was taken by [209] for protecting C modules verified in VeriFast from the unverified ones: they used module private memory for storing data of verified modules and runtime checked all specifications at the boundaries. The key challenge they solved was how to check separation logic assertions at runtime. We avoid the need for any runtime isolation and the runtime cost associated with it by explicitly checking that we rely only on the validity of types, which is guaranteed by the Rust compiler for safe code.

[206]: Rivera et al. (2021), 'Keeping Safe Rust Safe with Galeed'

[207]: Lamowski et al. (2017), 'Sandcrust: Automatic Sandboxing of Unsafe Components in Rust'

[208]: Liu et al. (2020), 'Securing unsafe Rust programs with XRust'

[209]: Agten et al. (2015), 'Sound Modular Verification of C Code Executing in an Unverified Context'

23.3 Panic Safety

Rust panics are similar to exceptions in programming languages like C++ and Java. Abrahams [210] described three possible levels of exception safety in the context of C++:

[210]: Abrahams (1998), 'Exception-Safety in Generic Components'

1. *Basic*: the invariants are preserved and no resources are leaked.
2. *Strong*: the operation either completes successfully, or throws an exception leaving the state exactly as it was before throwing the exception.
3. *No-throw*: the operation guarantees to not throw an exception.

Our approach by default guarantees level *basic* but without a guarantee that no resources are leaked because leaking in Rust is considered safe. The user can optionally ensure level *no-throw* by using the `#[no_panic(...)]` attribute. Currently, our approach cannot be used to ensure level *strong* (we could ensure it by adopting the technique from [205]). However, since panics in Rust are intended for unrecoverable errors, level *strong* does not seem to be useful in practice.

[205]: Cok et al. (2022), 'Specifying the Boundary Between Unverified and Verified Code'

Spec#, a superset of C# designed for verification, ensures exception safety in a similar way to ours [187]: it requires object invariants to hold regardless wither the execution leaves the method due to successfully returning or due to throwing an exception. However, since the languages are quite different, the mechanisms used to ensure the invariants are very different. Spec# does not have drop handlers and instead uses `try { ... } catch { ... }` to handle exceptions and potentially perform the necessary ghost operations. In contrast, our approach based on attaching ghost code to drop handlers is idiomatic to Rust.

[187]: Leino et al. (2004), 'Exception Safety for C#'

Conclusions of Part III 24

In this thesis we aim to enable incremental verification in two dimensions. In Part I, we focused on the first dimension of making incremental the verification of safe Rust. In this part of the thesis, we focused on the second dimension of enabling a smooth transition to verifying unsafe code where some of the Rust type system guarantees do not hold anymore and memory safety has to be ensured manually. The approach we presented in this part of the thesis addresses two key challenges. First, it enables functional verification of mixed safe-unsafe code that may use raw pointers and call unsafe functions. For parts of the code that are completely safe, the verification approach is as lightweight as the one in Part I while the complexity of verifying unsafe code directly depends on the complexity of unsafe code. We achieve this goal by extending the approach from Part I to work on mixed safe-unsafe code. Importantly, we adapt the PCS elaboration algorithm to generate the core proof for safe parts of the code even if the same function contains some unsafe code, which enables us to hide a large part of tedious and complex work of writing the core proof from the user. Second, our approach enables verifying that safe abstractions are memory safe even when used by unverified safe clients. We achieve this goal by verifying that safe abstractions do not make unjustified assumptions (for example, that their clients respect their preconditions) and providing specification constructs that enable the programmers to verify correctness of error-prone patterns used to ensure panic safety. While our evaluation is limited due to performance problem in the underlying infrastructure, it still shows that our approach is expressive enough to verify important patterns used when writing unsafe code.

25 Conclusions and Future Work

At the beginning of the thesis, we identified five requirements that a deductive verifier needs to satisfy if we want it to be considered by programmers as part of their toolbox: automation, scalability, expressivity, low entry bar, and ability to focus. Advancements in automated theorem provers such as SMT solvers enabled building verifiers that can automatically verify simple programs. However, scaling verification beyond trivial examples requires using modular techniques that require addressing the frame problem. The existing verification techniques that address the frame problem can be divided into two camps. The first camp contains approaches like Universe Types, Spec#, SYMPLAR, and SPARK that provide a relatively low entry bar and allow the programmer to focus on the properties they care about without large prior investment at the cost of expressivity. The second camp contains approaches like VCC and permission logics that are expressive, but require expert knowledge and a significant upfront investment before a user can focus on the properties they are interested in. Importantly, when starting a new project, the user had to choose a methodology and, for example, if they chose one from the easier-to-use camp, they could not switch to a more expressive one for the parts of the project that require more expressive power.

In this thesis, we improved the state of the art by developing an approach that enables incremental verification in two dimensions. In Part I, we showed that by using the strong uniqueness guarantees available in safe Rust, we can make the verification truly incremental. The prior investment that a programmer needs to make with our approach is effectively zero; the programmer can immediately focus on the properties they are interested in. Moreover, our specification language based on Rust expressions enables users to focus on verifying functional properties without learning complex logics, which keeps the entry bar low. We achieved these goals by solving three challenges. We developed a technique that, as shown by our evaluation, can completely automatically generate core proofs for realistic Rust programs, including ones that use borrows. We showed how user-written functional specifications can be intertwined with the automatically generated core proof. Moreover, we developed pledges, a specification construct that enables specifying the effects of mutable borrows in a modular and information hiding preserving way. As we have shown in our analysis in Part II, most Rust code is safe and, therefore, the technique described in Part I is applicable to it. However, the analysis also showed that there is a substantial amount of unsafe code, most of which calls unsafe functions and manipulates raw pointers. Therefore, in Part III, we extended our technique from Part I with primitives necessary for verifying unsafe code. An important difference between our approach and prior work is that we enable the users to still verify safe parts of their code using the simple technique presented in Part I and require to use more advanced constructs only for the parts that use unsafe code. While an in-depth evaluation was prevented by performance issues in

the underlying tools, our evaluation still showed that our technique is expressive enough to verify important data structures.

In this thesis, we focused on the Rust programming language. However, since Rust took off, the designers of over languages started experimenting with integrating uniqueness and borrowing. For example, SPARK added support for Rust-inspired unique pointers [211]. Therefore, we believe that our findings will be useful beyond Rust.

[211]: Maalej et al. (2018), 'Safe Dynamic Memory Management in Ada and SPARK'

In the remainder of this chapter, we discuss possible directions of future work related to deductive verification of imperative heap-manipulating programs.

Performance of Iterated Separating Conjunctions. The key problem that blocked us from properly evaluating our approach in Part III is the performance of Viper's symbolic execution algorithm used for handling iterated separating conjunctions. Since addressing this performance problem is crucial for making the approach presented in Part III for verifying existing codebases such as data structures in the Rust standard library, fixing it should be the immediate goal of the future work. We started an investigation and present our preliminary results in Chapter A. However, more time is needed to fully understand the problem and evaluate the fixes we proposed in Chapter A.

Lifetime Logic for Functional Verification. In Chapter 21, we presented an interface of a RustBelt and RustHornBelt inspired lifetime logic suitable for SMT-based verification of functional correctness. An important future work is to formally define and proof soundness of such a logic. One possible way would be to try to hide the RustHornBelt logic behind the interface we proposed.

Specification Language. When designing the specification language in Part I, we tried to make it as close to executable Rust as possible with the hope that it will be more intuitive for Rust programmers. For example, we borrow check and type-check the specifications and clearly separate the before and after states of the borrow. Other Rust verifiers such as Creusot took a different design and treat all values in specifications as if they were copy types. It would be interesting to conduct a user study to determine, which specification style is more intuitive for programmers.

[185]: Furrer (2023), 'Verifying Vulnerability Fixes in a Rust Verifier'

Switchable Type Invariants. The study done by Olivia [185] revealed that often unsafe code needs to temporarily use a different type invariant for a specific value. In her thesis, Olivia worked around the problem by creating custom types. However, to be able to verify existing unsafe code without modifying it, we would need a mechanism for temporarily changing the type invariant of a value. While it is clear how to change the encoding to support such a feature, there are many questions related to specification design and implementation in the tool.

Verifying Adherence to Tree Borrows. Tree Borrows maintains for each byte a tree data structure that tracks all pointers and references that could potentially access that byte. The key challenge in verifying that code adheres to the Tree Borrows model is finding an abstraction that allows soundly overapproximating the state of the tree. Since the Tree Borrows model was designed as an operational version of the borrow checker, it is likely that a more flexible version of the lifetime annotations (for example, that supports using constraints as arbitrary predicates in specifications) should be sufficient for specifying the abstract state of the tree in a way suitable for verification. However, a complete design and evaluation is needed to determine whether this idea could work in practice.

Bibliography

[1] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997 (cited on page 2).

[2] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 'VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java'. In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. Ed. by Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 41–55. DOI: 10.1007/978-3-642-20398-5_4 (cited on pages 2, 3, 5, 7).

[3] Manuel Fähndrich and K. Rustan M. Leino. 'Declaring and checking non-null types in an object-oriented language'. In: *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*. Ed. by Ron Crocker and Guy L. Steele Jr. ACM, 2003, pp. 302–312. DOI: 10.1145/949305.949332 (cited on page 2).

[4] John McCarthy and Patrick J Hayes. 'Some philosophical problems from the standpoint of artificial intelligence'. In: *Machine intelligence*. Edinburgh University Press, 1969, pp. 463–502 (cited on page 2).

[5] Alexander Borgida, John Mylopoulos, and Raymond Reiter. 'On the Frame Problem in Procedure Specifications'. In: *IEEE Trans. Software Eng.* 21.10 (1995), pp. 785–798. DOI: 10.1109/32.469460 (cited on page 2).

[6] R. W. Floyd. 'Assigning meanings to programs'. In: *Proceedings of the American Mathematical Society Symposia on Applied Mathematics* 19 (1967), pp. 19–31 (cited on page 3).

[7] C. A. R. Hoare. 'An Axiomatic Basis for Computer Programming'. In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259 (cited on page 3).

[8] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 'Z3: An Efficient SMT Solver'. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24 (cited on pages 3, 9).

[9] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 'CVC5: A Versatile and Industrial-Strength SMT Solver'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24 (cited on page 3).

[10] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. Vol. 2262. Lecture Notes in Computer Science. Springer, 2002 (cited on pages 4, 8).

[11] David G. Clarke, John Potter, and James Noble. 'Ownership Types for Flexible Alias Protection'. In: *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1998, Vancouver, British Columbia, Canada, October 18-22, 1998*. Ed. by Bjørn N. Freeman-Benson and Craig Chambers. ACM, 1998, pp. 48–64. DOI: 10.1145/286936.286947 (cited on page 4).

[12] K. Rustan M. Leino and Peter Müller. 'Object Invariants in Dynamic Contexts'. In: *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*. Ed. by Martin Odersky. Vol. 3086. Lecture Notes in Computer Science. Springer, 2004, pp. 491–516. DOI: 10.1007/978-3-540-24851-4_22 (cited on pages 4, 9).

[13] Werner Dietl and Peter Müller. 'Universes: Lightweight Ownership for JML'. In: *J. Object Technol.* 4.8 (2005), pp. 5–32. DOI: 10.5381/jot.2005.4.8.a1 (cited on page 4).

[14] Werner Michael Dietl. 'Universe Types - Topology, Encapsulation, Genericity, and Tools'. PhD thesis. ETH Zürich, Switzerland, 2009. DOI: 10.3929/ethz-a-005951213 (cited on page 4).

[15] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. 'Modular specification of frame properties in JML'. In: *Concurr. Comput. Pract. Exp.* 15.2 (2003), pp. 117–154. DOI: 10.1002/CPE.713 (cited on page 4).

[16] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 'Specification and verification: the Spec# experience'. In: *Commun. ACM* 54.6 (2011), pp. 81–91. DOI: 10.1145/1953122.1953145 (cited on page 4).

[17] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 'VCC: A Practical System for Verifying Concurrent C'. In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, pp. 23–42. DOI: 10.1007/978-3-642-03359-9_2 (cited on page 5).

[18] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 'Local Reasoning about Programs that Alter Data Structures'. In: *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*. Ed. by Laurent Fribourg. Vol. 2142. Lecture Notes in Computer Science. Springer, 2001, pp. 1–19. DOI: 10.1007/3-540-44802-0_1 (cited on page 5).

[19] Jan Smans, Bart Jacobs, and Frank Piessens. 'Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic'. In: *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*. Ed. by Sophia Drossopoulou. Vol. 5653. Lecture Notes in Computer Science. Springer, 2009, pp. 148–172. DOI: 10.1007/978-3-642-03013-0_8 (cited on pages 5, 7, 8, 30).

[20] Peter W. O'Hearn. 'Resources, concurrency, and local reasoning'. In: *Theor. Comput. Sci.* 375.1-3 (2007), pp. 271–307. DOI: 10.1016/j.tcs.2006.12.035 (cited on page 5).

[21] John C. Reynolds. 'Separation Logic: A Logic for Shared Mutable Data Structures'. In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817 (cited on pages 6, 7, 65, 69).

[22] Neelakantan R. Krishnaswami. 'Reasoning about iterators with separation logic'. In: *Proceedings of the 2006 Conference on Specification and Verification of Component-Based Systems, SAVCBS '06, Portland, Oregon, USA, November 10-11, 2006*. ACM, 2006, pp. 83–86. DOI: 10.1145/1181195.1181213 (cited on page 6).

[23] John Boyland. 'Checking Interference with Fractional Permissions'. In: *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*. Ed. by Radhia Cousot. Vol. 2694. Lecture Notes in Computer Science. Springer, 2003, pp. 55–72. DOI: 10.1007/3-540-44898-5_4 (cited on pages 6, 59, 62).

[24] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 'Permission accounting in separation logic'. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. Ed. by Jens Palsberg and Martin Abadi. ACM, 2005, pp. 259–270. DOI: 10.1145/1040305.1040327 (cited on page 6).

[25] Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. 'Abstract Read Permissions: Fractional Permissions without the Fractions'. In: *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*. Ed. by Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni. Vol. 7737. Lecture Notes in Computer Science. Springer, 2013, pp. 315–334. DOI: 10.1007/978-3-642-35873-9_20 (cited on pages 6, 62, 84, 230).

[26] John Tang Boyland, Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 'Constraint Semantics for Abstract Read Permissions'. In: *Proceedings of 16th Workshop on Formal Techniques for Java-like Programs, FTfJP@ECOOP 2014, Uppsala, Sweden, July 28 - August 1, 2014*. Ed. by David J. Pearce. ACM, 2014, 2:1–2:6. DOI: 10.1145/2635631.2635847 (cited on page 6).

[27] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 'Smallfoot: Modular Automatic Assertion Checking with Separation Logic'. In: *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, pp. 115–137. DOI: 10.1007/11804192_6 (cited on pages 7, 9).

[28] Dino Distefano and Matthew J. Parkinson. 'jStar: towards practical verification for java'. In: *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*. Ed. by Gail E. Harris. ACM, 2008, pp. 213–226. DOI: 10.1145/1449764.1449782 (cited on page 7).

[29] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 'GRASShopper - Complete Heap Verification with Mixed Specifications'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. Springer, 2014, pp. 124–139. DOI: 10.1007/978-3-642-54862-8_9 (cited on page 7).

[30] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 'Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution'. In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9779. Lecture Notes in Computer Science. Springer, 2016, pp. 405–425. DOI: 10.1007/978-3-319-41528-4_22 (cited on pages 7, 97, 184, 188).

[31] Malte Schwerhoff and Alexander J. Summers. 'Lightweight Support for Magic Wands in an Automatic Verifier'. In: *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*. Ed. by John Tang Boyland. Vol. 37. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 614–638. DOI: 10.4230/LIPIcs.ECOOP.2015.614 (cited on pages 7, 97).

[32] Stefan Blom and Marieke Huisman. 'Witnessing the elimination of magic wands'. In: *Int. J. Softw. Tools Technol. Transf.* 17.6 (2015), pp. 757–781. DOI: 10.1007/s10009-015-0372-3 (cited on pages 7, 102).

[33] Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. 'Sound Automation of Magic Wands'. In: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*. Ed. by Sharon Shoham and Yakir Vizel. Vol. 13372. Lecture Notes in Computer Science. Springer, 2022, pp. 130–151. DOI: 10.1007/978-3-031-13188-2_7 (cited on pages 7, 97).

[34] Felix A. Wolf, Malte Schwerhoff, and Peter Müller. 'Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA'. In: *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*. Ed. by Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan. Vol. 13047. Lecture Notes in Computer Science. Springer, 2021, pp. 407–426. DOI: 10.1007/978-3-030-90870-6_22 (cited on page 7).

[35] Alexander Bakst and Ranjit Jhala. 'Predicate Abstraction for Linked Data Structures'. In: *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Vol. 9583. Lecture Notes in Computer Science. Springer, 2016, pp. 65–84. DOI: 10.1007/978-3-662-49122-5_3 (cited on page 7).

[36] Matthew J. Parkinson and Alexander J. Summers. 'The Relationship Between Separation Logic and Implicit Dynamic Frames'. In: *Log. Methods Comput. Sci.* 8.3 (2012). DOI: 10.2168/LMCS-8(3:1)2012 (cited on page 7).

[37] K. Rustan M. Leino and Peter Müller. 'A Basis for Verifying Multi-threaded Programs'. In: *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Giuseppe Castagna. Vol. 5502. Lecture Notes in Computer Science. Springer, 2009, pp. 378–393. DOI: 10.1007/978-3-642-00590-9_27 (cited on page 8).

[38] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 'Viper: A Verification Infrastructure for Permission-Based Reasoning'. In: *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Vol. 9583. Lecture Notes in Computer Science. Springer, 2016, pp. 41–62. DOI: 10.1007/978-3-662-49122-5_2 (cited on pages 8, 10, 23, 30).

[39] Marco Eilers and Peter Müller. 'Nagini: A Static Verifier for Python'. In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 596–603. DOI: 10.1007/978-3-319-96145-3_33 (cited on page 8).

[40] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. 'Gobra: Modular Specification and Verification of Go Programs'. In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. Lecture Notes in Computer Science. Springer, 2021, pp. 367–379. DOI: 10.1007/978-3-030-81685-8_17 (cited on page 8).

[41] Ioannis T. Kassios. 'Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions'. In: *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*. Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Vol. 4085. Lecture Notes in Computer Science. Springer, 2006, pp. 268–283. DOI: 10.1007/11813040_19 (cited on page 8).

[42] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. 'Regional Logic for Local Reasoning about Global Invariants'. In: *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*. Ed. by Jan Vitek. Vol. 5142. Lecture Notes in Computer Science. Springer, 2008, pp. 387–411. DOI: 10.1007/978-3-540-70592-5_17 (cited on page 8).

[43] K. Rustan M. Leino. 'Dafny: An Automatic Program Verifier for Functional Correctness'. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20 (cited on pages 8, 37, 188).

[44] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. 'The KeY Platform for Verification and Analysis of Java Programs'. In: *Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers*. Ed. by Dimitra Giannakopoulou and Daniel Kroening. Vol. 8471. Lecture Notes in Computer Science. Springer, 2014, pp. 55–71. DOI: 10.1007/978-3-319-12154-3_4 (cited on page 8).

[45] Kevin Bierhoff. 'Automated program verification made SYMPLAR: symbolic permissions for lightweight automated reasoning'. In: *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2011, part of SPLASH '11, Portland, OR, USA, October 22-27, 2011*. Ed. by Robert Hirschfeld and Eelco Visser. ACM, 2011, pp. 19–32. DOI: 10.1145/2048237.2048242 (cited on pages 8, 19, 83, 90).

[46] John Boyland, James Noble, and William Retert. 'Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only'. In: *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*. Ed. by Jørgen Lindskov Knudsen. Vol. 2072. Lecture Notes in Computer Science. Springer, 2001, pp. 2–27. DOI: 10.1007/3-540-45337-7_2 (cited on pages 8, 83).

[47] John Hogg. 'Islands: Aliasing Protection in Object-Oriented Languages'. In: *Proceedings of the Sixth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 1991, Phoenix, Arizona, USA, October 6-11, 1991*. Ed. by Andreas Paepcke. ACM, 1991, pp. 271–285. DOI: 10.1145/117954.117975 (cited on page 8).

[48] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 'LiquidHaskell: experience with refinement types in the real world'. In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*. Ed. by Wouter Swierstra. ACM, 2014, pp. 39–51. DOI: 10.1145/2633357.2633366 (cited on page 9).

[49] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 'Liquid types'. In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. Ed. by Rajiv Gupta and Saman P. Amarasinghe. ACM, 2008, pp. 159–169. DOI: 10.1145/1375581.1375602 (cited on page 9).

[50] Bernard Carré and Jonathan Garnsworthy. 'SPARK—an Annotated Ada Subset for Safety-critical Programming'. In: *Proceedings of the Conference on TRI-ADA '90*. TRI-Ada '90. Baltimore, Maryland, USA: ACM, 1990, pp. 392–402. DOI: 10.1145/255471.255563 (cited on page 9).

[51] Petar Maksimovic, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 'Gillian, Part II: Real-World Verification for JavaScript and C'. In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 827–850. DOI: 10.1007/978-3-030-81688-9_38 (cited on page 10).

[52] José Fragoso Santos, Petar Maksimovic, Sacha-Élie Ayoun, and Philippa Gardner. 'Gillian, part I: a multi-language platform for symbolic execution'. In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by Alastair F. Donaldson and Emina Torlak. ACM, 2020, pp. 927–942. DOI: 10.1145/3385412.3386014 (cited on page 10).

[53] Afshin Amighi, Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. 'The VerCors project: setting up basecamp'. In: *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012*. Ed. by Koen Claessen and Nikhil Swamy. ACM, 2012, pp. 71–82. DOI: 10.1145/2103776.2103785 (cited on page 10).

[54] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 'Boogie: A Modular Reusable Verifier for Object-Oriented Programs'. In: *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, pp. 364–387. DOI: 10.1007/11804192_17 (cited on pages 10, 102).

[55] Mário Pereira and António Ravara. 'Cameleer: A Deductive Verification Tool for OCaml'. In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 677–689. DOI: 10.1007/978-3-030-81688-9_31 (cited on page 10).

[56] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 'Frama-C: A software analysis perspective'. In: *Formal Aspects Comput.* 27.3 (2015), pp. 573–609. DOI: 10.1007/s00165-014-0326-7 (cited on page 10).

[57] Jean-Christophe Filliâtre and Andrei Paskevich. 'Why3 - Where Programs Meet Provers'. In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 125–128. DOI: 10.1007/978-3-642-37036-6_8 (cited on pages 10, 93).

[58] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. Tech. rep. Inria, 1997 (cited on pages 10, 218).

[59] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002 (cited on page 10).

[60] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 'Uniqueness and reference immutability for safe parallelism'. In: *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. Ed. by Gary T. Leavens and Matthew B. Dwyer. ACM, 2012, pp. 21–40. DOI: 10.1145/2384616.2384619 (cited on pages 10, 11, 83).

[61] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 'The Design and Formalization of Mezzo, a Permission-Based Programming Language'. In: *ACM Trans. Program. Lang. Syst.* 38.4 (2016), 14:1–14:94 (cited on pages 10, 11, 83).

[62] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 'Deny capabilities for safe, fast actors'. In: *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*. Ed. by Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos A. Varela. ACM, 2015, pp. 1–12. DOI: 10.1145/2824815.2824816 (cited on pages 11, 83).

[63] Sven Stork, Karl Naden, Joshua Sunshine, Manuel Mohr, Alcides Fonseca, Paulo Marques, and Jonathan Aldrich. 'Æminium: a permission based concurrent-by-default programming language approach'. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. Ed. by Michael F. P. O'Boyle and Keshav Pingali. ACM, 2014, p. 26. DOI: 10.1145/2594291.2594344 (cited on pages 11, 83).

[64] Nikhil Swamy, Michael W. Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 'Safe manual memory management in Cyclone'. In: *Sci. Comput. Program.* 62.2 (2006), pp. 122–144. DOI: 10.1016/j.scico.2006.02.003 (cited on pages 11, 83).

[65] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. 'Language support for fast and reliable message-based communication in singularity OS'. In: *Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006*. Ed. by Yolande Berbers and Willy Zwaenepoel. ACM, 2006, pp. 177–190. DOI: 10.1145/1217935.1217953 (cited on pages 11, 83).

[66] Philipp Haller and Martin Odersky. 'Capabilities for Uniqueness and Borrowing'. In: *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*. Ed. by Theo D'Hondt. Vol. 6183. Lecture Notes in Computer Science. Springer, 2010, pp. 354–378. DOI: 10.1007/978-3-642-14107-2_17 (cited on pages 11, 83).

[67] Dave Clarke and Tobias Wrigstad. 'External Uniqueness Is Unique Enough'. In: *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*. Ed. by Luca Cardelli. Vol. 2743. Lecture Notes in Computer Science. Springer, 2003, pp. 176–200. DOI: 10.1007/978-3-540-45070-2_9 (cited on page 11).

[68] Mads Tofte and Jean-Pierre Talpin. 'Region-based Memory Management'. In: *Inf. Comput.* 132.2 (1997), pp. 109–176. DOI: 10.1006/inco.1996.2613 (cited on page 11).

[69] Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. 'Region-Based Memory Management in Cyclone'. In: *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*. Ed. by Jens Knoop and Laurie J. Hendren. ACM, 2002, pp. 282–293. DOI: 10.1145/512529.512563 (cited on page 11).

[70] The Rust Project Developers. *The Rustonomicon: What Unsafe Rust Can Do*. Accessed March 21, 2023. 2023. URL: https://doc.rust-lang.org/nomicon/what-unsafe-does.html#what-unsafe-rust-can-do (cited on pages 11, 194).

[71] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 'RustBelt: securing the foundations of the Rust programming language'. In: *Proc. ACM Program. Lang.* 2.POPL (2018), 66:1–66:34. DOI: 10.1145/3158154 (cited on pages 11, 83, 87, 90, 144, 152, 167, 217).

[72] The Rust Project Developers. *The Rustonomicon: How Safe and Unsafe Interact*. Accessed March 21, 2023. 2023. URL: https://doc.rust-lang.org/nomicon/safe-unsafe-meaning.html (cited on pages 11, 151).

[73] The Rust Project Developers. *Announcing Rust 1.31 and Rust 2018*. Accessed September 27, 2023. 2018. URL: https://blog.rust-lang.org/2018/12/06/Rust-1.31-and-rust-2018.html#non-lexical-lifetimes (cited on pages 12, 42).

[74] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 'Leveraging Rust types for modular specification and verification'. In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 147:1–147:30. DOI: 10.1145/3360573 (cited on pages 13, 17, 20, 21, 27, 69, 72, 75, 76, 78, 79).

[75] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. *Leveraging Rust types for modular specification and verification (extended version)*. Tech. rep. ETH Zurich, 2019. DOI: 10.3929/ethz-b-000311092 (cited on page 13).

[76] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 'How do programmers use unsafe Rust?' In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 136:1–136:27. DOI: 10.1145/3428204 (cited on pages 13, 109, 121, 143, 144).

[77] Till Arnold. 'Optimization of Viper-Based Verifier'. BA thesis. ETH Zurich, 2020 (cited on page 14).

[78] Nicolas Winkler. 'Semantic Querying of Rust Code'. BA thesis. ETH Zurich, 2018 (cited on page 14).

[79] Rust community. *Rust: The Reference — Place Expressions and Value Expressions*. Accessed August 17, 2023. 2023. URL: https://doc.rust-lang.org/reference/expressions.html#place-expressions-and-value-expressions (cited on page 18).

[80] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 'Compositional shape analysis by means of bi-abduction'. In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. Ed. by Zhong Shao and Benjamin C. Pierce. ACM, 2009, pp. 289–300. DOI: 10.1145/1480881.1480917 (cited on page 29).

[81] Martin Abadi and Marcelo P. Fiore. 'Syntactic Considerations on Recursive Types'. In: *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*. IEEE Computer Society, 1996, pp. 242–252. DOI: 10.1109/LICS.1996.561324 (cited on page 30).

[82] Karl Crary, Robert Harper, and Sidd Puri. 'What is a Recursive Module?' In: *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*. Ed. by Barbara G. Ryder and Benjamin G. Zorn. ACM, 1999, pp. 50–63. DOI: 10.1145/301618.301641 (cited on page 30).

[83] Alexander J. Summers and Sophia Drossopoulou. 'A Formal Semantics for Isorecursive and Equirecursive State Abstractions'. In: *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*. Ed. by Giuseppe Castagna. Vol. 7920. Lecture Notes in Computer Science. Springer, 2013, pp. 129–153. DOI: 10.1007/978-3-642-39038-8_6 (cited on page 30).

[84] Jonathan Reem. *Void*. Accessed October 3, 2023. 2023. URL: https://crates.io/crates/void (cited on page 34).

[85] Federico Poli. 'Enabling Rich Lightweight Verification of Rust Software'. In progress. PhD thesis. ETH Zurich, 2024 (cited on page 37).

[86] The Rust Project Developers. *MIR borrow check*. Accessed August 22, 2023. 2023. URL: https://rustc-dev-guide.rust-lang.org/borrow_check.html (cited on pages 41, 42).

[87] The Rust Project Developers. *The Rust RFC Book: nll*. Accessed October 10, 2023. 2017. URL: https://rust-lang.github.io/rfcs/2094-nll.html (cited on pages 42, 113, 114).

[88] Nicholas D. Matsakis. *An alias-based formulation of the borrow checker*. Accessed September 27, 2023. 2018. URL: https://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/ (cited on page 42).

[89] Matthias Erdin. 'Verification of Rust Generics, Typestates, and Traits'. MA thesis. ETH Zurich, 2018 (cited on page 73).

[90] Florian Hahn. 'Rust2Viper: Building a static verifier for Rust'. MA thesis. ETH Zurich, 2015 (cited on pages 74, 84).

[91] Clippy contributors. *Clippy*. Accessed June 26, 2023. 2023. URL: https://github.com/rust-lang/rust-clippy (cited on pages 75, 138).

[92] Rust community. *The Rust community's crate registry*. Accessed March 30, 2023. 2023. URL: https://crates.io (cited on page 76).

[93] Rosetta Code contributors. *Rosetta Code*. Accessed November 5, 2018. 2018. URL: https://rosettacode.org/wiki/Category:Rust (cited on pages 78, 79).

[94] Rust community. *Learn Rust by writing Entirely Too Many Linked Lists*. Accessed April 4, 2019. 2019. URL: https://rust-unofficial.github.io/too-many-lists/first-final.html (cited on pages 78, 79).

[95] Nicholas D. Matsakis. *MIR-based borrowck is almost here*. Accessed April 4, 2019. 2018. URL: http://smallcultfollowing.com/babysteps/blog/2018/10/31/mir-based-borrowck-is-almost-here/ (cited on pages 78, 79).

[96] Nicholas D. Matsakis. *MIR-based borrow check (NLL) status update*. Accessed April 4, 2019. 2018. URL: http://smallcultfollowing.com/babysteps/blog/2018/06/15/mir-based-borrow-check-nll-status-update (cited on pages 78, 79, 113).

[97] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. *The CREUSOT Environment for the Deductive Verification of Rust Programs*. Research Report RR-9448. Inria Saclay - Île de France, Dec. 2021 (cited on pages 78, 93).

[98] Rust community. *Learn Rust by writing Entirely Too Many Linked Lists*. Accessed April 4, 2019. 2019. URL: https://rust-unofficial.github.io/too-many-lists/second-final.html (cited on page 80).

[99] John Toman, Stuart Pernsteiner, and Emina Torlak. 'Crust: A Bounded Verifier for Rust (N)'. In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. Ed. by Myra B. Cohen, Lars Grunske, and Michael Whalen. IEEE Computer Society, 2015, pp. 75–80. DOI: 10.1109/ASE.2015.77 (cited on pages 84, 253).

[100] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 'A Tool for Checking ANSI-C Programs'. In: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15 (cited on page 84).

[101] Marek S. Baranowski, Shaobo He, and Zvonimir Rakamaric. 'Verifying Rust Programs with SMACK'. In: *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 11138. Lecture Notes in Computer Science. Springer, 2018, pp. 528–535. DOI: 10.1007/978-3-030-01090-4_32 (cited on pages 84, 253).

[102] Zvonimir Rakamaric and Michael Emmi. 'SMACK: Decoupling Source Language Details from Verifier Implementations'. In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 106–113. DOI: 10.1007/978-3-319-08867-9_7 (cited on page 84).

[103] Marcus Lindner, Jorge Aparicius, and Per Lindgren. 'No Panic! Verification of Rust Programs by Symbolic Execution'. In: *16th IEEE International Conference on Industrial Informatics, INDIN 2018, Porto, Portugal, July 18-20, 2018*. IEEE, 2018, pp. 108–114. DOI: 10.1109/INDIN.2018.8471992 (cited on pages 84, 253).

[104] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 'KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs'. In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 209–224 (cited on page 84).

[105] Sebastian Ullrich. 'Simple Verification of Rust Programs via Functional Purification'. MA thesis. Karlsruhe Institute of Technology, Dec. 2016 (cited on pages 84, 91).

[106] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 'The Lean Theorem Prover (System Description)'. In: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. Lecture Notes in Computer Science. Springer, 2015, pp. 378–388. DOI: 10.1007/978-3-319-21401-6_26 (cited on page 84).

[107] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 'Combinators for bi-directional tree transformations: a linguistic approach to the view update problem'. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. Ed. by Jens Palsberg and Martin Abadi. ACM, 2005, pp. 233–246. DOI: 10.1145/1040305.1040325 (cited on page 84).

[108] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. 'Constructing Semantic Models of Programs with the Software Analysis Workbench'. In: *Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers*. Ed. by Sandrine Blazy and Marsha Chechik. Vol. 9971. Lecture Notes in Computer Science. 2016, pp. 56–72. DOI: 10.1007/978-3-319-48869-1_5 (cited on page 84).

[109] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 'Verus: Verifying Rust Programs using Linear Ghost Types'. In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (2023), pp. 286–315. DOI: 10.1145/3586037 (cited on pages 84, 152).

[110] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 'Verus: Verifying Rust Programs using Linear Ghost Types (extended version)'. In: *CoRR* abs/2303.05491 (2023). DOI: 10.48550/arXiv.2303.05491 (cited on page 84).

[111] Verus Developers. *Verus Tutorial and Reference: Memory safety is conditional on verification*. Accessed April 9, 2023. 2023. URL: https://verus-lang.github.io/verus/guide/memory-safety.html (cited on pages 85, 152).

[112] Nico Lehmann, Adam Geller, Gilles Barthe, Niki Vazou, and Ranjit Jhala. 'Flux: Liquid Types for Rust'. In: *CoRR* abs/2207.04034 (2022). DOI: 10.48550/arXiv.2207.04034 (cited on page 85).

[113] Fabian Wolff, Aurel Bilý, Christoph Matheja, Peter Müller, and Alexander J. Summers. 'Modular specification and verification of closures in Rust'. In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–29. DOI: 10.1145/3485522 (cited on page 85).

[114] Aurel Bılý, Jonas Hansen, Peter Müller, and Alexander J. Summers. 'Compositional Reasoning for Side-effectful Iterators and Iterator Adapters'. In: *CoRR* abs/2210.09857 (2022). DOI: 10.48550/ARXIV.2210.09857 (cited on page 85).

[115] Xavier Denis and Jacques-Henri Jourdan. 'Specifying and Verifying Higher-order Rust Iterators'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Vol. 13994. Lecture Notes in Computer Science. Springer, 2023, pp. 93–110. DOI: 10.1007/978-3-031-30820-8_9 (cited on page 85).

[116] Eric Reed. 'Patina: A formalization of the Rust programming language'. In: *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02* 264 (2015) (cited on page 85).

[117] Sergio Benitez. 'Short Paper: Rusty Types for Solid Safety'. In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*. Ed. by Toby C. Murray and Deian Stefan. ACM, 2016, pp. 69–75. DOI: 10.1145/2993600.2993604 (cited on page 85).

[118] Karl Crary, David Walker, and J. Gregory Morrisett. 'Typed Memory Management in a Calculus of Capabilities'. In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. by Andrew W. Appel and Alex Aiken. ACM, 1999, pp. 262–275. DOI: 10.1145/292540.292564 (cited on page 85).

[119] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 'Oxide: The Essence of Rust'. In: *CoRR* abs/1903.00982 (2019) (cited on pages 86, 100).

[120] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 'Oxide: The Essence of Rust'. In: *CoRR* abs/1903.00982 (2021) (cited on page 86).

[121] Will Crichton, Marco Patrignani, Maneesh Agrawala, and Pat Hanrahan. 'Modular information flow through ownership'. In: *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. Ed. by Ranjit Jhala and Isil Dillig. ACM, 2022, pp. 1–14. DOI: 10.1145/3519939.3523445 (cited on page 86).

[122] Shuanglong Kan, David Sanán, Shang-Wei Lin, and Yang Liu. 'K-Rust: An Executable Formal Semantics for Rust'. In: *CoRR* abs/1804.07608 (2018) (cited on page 86).

[123] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. 'KRust: A Formal Executable Semantics of Rust'. In: *2018 International Symposium on Theoretical Aspects of Software Engineering, TASE 2018, Guangzhou, China, August 29-31, 2018*. Ed. by Jun Pang, Chenyi Zhang, Jifeng He, and Jian Weng. IEEE Computer Society, 2018, pp. 44–51. DOI: 10.1109/TASE.2018.00014 (cited on page 86).

[124] Grigore Rosu and Traian-Florin Serbanuta. 'An overview of the K semantic framework'. In: *J. Log. Algebraic Methods Program.* 79.6 (2010), pp. 397–434. DOI: 10.1016/J.JLAP.2010.03.012 (cited on page 86).

[125] David J. Pearce. 'A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust'. In: *ACM Trans. Program. Lang. Syst.* 43.1 (2021), 3:1–3:73. DOI: 10.1145/3443420 (cited on page 86).

[126] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 'Stacked borrows: an aliasing model for Rust'. In: *Proc. ACM Program. Lang.* 4.POPL (2020), 41:1–41:32. DOI: 10.1145/3371109 (cited on pages 86, 151, 200, 202, 253).

[127] Neven Villani. 'Tree Borrows'. Accessed March 30, 2023. MA thesis. ENS Paris-Saclay, 2023 (cited on pages 86, 151, 200, 253).

[128] Neven Villani. *Tree Borrows: A new aliasing model for Rust*. Accessed March 30, 2023. 2023. URL: https://perso.crans.org/vanille/treebor/ (cited on pages 86, 151, 200).

[129] Ralf Jung. 'Understanding and evolving the Rust programming language'. PhD thesis. Saarland University, Saarbrücken, Germany, 2020 (cited on pages 87, 88, 219, 220, 223, 225).

[130] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 'RustBelt meets relaxed memory'. In: *Proc. ACM Program. Lang.* 4.POPL (2020), 34:1–34:29. DOI: 10.1145/3371102 (cited on page 90).

[131] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 'GhostCell: separating permissions from data in Rust'. In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–30. DOI: 10.1145/3473597 (cited on pages 90, 231).

[132] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 'RustHorn: CHC-Based Verification for Rust Programs'. In: *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. Ed. by Peter Müller. Vol. 12075. Lecture Notes in Computer Science. Springer, 2020, pp. 484–514. DOI: 10.1007/978-3-030-44914-8_18 (cited on pages 91, 93).

[133] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 'RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code'. In: *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. Ed. by Ranjit Jhala and Isil Dillig. ACM, 2022, pp. 841–856. DOI: 10.1145/3519939.3523704 (cited on pages 91, 93, 152, 217).

[134] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 'Creusot: A Foundry for the Deductive Verification of Rust Programs'. In: *Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings*. Ed. by Adrián Riesco and Min Zhang. Vol. 13478. Lecture Notes in Computer Science. Springer, 2022, pp. 90–105. DOI: 10.1007/978-3-031-17244-1_6 (cited on pages 91, 93, 239).

[135] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. 'Synthesizing software verifiers from proof rules'. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. Ed. by Jan Vitek, Haibo Lin, and Frank Tip. ACM, 2012, pp. 405–416. DOI: 10.1145/2254064.2254112 (cited on page 91).

[136] Nikolaj S. Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. 'Horn Clause Solvers for Program Verification'. In: *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. Ed. by Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte. Vol. 9300. Lecture Notes in Computer Science. Springer, 2015, pp. 24–51. DOI: 10.1007/978-3-319-23534-9_2 (cited on page 91).

[137] Malte Schwerhoff. 'Advancing Automated, Permission-Based Program Verification Using Symbolic Execution'. PhD thesis. ETH Zurich, Zürich, Switzerland, 2016. DOI: 10.3929/ETHZ-A-010835519 (cited on page 95).

[138] The dafny-lang community. *Dafny Reference Manual: Two-state lemmas and functions*. Accessed October 31, 2023. 2023. URL: https://dafny.org/latest/DafnyRef/DafnyRef#sec-two-state (cited on page 99).

[139] Son Ho and Jonathan Protzenko. 'Aeneas: Rust verification by functional translation'. In: *Proc. ACM Program. Lang.* 6.ICFP (2022), pp. 711–741. DOI: 10.1145/3547647 (cited on page 100).

[140] Diem Association. *Move Specification Language (version 1.4): Expressiveness*. Accessed November 2, 2023. 2023. URL: https://github.com/diem/move/blob/main/language/move-prover/doc/user/spec-lang.md#expressiveness (cited on page 102).

[141] Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark W. Barrett, and David L. Dill. 'The Move Prover'. In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12224. Lecture Notes in Computer Science. Springer, 2020, pp. 137–150. DOI: 10.1007/978-3-030-53288-8_7 (cited on page 102).

[142] David L. Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Jingyi Emma Zhong. 'Fast and Reliable Formal Verification of Smart Contracts with the Move Prover'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 183–200. DOI: 10.1007/978-3-030-99524-9_10 (cited on page 102).

[143] Georges-Axel Jaloyan, Claire Dross, Maroua Maalej, Yannick Moy, and Andrei Paskevich. 'Verification of Programs with Pointers in SPARK'. In: *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings*. Ed. by Shang-Wei Lin, Zhe Hou, and Brendan P. Mahony. Vol. 12531. Lecture Notes in Computer Science. Springer, 2020, pp. 55–72. DOI: 10.1007/978-3-030-63406-3_4 (cited on page 102).

[144] Claire Dross and Johannes Kanig. 'Recursive Data Structures in SPARK'. In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 178–189. DOI: 10.1007/978-3-030-53291-8_11 (cited on page 102).

[145] SPARK Developers. *SPARK's User's Guide: Annotation for Referring to a Value at the End of a Local Borrow*. Accessed February 8, 2024. 2023. URL: https://docs.adacore.com/spark2014-docs/html/ug/en/appendix/additional_annotate_pragmas.html#annotation-for-referring-to-a-value-at-the-end-of-a-local-borrow (cited on page 102).

[146] Unsafe Code Guidelines Working Group. *Project website*. Accessed August 17, 2020. 2020. URL: https://github.com/rust-lang/unsafe-code-guidelines (cited on page 109).

[147] Ralf Jung. *The Scope of Unsafe*. Accessed April 4, 2019. 2016. URL: https://www.ralfj.de/blog/2016/01/09/the-scope-of-unsafe.html (cited on pages 109, 144).

[148] The Rust Project Developers. *The Rustonomicon*. Accessed May 2, 2020. 2023. URL: https://doc.rust-lang.org/nomicon/ (cited on pages 110, 133).

[149] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. Accessed May 11, 2020. 2019. URL: https://doc.rust-lang.org/book/ (cited on page 110).

[150] Rust Secure Code Working Group. *Mission Statement of the Secure Code Working Group*. Accessed May 11, 2020. 2019. URL: https://github.com/rust-secure-code/wg (cited on pages 110, 127).

[151] Nicholas D. Matsakis. *Unsafe abstractions*. Accessed on May 5th, 2020; Matsakis is co-leader of the Rust compiler team. 2016. URL: http://smallcultfollowing.com/babysteps/blog/2016/05/23/unsafe-abstractions (cited on page 110).

[152] Fuchsia Team. *Fuchsia Documentation - Unsafe Code in Rust*. Accessed May 11, 2020. 2020. URL: https://fuchsia.googlesource.com/fuchsia/+/master/docs/development/languages/rust/unsafe.md (cited on page 110).

[153] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 'Safe systems programming in Rust'. In: *Commun. ACM* 64.4 (2021), pp. 144–152. DOI: 10.1145/3418295 (cited on page 110).

[154] Qrates Team. *Qrates artefact*. Source code and dataset: https://github.com/rust-corpus/qrates. 2020. DOI: 10.5281/zenodo.4085004 (cited on pages 111, 125).

[155] LLVM Team. *LLVM Atomic Instructions and Concurrency Guide*. Accessed May 11, 2020. 2020. URL: https://llvm.org/docs/Atomics.html (cited on page 115).

[156] Rust Team. *The Cargo Book*. Accessed May 11, 2020. 2019. URL: https://doc.rust-lang.org/cargo/reference/build-scripts.html#-sys-packages (cited on page 118).

[157] Rust Team. *Rust Documentation of Transmute*. Accessed September 10, 2020. 2020. URL: https://doc.rust-lang.org/std/mem/fn.transmute.html (cited on page 120).

[158] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 'Understanding memory and thread safety practices and issues in real-world Rust programs'. In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by Alastair F. Donaldson and Emina Torlak. ACM, 2020, pp. 763–779. DOI: 10.1145/3385412.3386036 (cited on pages 121, 144, 151, 253).

[159] Rust community. *Rust: The Reference*. Accessed May 11, 2020. 2018. URL: https://doc.rust-lang.org/reference/ (cited on page 122).

[160] Nicholas D. Matsakis. *Project idea: datalog output from rustc*. Accessed May 11, 2020. 2017. URL: https://smallcultfollowing.com/babysteps/blog/2017/02/17/project-idea-datalog-output-from-rustc/ (cited on page 123).

[161] Code QL. *Website of Code QL*. Accessed May 11, 2020. 2020. URL: https://semmle.com/codeql (cited on page 123).

[162] Pietro Albini. *The Rustwide library*. Accessed May 11, 2020. 2020. URL: https://crates.io/crates/rustwide (cited on page 123).

[163] Jupyter Team. *The Jupyter project*. Accessed May 11, 2020. 2020. URL: https://jupyter.org/ (cited on page 125).

[164] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 'Is Rust used safely by software developers?' In: *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. Ed. by Gregg Rothermel and Doo-Hwan Bae. ACM, 2020, pp. 246–257. DOI: 10.1145/3377811.3380413 (cited on pages 126, 143, 151).

[165] Rust Team. *File: check_unsafety.rs*. Accessed May 11, 2020. 2020. URL: https://github.com/rust-lang/rust/blob/27ae2f0d60d9201133e1f9ec7a04c05c8e55e665/src/librustc_mir/transform/check_unsafety.rs (cited on page 131).

[166] Compiler-builtins developers. *Safety of intrinsics*. Accessed September 7, 2020. 2020. URL: https://github.com/rust-lang/compiler-builtins/issues/355 (cited on page 134).

[167] Reinhard Wilhelm, Thomas W. Reps, and Shmuel Sagiv. 'Shape Analysis and Applications'. In: *The Compiler Design Handbook: Optimizations and Machine Code Generation*. Ed. by Y. N. Srikant and Priti Shankar. CRC Press, 2002, pp. 175–218. DOI: 10.1201/9781420040579.ch5 (cited on page 138).

[168] Alex Ozdemir. *Unsafe in Rust: Syntactic Patterns*. Accessed on May 5th, 2020. 2016. URL: https://cs.stanford.edu/~aozdemir/blog/unsafe-rust-syntax (cited on page 144).

[169] Kai Mindermann, Philipp Keck, and Stefan Wagner. 'How Usable Are Rust Cryptography APIs?' In: *2018 IEEE International Conference on Software Quality, Reliability and Security, QRS 2018, Lisbon, Portugal, July 16-20, 2018*. IEEE, 2018, pp. 143–154. DOI: 10.1109/QRS.2018.00028 (cited on page 144).

[170] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamaric, and Leonid Ryzhyk. 'System Programming in Rust: Beyond Safety'. In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*. Ed. by Alexandra Fedorova, Andrew Warfield, Ivan Beschastnikh, and Rachit Agarwal. ACM, 2017, pp. 156–161. DOI: 10.1145/3102980.3103006 (cited on page 144).

[171] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Alexander Levis. 'The Case for Writing a Kernel in Rust'. In: *Proceedings of the 8th Asia-Pacific Workshop on Systems, Mumbai, India, September 2, 2017*. ACM, 2017, 1:1–1:7. DOI: 10.1145/3124680.3124717 (cited on page 144).

[172] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 'Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale'. In: *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. Ed. by Robbert van Renesse and Nickolai Zeldovich. ACM, 2021, pp. 84–99. DOI: 10.1145/3477132.3483570 (cited on pages 144, 151, 159, 244, 248, 253).

[173] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R. Lyu. 'Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs'. In: *ACM Trans. Softw. Eng. Methodol.* 31.1 (2022), 3:1–3:25. DOI: 10.1145/3466642 (cited on pages 144, 151, 159).

[174] Xiaoye Zheng, Zhiyuan Wan, Yun Zhang, Rui Chang, and David Lo. 'A Closer Look at the Security Risks in the Rust Ecosystem'. In: *CoRR* abs/2308.15046 (2023). DOI: 10.48550/ARXIV.2308.15046 (cited on pages 144, 145, 151).

[175] Mohan Cui, Suran Sun, Hui Xu, and Yangfan Zhou. 'Is unsafe an Achilles' Heel? A Comprehensive Study of Safety Requirements in Unsafe Rust Programming'. In: *CoRR* abs/2308.04785 (2023). DOI: 10.48550/ARXIV.2308.04785 (cited on page 145).

[176] Rust Language Team. *Rust Survey 2019 Results*. Accessed May 15, 2020. 2019. URL: https://blog.rust-lang.org/2020/04/17/Rust-survey-2019.html (cited on page 147).

[177] The Rust Project Developers. *Struct Vec*. Accessed December 22, 2023. 2023. URL: https://doc.rust-lang.org/std/vec/struct.Vec.html (cited on page 156).

[178] The Rust Project Developers. *Function std::ptr::copy*. Accessed December 22, 2023. 2023. URL: https://doc.rust-lang.org/std/ptr/fn.copy.html (cited on page 156).

[179] The Rust Project Developers. *Function std::mem::swap*. Accessed December 22, 2023. 2023. URL: https://doc.rust-lang.org/std/mem/fn.swap.html (cited on page 160).

[180] Alexis Beingessner. *Pre-Pooping Your Pants With Rust*. Accessed March 30, 2023. 2015. URL: https://cglab.ca/~abeinges/blah/everyone-poops/ (cited on page 160).

[181] The Rust Project Developers. *Function std::mem::forget*. Accessed December 22, 2023. 2023. URL: https://doc.rust-lang.org/std/mem/fn.forget.html (cited on pages 160, 247).

[182] The Rust Project Developers. *Method Vec::dedup_by*. Accessed December 22, 2023. 2023. URL: https://doc.rust-lang.org/std/vec/struct.Vec.html#method.dedup_by (cited on pages 160, 161).

[183] Jonas Maier. 'Towards Verifying Real-World Rust Programs'. BA thesis. ETH Zurich, 2022 (cited on page 175).

[184] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. *The CompCert Memory Model, Version 2*. Research Report RR-7987. INRIA, June 2012, p. 26 (cited on page 185).

[185] Olivia Furrer. 'Verifying Vulnerability Fixes in a Rust Verifier'. BA thesis. ETH Zurich, 2023 (cited on pages 185, 244, 248, 260).

[186] David Detlefs, Greg Nelson, and James B. Saxe. 'Simplify: a theorem prover for program checking'. In: *J. ACM* 52.3 (2005), pp. 365–473. DOI: 10.1145/1066100.1066102 (cited on page 188).

[187] K. Rustan M. Leino and Wolfram Schulte. 'Exception Safety for C#'. In: *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), 28-30 September 2004, Beijing, China*. IEEE Computer Society, 2004, pp. 218–227. DOI: 10.1109/SEFM.2004.14 (cited on pages 193, 255).

[188] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. *JML Reference Manual*. http://www.jmlspecs.org/. 2011 (cited on page 193).

[189] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. 'Modular invariants for layered object structures'. In: *Sci. Comput. Program.* 62.3 (2006), pp. 253–286. DOI: 10.1016/J.SCICO.2006.03.001 (cited on page 194).

[190] Pascal Huber. 'Automatically Generating Memory Safety Certificates for Rust Programs'. MA thesis. ETH Zurich, 2022 (cited on pages 217, 227).

[191] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 'Iris from the ground up: A modular foundation for higher-order concurrent separation logic'. In: *J. Funct. Program.* 28 (2018), e20. DOI: 10.1017/S0956796818000151 (cited on page 218).

[192] The Rust Project Developers. *Implementation of function std::mem::replace*. Accessed January 19, 2024. 2024. URL: https://doc.rust-lang.org/src/core/mem/mod.rs.html#912-925 (cited on page 246).

[193] Rust for Linux Project Developers. *Rust for Linux Vec::try_with_capacity*. Accessed January 15, 2024. 2023. URL: https://github.com/Rust-for-Linux/linux/blob/18b7491480025420896e0c8b73c98475c3806c6f/rust/alloc/vec/mod.rs#L532-L536 (cited on page 247).

[194] The Rust Project Developers. *Struct core::mem::ManuallyDrop*. Accessed January 21, 2024. 2023. URL: https://doc.rust-lang.org/core/mem/struct.ManuallyDrop.html (cited on page 247).

[195] Ralf Jung. *Comment on Rust issue #60847*. Accessed January 21, 2024. 2021. URL: https://github.com/rust-lang/rust/issues/60847#issuecomment-847629613 (cited on page 249).

[196] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. 'Verifying Dynamic Trait Objects in Rust'. In: *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022*. IEEE, 2022, pp. 321–330. DOI: 10.1109/ICSE-SEIP55303.2022.9794041 (cited on page 253).

[197] Shuanglong Kan, Zhe Chen, David Sanán, Shang-Wei Lin, and Yang Liu. 'An Executable Operational Semantics for Rust with the Formalization of Ownership and Borrowing'. In: *CoRR* abs/1804.07608 (2020) (cited on page 253).

[198] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 'SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-Flow Analysis'. In: *CoRR* abs/2103.15420 (2021) (cited on page 253).

[199] Baojian Hua, Wanrong Ouyang, Chengman Jiang, Qiliang Fan, and Zhizhong Pan. 'Rupair: Towards Automatic Buffer Overflow Detection and Rectification for Rust'. In: *ACSAC '21: Annual Computer Security Applications Conference, Virtual Event, USA, December 6 - 10, 2021*. ACM, 2021, pp. 812–823. DOI: 10.1145/3485832.3485841 (cited on page 253).

[200] Herman Venter. *MIRAI: Rust mid-level IR Abstract Interpreter*. Accessed March 30, 2023. 2023. URL: https://github.com/facebookexperimental/MIRAI (cited on page 253).

[201] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C. S. Lui. 'MirChecker: Detecting Bugs in Rust Programs via Static Analysis'. In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi. ACM, 2021, pp. 2183–2196. DOI: 10.1145/3460120.3484541 (cited on page 253).

[202] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C. S. Lui. 'Detecting Cross-language Memory Management Issues in Rust'. In: *Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26-30, 2022, Proceedings, Part III*. Ed. by Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng. Vol. 13556. Lecture Notes in Computer Science. Springer, 2022, pp. 680–700. DOI: 10.1007/978-3-031-17143-7_33 (cited on page 253).

[203] Nicholas Nethercote and Julian Seward. 'Valgrind: a framework for heavyweight dynamic binary instrumentation'. In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. Ed. by Jeanne Ferrante and Kathryn S. McKinley. ACM, 2007, pp. 89–100. DOI: 10.1145/1250734.1250746 (cited on page 253).

[204] Miri Team. *Miri: An interpreter for Rust's mid-level intermediate representation*. Accessed March 30, 2023. 2023. URL: https://github.com/rust-lang/miri/ (cited on page 253).

[205] David R. Cok and K. Rustan M. Leino. 'Specifying the Boundary Between Unverified and Verified Code'. In: *The Logic of Software. A Tasting Menu of Formal Methods - Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*. Ed. by Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, and Einar Broch Johnsen. Vol. 13360. Lecture Notes in Computer Science. Springer, 2022, pp. 105–128. DOI: 10.1007/978-3-031-08166-8_6 (cited on pages 254, 255).

[206] Elijah Rivera, Samuel Mergendahl, Howard E. Shrobe, Hamed Okhravi, and Nathan Burow. 'Keeping Safe Rust Safe with Galeed'. In: *ACSAC '21: Annual Computer Security Applications Conference, Virtual Event, USA, December 6 - 10, 2021*. ACM, 2021, pp. 824–836. DOI: 10.1145/3485832.3485903 (cited on page 254).

[207] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 'Sandcrust: Automatic Sandboxing of Unsafe Components in Rust'. In: *Proceedings of the 9th Workshop on Programming Languages and Operating Systems, Shanghai, China, October 28, 2017*. Ed. by Julia Lawall. ACM, 2017, pp. 51–57. DOI: 10.1145/3144555.3144562 (cited on page 254).

[208] Peiming Liu, Gang Zhao, and Jeff Huang. 'Securing unsafe Rust programs with XRust'. In: *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. Ed. by Gregg Rothermel and Doo-Hwan Bae. ACM, 2020, pp. 234–245. DOI: 10.1145/3377811.3380325 (cited on page 254).

[209] Pieter Agten, Bart Jacobs, and Frank Piessens. 'Sound Modular Verification of C Code Executing in an Unverified Context'. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 581–594. DOI: 10.1145/2676726.2676972 (cited on page 254).

[210] David Abrahams. 'Exception-Safety in Generic Components'. In: *Generic Programming, International Seminar on Generic Programming, Dagstuhl Castle, Germany, April 27 - May 1, 1998, Selected Papers*. Ed. by Mehdi Jazayeri, Rüdiger Loos, and David R. Musser. Vol. 1766. Lecture Notes in Computer Science. Springer, 1998, pp. 69–79. DOI: 10.1007/3-540-39953-4_6 (cited on page 254).

[211] Maroua Maalej, Tucker Taft, and Yannick Moy. 'Safe Dynamic Memory Management in Ada and SPARK'. 2018 (cited on page 260).

[212] Philippe Voinov. 'Optimisation of a Deductive Program Verifier'. BA thesis. ETH Zurich, 2019 (cited on page 281).

[213] Robin Sierra. 'Towards Customizability of a Symbolic-Execution-Based Program Verifier'. BA thesis. ETH Zurich, 2017 (cited on page 284).

[214] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 'CN: Verifying Systems C Code with Separation-Logic Refinement Types'. In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 1–32. DOI: 10.1145/3571194 (cited on page 284).

[215] alias. *Stack Overflow Question: Is branching necessary when proving a sum of non-negative terms?* Accessed January 21, 2024. 2023. URL: https://stackoverflow.com/a/76418255/2591676 (cited on page 285).

Appendix

Understanding Performance of Viper's Separated Iterated Conjunction Algorithm

A

This chapter presents our investigation of the Viper's performance problem, which impeded our evaluation of Prusti presented in Chapter 22. From the user's perspective, the problem could be described as follows: after making a small change to a Rust function, which uses a predicate encoded into an iterated separating conjunction, Viper[1] seemingly does not terminate anymore. Typical reasons why a SMT-based verifier does not terminate are non-linear arithmetic and infinite quantifier instantiations. We quickly ruled out non-linear arithmetic because we were able to reproduce the problem with non-linear arithmetic solver disabled. Determining whether quantifiers could be the root cause of non-termination was significantly harder because in Z3[2] traces we could see a huge number of quantifier instantiations, but we could not find so-called matching loop: a set of quantifiers that cause an infinite chain of instantiations by causing instantiations of each other. By analysing the trace, we noticed that many of these instantiations are caused by quantifiers being instantiated with the same ground terms over and over again. To the best of our knowledge, a quantifier can be instantiated with the same ground terms multiple times only if the SMT solver is resetting its state. This observation helped us to notice that Z3 traces for Viper programs with iterated separating conjunction contains many `DecideAndOr` events that indicate that the solver is doing a case split. Case splits explain why quantifiers are instantiated multiple times with the same ground terms: after exploring a case, Z3 rolls back all changes made to the state while exploring that case, which also includes forgetting all quantifier instantiations. When comparing Prusti-generated Viper programs with iterated separating conjunctions and without them, we noticed that `DecideAndOr` events are extremely common in the former and almost never occur in the latter. Therefore, we hypothesise that the performance problem is caused by exhaustive branching and we can prevent or at least mitigate non-termination by addressing the reasons that cause the branching.

1: Version `v-2023-08-26-2125`.

2: The SMT solver used by Viper.

We pursued the goal of finding the root cause for branching by trying to find a minimal Viper example that demonstrates the branching behaviour. Figure A.1 shows a Viper example with `n` inhales of predicate `P` followed by `n` exhales. For a human, it is obvious that this example should always trivially verify because we are exhaling the same predicates we just inhaled. However, if we force the symbolic execution backend to use the algorithm for iterated separating conjunctions by adding line 20, we can see in Figure A.2 that the verification time (averaged over 5 runs) is exponential with respect to `n` showing that the SMT solver is doing significant thinking[3]. From the same graph, we can see that the default algorithm, called *greedy*, scales much better. A very similar example was discovered by Philippe Voinov in his bachelor thesis [212], in which he investigated the slow performance of Viper's verification condition backend. For Prusti, we always used the symbolic execution backend because the verification condition backend was too slow even for very small examples. Philippe, in his example, used unfolding instead of exhale, but his and our investigation shows that the problem comes from

3: All measurements in this chapter were done on Lenovo T470p laptop with Intel(R) Core(TM) i7-7700HQ CPU with 4 cores (8 threads), 16 GB of RAM, and Ubuntu 20.04.6, symbolic execution backend version `151f2218`.

[212]: Voinov (2019), 'Optimisation of a Deductive Program Verifier'

```
1  predicate P(a: Address)
2
3  method test() {
4      var a01: Address
5      var a02: Address
6      // ...
7      var a_n: Address
8
9      inhale acc(P(a01), write)
10     inhale acc(P(a02), write)
11     // ...
12     inhale acc(P(a_n), write)
13
14     exhale acc(P(a01), write)
15     exhale acc(P(a02), write)
16     // ...
17     exhale acc(P(a_n), write)
18
19     assume false
20     inhale forall a: Address :: {P(a)} false ==> acc(P(a))
21 }
```

Figure A.1: A simple example of `n` inhales followed by `n` exhales demonstrating the performance problem in Viper. Line 20 tricks the backend into using the algorithm for iterated separating conjunctions.

consuming a predicate instance, which can be done either by exhaling or unfolding it. Philippe explained why verification condition backend is slow on his example on the level of encoding. We believe that the problem can be understood on a higher level as a fundamental trade-off between completeness and performance.

To understand the problem, we need to look into what a verifier does when it consumes a resource. For the first exhale of predicate `P` on line 14, the operations performed by the verifier are straightforward. The verifier first checks that it has at least `write` permissions to predicate `P(a01)`. This check corresponds conceptually to checking the assertion shown in the following snippet that sums up the permissions of predicate instances inhaled for location `a01` (remember that predicate instance `P(a01)` is stored not at syntactic place `a01`, but at the value of `a01`).

```
assert write <= (a01 == a01 ? write : none) +
                (a01 == a02 ? write : none) +
                /* ... */
                (a01 == a_n ? write : none)
```

In this case, the assertion can be easily proven because we have a clear match (`a01 == a01`) and all other summands are non-negative. After performing the check, the verifier remembers that it removed `write` permission amount from predicate `P` at address `a01`. For the second exhale, the verifier performs the same two operations, but the sufficient permission check is more interesting because it needs to take into account the exhale for address `a01`. The assert that corresponds conceptually to

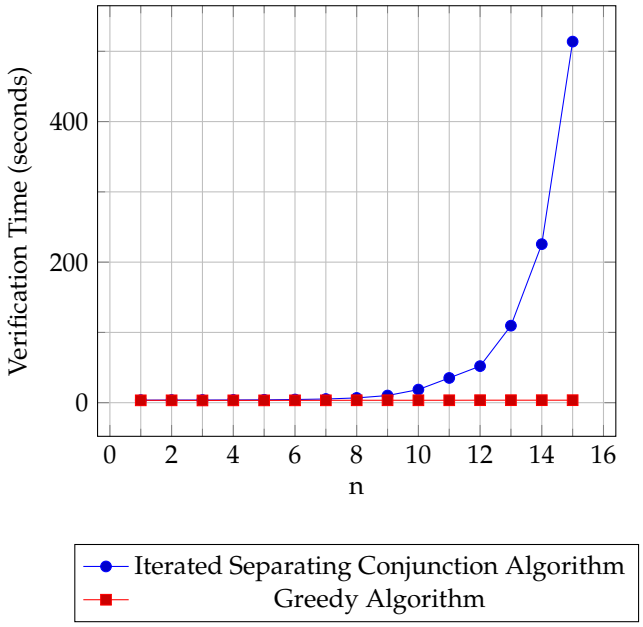


Figure A.2: Verification time of the example from Figure A.1 for different `n` using Viper symbolic execution backend.

this permission check is shown in the following snippet.

```
assert write <= (a02 == a01 ? write : none) +
                (a02 == a02 ? write : none) +
                /* ... */
                (a02 == a_n ? write : none) +
                // exhales:
                (a02 == a01 ? -write : none)
```

In this assertion, we have a negative summand that could potentially make the sum smaller than `write`. Therefore, the SMT solver needs to consider two cases: when equality `a02 == a01` holds and when it does not hold (we will discuss later whether this case split is really necessary). If we write the permission check for the `n`th exhale, we can see that the solver needs to do `n-1` case splits.

```
assert write <= (a_n == a01 ? write : none) +
                (a_n == a02 ? write : none) +
                /* ... */
                (a_n == a_n ? write : none) +
                // exhales:
                (a_n == a01 ? -write : none) +
                (a_n == a02 ? -write : none)
                /* ... */
                (a_n == a_(n-1) ? -write : none)
```

`n-1` case splits generate 2^{n-1} combinations, which can easily explain the exponential slow-down shown in Figure A.2.

In Figure A.2, we have also shown that the greedy algorithm scales much better than iterated separating conjunctions one, which raises a natural question: why is this the case? If we consider the last permission check again, we can see that we can group summands by addresses as shown

in the following snippet that ensures that the value of each group is non-negative.

```
assert write <= // a01 group
                ((a_n == a01 ? write : none) +
                    (a_n == a01 ? -write : none)) +
                // a02 group
                ((a_n == a02 ? write : none) +
                    (a_n == a02 ? -write : none)) +
                /* ... */
                // a_(n-1) group
                ((a_n == a_(n-1) ? -write : none) +
                    (a_n == a_(n-1) ? -write : none)) +
                // a_n group
                (a_n == a_n ? write : none)
```

Since `a_n == a_n` holds syntactically, we know that the total sum is the sum of `write` plus some summands that are guaranteed to be non-negative, which means that the total sum is guaranteed to be at least `write`. Grouping by addresses is exploited by the greedy algorithm to achieve performance. Instead of trying to compute the sum, when exhaling a predicate instance, the greedy algorithm tries to find the group that has provably the same address and subtracts the exhaled permission amount from that group. For instance, for exhale of predicate instance `P(a01)` in our example, the greedy algorithm would consider to take permissions only from inhale of predicate instance `P(a01)` because `a01` is the only aliasing relation that the SMT solver can prove. This greedy approach gives better performance at the cost of completeness. One example that is not supported by the greedy approach is, so called, disjunctive aliasing shown in the following snippet.

```
inhale acc(P(x), write) && acc(P(y), write)
assume z == x  z == y
exhale acc(P(z), write)
```

The exhale in this snippet should clearly verify, but the greedy algorithm rejects it because it cannot find a group from which it could *always* take the necessary permission. This example, however, verifies if we use a verification condition generation backend or symbolic execution backend with iterated separating conjunctions algorithm. In the spectrum between the greedy algorithm, which is fast and incomplete, and iterated separating conjunction algorithm, which is significantly more complete, but slow, there are also other algorithms that provide different trade-offs between completeness and performance. Important examples are an algorithm dubbed "more complete exhale" [213], which does not support iterated separating conjunctions, but can verify disjunctive aliasing, and an algorithm presented in [214] that could probably be best summarised as a greedy algorithm for iterated separating conjunctions. The algorithm from [214] supports taking permission to a single element from an iterated separating conjunction and returning it back, but does not support splitting or joining iterated separating conjunctions, which we saw are necessary for verifying unsafe Rust code. It is important to note that symbolic execution backend uses many heuristics to improve the performance of the iterated separating conjunctions algorithm, which make it to typically perform better than on our artificial example, but also

[213]: Sierra (2017), 'Towards Customizability of a Symbolic-Execution-Based Program Verifier'

[214]: Pulte et al. (2023), 'CN: Verifying Systems C Code with Separation-Logic Refinement Types'

[214]: Pulte et al. (2023), 'CN: Verifying Systems C Code with Separation-Logic Refinement Types'

making the performance less predictable, which could be one explanation why on Prusti examples we observe that Viper after a small change does not terminate anymore.

To the best of our knowledge, the researchers up to now focused on algorithms that treat all resources uniformly as fractional ones and use SMT solvers as opaque components. We think that there are at least three research directions worth pursuing in the future work. One research direction could be to give more control to the users over how resources are managed. Sometimes the frontend generating the Viper encoding knows that predicate instance `P(a)` is guaranteed to be disjoint from all predicate instances mentioned in iterated separating conjunctions. For example, in Prusti we know that `MemoryBlock` of a local variable to which no pointer was created cannot be included into `raw_range!(...)` predicate. Such a predicate instance could be explicitly specified as managed by the greedy algorithm reducing the number of predicate instances managed by the iterated separating conjunction, which also should improve its performance. A variation of this approach we use in the evaluation presented in Chapter 22 to mitigate the performance problem. Another research direction could be designing algorithms that take advantage of special properties of the resources. For example, in Prusti, for some resources fractional permissions are not needed and, therefore, permissions to them could be modelled as boolean values, for which SMT solvers have better decision procedures. A variation of this research direction would be investigating whether there are theories with faster decision procedures that could be used to encode fractional permissions; a promising one, suggested on Stack Overflow [215] is pseudo-booleans that supports reasoning about expressions that look very similar to proof obligations that we need to check when consuming resources. The last research direction would be to to explore whether APIs provided by Z3 could be used to guide the proof search to perform the case split only when it is strictly necessary. Unfortunately, we did not have time to properly explore any of these research directions and leave them to future work.

[215]: alias (2023), *Stack Overflow Question: Is branching necessary when proving a sum of non-negative terms?*