Federico Poli

Enabling Rich Lightweight Verification of Rust Software

DISS. ETH NO. 30273 | YEAR 2024

DISS. ETH NO. 30273

Enabling Rich Lightweight Verification of Rust Software

A thesis submitted to attain the degree of

Doctor of Sciences

(Dr. sc. ETH Zurich)

Presented by

Federico Poli

M.Sc. in Computer Science, University of Padova

Born on 04.11.1992

Accepted on the recommendation of

Prof. Dr. Peter Müller, examiner
Prof. Dr. Alexander J. Summers, co-examiner
Prof. Dr. Ralf Jung, co-examiner
Prof. Dr. Viktor Kunčak, co-examiner

2024

Enabling Rich Lightweight Verification of Rust Software.

Abstract

Rust is a novel programming language that is rapidly gaining adoption in the software industry thanks to its performance and safety properties. Among its strengths, Rust's type system is designed to detect at compile time certain kinds of memory-related bugs, such as dangling pointers and null dereferences. However, the language does not aim to detect functional errors; how to do this efficiently is in general an open topic in the field of software verification.

This thesis explores the link between Rust and software verification, showing how the static properties of Rust can be leveraged to develop new, simpler, verification techniques. In particular, we identify formalization, specification and automation challenges that block user-friendly, scalable, deductive verification. To address these challenges we develop two novel verification techniques for Rust that work on different subsets of the language.

Our first new technique focuses on verification of programs that are entirely written in *safe Rust*: the portion of the Rust language in which the compiler takes care of guaranteeing absence of memory-related bugs. For this setting, we define a model of the properties of Rust types in the implicit dynamic frames logic — a variant of separation logic — using the Viper language. We then develop a new static analysis that tracks the capabilities of Rust types (e. g., the ability to read or write a certain memory location), and we use its results in an algorithm that constructs automatically a *core proof* of the program, which encodes the immutability and aliasing guarantees of Rust types. This proof is expressed in the implicit dynamic frames logic, but our technique hides this as an implementation detail, effectively shielding the verification users from the complex details of the logic. To verify the functional correctness of a program, users only have to specify the functional behavior of its functions, writing contracts such as preconditions and postconditions. To this end, we offer program annotations with syntax and semantics identical to those of boolean Rust expressions, with some restrictions regarding side-effects and determinism that we describe with a definition of *purity*. Some aspects of Rust require special annotations. For example, to describe formally the relation between returned references and the function arguments that they block according to type system rules, we introduce a new program annotation called a *pledge*. We implement our technique in an open-source verification tool called *Prusti*, which we use to evaluate our technique on several collections of programs. Overall, by leveraging Rust types to construct the core proof automatically, the annotations required by our technique are drastically simpler, and shorter, than those of verification techniques for other programming languages.

When a Rust program is not entirely safe, but also contains *unsafe* code, the strong properties of Rust types are weakened by some exceptions. For example, libraries that are implemented with unsafe code can expose *interior mutability*, meaning that they can mutate values behind shared references that, in a fully-safe program, would be immutable. Naively applying the verification techniques for fully-safe code to this setting would be unsound, because a verifier would incorrectly assume that interior mutability cannot cause mutations. To handle these cases, we develop a second new verification technique to reason about *safe clients*: safe programs that use trusted libraries implemented with unsafe code. These clients pose verification challenges due to the

ability to observe interior mutability. To address this, we base our technique on a new notion of *implicit* library capabilities: properties of a Rust type, such as immutability or non-aliasing, that are not defined by the Rust language as in the case of references, but are determined by the developers of the Rust library. We introduce new library annotations to declare these capabilities explicitly, and we develop a verification technique that uses the capabilities to verify the functional correctness of safe clients based on an encoding into first-order logic. Our technique supports an expressive variety of capabilities: *core* capabilities correspond to the properties of regular Rust references in safe code, while *extended* capabilities go one step beyond and enable expressing new properties that do not have an equivalent in safe code. Each of these capabilities has particular properties, including unique access to a memory location, write access, immutability, and absence of concurrent usages. For our core capabilities, we prove correctness of their properties by developing a novel proof technique. This derives the properties of core capabilities from those of Rust references, side-stepping some problems posed by the current absence of formal semantics for unsafe code. We implement our technique in an open-source verifier called *Mendel*, which we use to verify functional properties of sequences of API calls to popular Rust libraries, including `Rc`, `Arc`, `Cell`, `RefCell`, `AtomicI32`, `Mutex`, and `RwLock`. Our results show that our technique is expressive enough to specify the functional behavior of real-world API methods, while requiring little annotation on the client side of their libraries.

Overall, the verification techniques that we develop are *lightweight* because they require few user annotations, while still being *rich* enough to verify expressive user-chosen properties. The verification tools that we build around them are designed to offer a particularly curated user experience, thanks to some usability choices and solutions that we also describe.

Sommario

Rust è un linguaggio di programmazione innovativo che sta rapidamente guadagnando terreno nell'industria del software grazie alle sue prestazioni e proprietà di sicurezza. Tra i suoi punti di forza, il sistema di tipi di Rust è progettato per rilevare durante la compilazione certi tipi di errori di memoria, come puntatori non validi e dereferenziazioni di puntatori nulli. Tuttavia, il linguaggio non mira a rilevare errori funzionali; come farlo in modo efficiente è in generale un problema aperto nell'ambito della verifica del software.

Questa tesi esplora il legame tra Rust e la verifica del software, mostrando come le proprietà statiche di Rust possano essere sfruttate per sviluppare nuove e più semplici tecniche di verifica. In particolare, identifichiamo sfide di formalizzazione, specifica, e automazione che ostacolano la verifica deduttiva, scalabile, e di facile utilizzo del software. Risolviamo questi ostacoli sviluppando due innovative tecniche di verifica per Rust che operano su diversi sottoinsiemi del linguaggio.

La nostra prima nuova tecnica si focalizza sulla verifica di programmi interamente scritti in *Rust sicuro*: la porzione del linguaggio Rust per cui il compilatore si occupa di garantire l'assenza di errori di memoria. Per questo contesto, definiamo un modello delle proprietà dei tipi di Rust nella logica dei frame dinamici impliciti — una variante della logica delle separazioni — usando il linguaggio Viper. Successivamente, sviluppiamo una nuova analisi statica che tiene traccia delle capacità dei tipi di Rust (ad esempio, la capacità di leggere o scrivere in una determinata posizione della memoria) e utilizziamo i suoi risultati in un algoritmo che costruisce automaticamente una *dimostrazione base* (*"core proof"*) del programma, la quale codifica le garanzie di immutabilità e assenza di aliasing dei tipi di Rust. Questa dimostrazione è espressa nella logica dei frame dinamici impliciti, ma la nostra tecnica nasconde ciò come un dettaglio implementativo, effettivamente difendendo gli utenti della verifica dai dettagli complessi della logica. Per dimostrare la correttezza funzionale di un programma, agli utenti rimane solo di specificare il comportamento delle proprie funzioni, scrivendo contratti come ad esempio precondizioni e postcondizioni. Per far ciò, offriamo annotazioni la cui sintassi e semantica sono identiche a quelle delle espressioni booleane di Rust, con alcune restrizioni riguardanti gli effetti collaterali e il determinismo che descriviamo con una definizione di *purezza*. Alcuni aspetti di Rust richiedono annotazioni speciali. Per esempio, per descrivere formalmente la relazione tra i riferimenti ritornati da una funzione e gli argomenti che blocca secondo le regole del sistema di tipi, introduciamo una nuova annotazione chiamata *pledge* (*"impegno"*). Implementiamo la nostra tecnica in uno strumento open-source di verifica chiamato *Prusti*, che utilizziamo per valutare la nostra tecnica su varie raccolte di programmi. Complessivamente, sfruttando i tipi di Rust per costruire automaticamente la dimostrazione base, le annotazioni richieste dalla nostra tecnica sono drasticamente più semplici e più brevi rispetto a quelle delle tecniche di verifica per altri linguaggi di programmazione.

Quando un programma scritto in Rust non è interamente sicuro, ma contiene anche una parte di codice *insicuro*, le forti proprietà dei tipi di Rust sono indebolite da alcune eccezioni. Per esempio, le librerie che sono implementate con codice insicuro possono esporre *mutabilità interiore*, il che significa che possono modificare i valori

raggiungibili dai riferimenti condivisi, che in un programma completamente sicuro sarebbero immutabili. In questo contesto sarebbe sbagliato applicare con leggerezza le tecniche di verifica per codice completamente sicuro, perché un verificatore arriverebbe ad assumere incorrettamente che la mutabilità interiore non può causare modifiche. Per gestire questi casi, sviluppiamo una seconda nuova tecnica per ragionare sui *client sicuri* (“*safe clients*”): programmi sicuri che usano librerie fidate implementate con codice insicuro. Questi client presentano sfide di verifica a causa della loro capacità di osservare mutabilità interna. Per risolvere questo problema, fondiamo la nostra tecnica su una nuova nozione di capacità *implicita* di libreria: proprietà di un tipo di Rust (come ad esempio l'immutabilità o l'assenza di aliasing) che non sono definite dal linguaggio Rust come nel caso dei riferimenti, ma che sono determinate dagli sviluppatori delle librerie Rust. Introduciamo nuove annotazioni di libreria per dichiarare queste capacità in modo esplicito, e sviluppiamo una tecnica di verifica che utilizza le capacità per verificare la correttezza funzionale di client sicuri basandosi su una codifica nella logica del primo ordine. La nostra tecnica supporta una espressiva varietà di capacità: le capacità *base* corrispondono alle proprietà di regolari riferimenti Rust in codice sicuro, mentre le capacità *estese* vanno un passo oltre e abilitano l'espressione di nuove proprietà che non hanno un equivalente nel codice sicuro. Ciascuna di queste capacità ha particolari proprietà, tra le quali l'accesso esclusivo ad una posizione della memoria, l'accesso in scrittura, l'immutabilità, e l'assenza di utilizzi concorrenti. Per le nostre capacità di base, dimostriamo la correttezza delle loro proprietà sviluppando una nuova tecnica di dimostrazione. Questa deduce le proprietà delle capacità di base a partire da quelle dei riferimenti di Rust, evitando alcuni problemi posti dall'attuale assenza di una semantica formale per il codice insicuro. Implementiamo la nostra tecnica in un verificatore open-source chiamato *Mendel*, che utilizziamo per verificare proprietà funzionali di sequenze di chiamate all'API di popolari librerie Rust, tra cui `Rc`, `Arc`, `Cell`, `RefCell`, `AtomicI32`, `Mutex`, and `RwLock`. I nostri risultati mostrano che la nostra tecnica è espressiva al punto da specificare il comportamento di metodi di API reali, richiedendo al tempo stesso poche annotazioni dal lato client delle librerie.

Nel complesso, le tecniche di verifica che sviluppiamo sono *leggere* perché richiedono poche annotazioni da parte dell'utente, pur essendo comunque *ricche* abbastanza da verificare espressive proprietà scelte dall'utente. Gli strumenti di verifica che costruiamo attorno ad esse sono progettati per offrire un'esperienza utente particolarmente curata, grazie anche ad alcune scelte di usabilità e soluzioni che descriviamo.

Acknowledgments

I would like to thank everyone who has been a part of the journey towards my PhD, whether mentioned here or not. You have made this an unforgettable experience. Thank you all!

First and foremost, thanks to my supervisor, Peter Müller, for offering me the opportunity to be part of his research group, for the invaluable advice, and for encouraging me to always aim higher (except in matters of fruit-based pizza toppings). Thanks also to my co-supervisor, Alexander Summers, for the countless technical discussions and friendly guidance, even when I dipped black bread into my English black tea.

Thanks to Vytautas Astrauskas for sharing an unusually sunny office with me all these years. I am grateful for the deep discussions from which I learned so much, and I am glad we survived the extreme conditions in which plants and USB drives perished. My son still enjoys our office's LEGO train and will surely have a fond memory of Scala.

Thanks to the members of the Prusti team: Aurel Bílý, Jonás Fiala, and Christoph Matheja. They joined during a dark time for the codebase, but with their help, we re-emerged stronger than before.

Thanks to the members of the awesome PM research group: Linard Arquint, Lea Brugger, Lucas Brutschy, Alexandra Bugariu, Martin Clochard, Thibault Dardinier, Xavier Denis, Jérôme Dohrau, Marco Eilers, Nicolas Klose, Anqi Li, Gaurav Parthasarathy, Fábio Pakk Selmi-Dei, João Carlos Mendes Pereira, Malte Schwerhoff, Michael Sammler, Dionisios Spiliopoulos, Arshavir Ter-Gabrielyan, Caterina Urban, and Felix Wolf, as well as the colleagues mentioned above. You have been both fantastic colleagues and friends during these years. Thanks also to the members of the (equally awesome) PLF research group: Isaac van Bakel, Johannes Hostert, Rudy Peterson, and Max Vistrup. Our group lunches have been particularly cheerful (and packed) since you joined. Thanks to Marlies Weissert and Sandra Schneider for always being available to assist us and, importantly, for making sure that we all went on vacation.

Thanks to my co-examiners, Ralf Jung and Viktor Kunčak, for taking the time to review my thesis and for their thorough and valuable feedback. Thanks to Andrea Lattuada and Niko Matsakis for the many engaging discussions about Rust. Your contributions have been greatly beneficial to my work.

During my PhD, I had the pleasure of supervising many talented students and interns, whose work contributed to this thesis in different ways: David Blaser, Jakob Beckmann, Dominic Dietler, Julian Dunskus, Lowis Engel, Matthias Erdin, Lorenz Gorse, Thomas Hader, Constantin Müller, Janis Peyer, Johannes Schilling, William Seddon, Nicolas Winkler, and Dylan Wolff. Thank you for your hard work and dedication; it has been a joy to work with each of you.

Thanks to Rajeev Joshi, Ernie Cohen, Bernhard Kragl, Rustan Leino, and the rest of the Dafny team for the wonderful experience during my internship at Amazon. Your guidance and support were crucial in advancing my research, which led to the development of the Mendel verifier.

During my internship at Google, I had the pleasure of working with Stefano Maggiolo, Natalie Doduc, and the rest of Android Location ML team. Thank you for the fantastic learning experience! I am still amazed by how much work goes into the devices we use every day.

Before my PhD, I met many friends who played an essential role in both my personal and professional development. Thanks to my friends from Brescia — Gabriele Farina, Lorenzo Moreschini, Giovanni Paolini, and Stefano Tisi — for sharing my passion for math and computer science. Thanks to my friends from Padova — the *Steakholders* and the Galileiana school — for all the fun we had while learning. Thanks to my ex-colleagues at Engineering and CERN for showing me how to work hard in a sustainable way.

Finally, but above all, I would like to thank my wife for her unwavering support throughout these years and for always bringing out the best in me. Thanks to my son for his infinite enthusiasm and hard work in typing part of this thesis. Thanks to my parents for feeding me with milk and bytes, and to my sisters for cheering me on from afar.

Contents

1 Introduction

As the importance and complexity of software increase, developers are increasingly pressured to find innovative solutions to ensure its reliability and robustness. One approach used in the design of programming languages consists of introducing language restrictions that make it more difficult, or entirely impossible, to implement certain kinds of bugs. In fact, there is a positive correlation between the static guarantees of a programming language and the ease of reasoning about it, with a progression from Assembly, through C, to Java. Compared to Assembly, C has a structured control flow, type declarations, and other abstractions that make it much easier to understand and check the intent of the program. This way, jumping to the wrong memory address or misinterpreting data bytes is much less common, although not entirely impossible. Java goes one step further, ensuring memory and type safety thanks to its type and runtime checks. Still, the language does not protect developers from unintended data races, aliasing or mutations, as well as uncaught exceptions such as null-pointer exceptions. In functional languages such as Haskell, in-place mutations are not possible, and all data structures are immutable. This effectively solves aliasing issues, but it also requires a significant mental shift for developers used to imperative languages.

Rust is a recent imperative programming language designed for speed, safety, and concurrency, which provides particularly strong static guarantees while remaining accessible to developers. Rust solves and mitigates many of Java's challenges by offering a *safe* language subset[1] in which unintended data races, aliasing, mutations, and memory errors are caught at compile time thanks to advanced type checks. For example, Rust makes sure that any mutable reference can be used only as long as the memory location that it points to is allocated, initialized, and not reachable via other references. Among Rust's types, *immutable* types guarantee absence of mutations — even from other threads — in a way that cannot be "cast away" like the `const` annotations in C++. Moreover, its *unique* types guarantee that certain memory locations can be reached only via one specific reference, preventing unintended aliasing. These benefits are not just theoretical. It is not a coincidence that the Rust language was found for the eighth year in a row to be the most admired language in Stack Overflow's developer survey [1].

1: The remaining subset is called *unsafe*.

In our work, we observe that many concepts of Rust's type system are not new and have similarities with *pointer capabilities* [2] and *separation logic* [3], a powerful logic that was developed in the formal verification field to ease the construction of memory-safety and correctness proofs regarding programs manipulating heap-allocated memory. For example, the memory-safety guarantees of Rust are based on type checks whose mechanism is similar to the role that permissions and, in general, resources have in separation logic. Moreover, Rust's immutable types resemble separation-logic resources with fractional permissions [4], which can be shared but cannot be used for modifications. Unique types, instead, resemble resources with full permissions, which can be used for modifications but cannot be duplicated. These similarities that we observe suggest that there might be a way of bringing the usability

[1]: Overflow (n.d.), *Stack Overflow Developer Survey 2023*

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

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

[4]: Boyland (2003), *Checking Interference with Fractional Permissions*

advantages of Rust to separation logic, and the verification advantages of separation logic to Rust. In fact, despite the static guarantees of existing programming languages, in situations where correctness is critical, software development still needs to be accompanied by formal techniques in order to guarantee absence of bugs with respect to a certain specification of its correctness. Among the verification approaches, *deductive verification* is one of the best suited for the verification of non-trivial codebases. The approach consists of expressing the correctness of a program as a set of mathematical statements, which are typically proved either manually in a theorem prover, or with the aid of automated solvers. However, the main downside of existing deductive verification techniques, such as those based on separation logic, is that they are usually expensive to use: they require a large amount of manual effort, as well as expert knowledge of the logic used in the proof.

Our Goal This thesis claims that Rust types follow a capability system that can be used to make program reasoning easier for formal verification techniques. We demonstrate so by developing novel deductive verification techniques, and functioning verification tools, for heap-manipulating Rust programs that leverage Rust's capabilities to achieve several goals: they automate the construction of a core component of program proofs, they are *lightweight* in the manual annotations that they require, they offer a *rich* language to express functional properties, and they do not require the verification users to be experts in advanced topics such as separation logic or interactive theorem proving.

1.1 Deductive Verification

At a high level, software verification techniques aim to statically guarantee that a program implementation agrees with a formal specification of its correctness. Both sides of this problem, expressing a formal specification, and checking it against the implementation, are subjects of study in this discipline.

The properties expressed by a formal specification can be divided into *safety* properties (i.e., "bad things never happen") and *liveness* properties (i.e., "good things eventually happen") [5]. While some of these properties can be defined based on the semantics of a programming language — for example, absence of crashes, or absence of undefined behavior — other aspects depend on what the program is supposed to do, and thus need to be expressed by verification users using, e.g., *program annotations*. Depending on the technique, these annotations can describe *functional correctness* properties of the desired input-output behavior [6], *temporal* properties such as that an event should *eventually* happen, and so on.

A formal specification is often based on a *specification logic* that gives meaning to the specifications. For example, temporal properties can be expressed within temporal logics such as LTL [7] or CTL [8], while functional correctness properties are often expressed using Hoare triples [9]. These logic systems can then be used to build several different verification techniques, such as static analysis [10], model checking [11], and deductive verification [9].

[5]: Lamport (1977), *Proving the Correctness of Multiprocess Programs*

[6]: Dunlop et al. (1982), *A Comparative Analysis of Functional Correctness*

[7]: Pnueli (1977), *The Temporal Logic of Programs*

[8]: Clarke et al. (1981), *Design and synthesis of synchronization skeletons using branching time temporal logic*

[9]: Hoare (1969), *An Axiomatic Basis for Computer Programming*

[10]: Cousot et al. (1977), *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*

[11]: Clarke et al. (2018), *Model checking, 2nd Edition*

[9]: Hoare (1969), *An Axiomatic Basis for Computer Programming*

This thesis focuses on deductive verification: a technique that can reason about complex program properties and can scale to large codebases thanks to its modularity. In deductive verification, the correctness of a program is expressed as a set of logical statements, which are then proven using a proof system such as Hoare logic. This proof can be constructed by hand using an *interactive theorem prover* such as Coq [12] or Isabelle/HOL [13]. Alternatively, it can be constructed (semi)automatically with the aid of SMT-solvers such as Z3 [14] or CVC5 [15]. What enables deductive verification to be modular is its usage of *function contracts* to over-approximate the behavior of a function call using only its annotations. This way, the verification of a large program can be divided into independent verification units that (1) can be verified in parallel and (2) make the program proof more robust to code changes, so that only the verification units whose code changed need to be reverified[2].

[12]: Bertot et al. (2004), *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*

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

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

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

2: Depending on the code change, if the proof of a verification unit cannot be reestablished, it may be be necessary to additionally update its function contract and reverify the verification units that depend on it. In the worst case, this process can still require rebuilding the entire program proof, but for most minor code changes, this is not necessary.

In Hoare logic, logical statements are expressed using a *Hoare triple* of the form $\{P\}C\{Q\}$, where C is a program, while P and Q, expressed logics such as first-order logic or separation logic, are called *precondition* and *postcondition*, respectively. Such a triple can be used to describe a *partial* functional correctness property, meaning that if the program is executed starting from a state that satisfies the precondition, then if it terminates without reaching an error its final state satisfies the postcondition. Separation logic was developed as an evolution of Hoare logic to reason more easily about heap modifications and concurrency. The core novelty is the separating conjunction operator $A \ast B$, expressing that there exists a partition of the program memory such that the assertion A holds in one partition, and B in the other. The advantage of this approach is that it makes it easy to prove that modifications to one partition of the memory cannot invalidate the assertions associated with other memory partitions; a reasoning step that is formalized with the *frame rule*.

Deductive verification techniques that are automated and modular have motivated the development of many *verification languages*, to express programs with formal specifications, and enabled the construction of *program verifiers*, to check a program against its specification. Among the verification systems based on first-order logic two notable examples are Boogie [16] and Why3 [17]. Both of them provide a verification tool with established industrial applications and an *intermediate* verification language, designed to ease the construction of new verifiers by translating other languages and their specifications into the intermediate one. One difference between the two systems is that Boogie builds on top of Z3 for automation, while Why3 is designed to use various external automated or interactive provers. Many automated program verifiers have been built based on Boogie: Spec# [18] for verification of an extension of C#, VCC [19] for verification of concurrent C code, and Dafny [20] for verification of a language that can be compiled into various other popular programming languages. Regarding Why3, some notable systems based on it are SPARK [21], which provides a verification tool for the Ada language, and a plugin for Frama-C [22], a static analyzer for C programs. Among the verification systems based on separation logic, Viper [23] provides an intermediate verification language and a verification tool with two backends: one based on a translation to Boogie, the other on symbolically executing a program and checking its conditions using Z3. The variant of separation logic that Viper uses, called *implicit dynamic*

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

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

[18]: Barnett et al. (2011), *Specification and verification: The Spec# experience*

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

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

[21]: Carré et al. (1990), *SPARK - an annotated Ada subset for safety-critical programming*

[22]: Cuoq et al. (2012), *Frama-C - A Software Analysis Perspective*

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

frames [24], provides benefits in terms of automation and conciseness, because in that logic functional specifications can be easily conjoined to the specification of memory safety, without having to intertwine the two as typically required in other separation logics.

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

Given this state of the art, achieving automation of separation logic is still not trivial. Even in Viper, designed for automation, constructing a proof of simple functional properties requires explicit proof-directive statements, such as the `fold` and `unfold` statements, to state where to apply the definition of the logical predicates used in the proof. For this reason, many program verifiers that are built on top of Viper — such as Nagini [25] for verification of Python and Gobra [26] for verification of Go — still require the user to write proof-directive statements to guide in certain cases the construction of the Viper proof.

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

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

1.2 Challenges

Despite the advancements made in the past few decades, program verification remains an expensive task, both in terms of manual effort (e. g., to write the specifications and proof hints that are needed by the verifier) and in terms of necessary expertise (e. g., knowledge of the intermediate verification language and program logic). In particular, separation logic and tools based on it are well-suited to reason about imperative programs that use shared mutable structures, but at the same time they also require deep — often doctorate-level — knowledge of the verification technique to reach proficiency.

The core goal of this thesis is to improve the state of the art by leveraging the Rust language and its strong type properties to simplify program verification. In particular, we aim to make the benefits of separation logic accessible to a larger verification audience, to the point where the usage of separation logic is an implementation detail that can be hidden from Rust verification users. Achieving our goal requires solving the following challenges.

Challenge 1: Modeling Types Rust's type system offers many static guarantees, among them transitive immutability, non-aliasing, and other properties related to memory safety. In some cases, these type properties are a design choice of Rust, while in other cases they are design choices of the API of a Rust library. Modeling these properties and their special exceptions in a program logic is challenging and, to the current day, a formalization that captures *all* of them does not exist. A model is necessary for two reasons. First, in techniques based on separation logic, a formal model of the types is a necessary step to enable reasoning and constructing proofs about their values. Second, the Rust developers intuitively expect certain properties, such as immutability, to be known to a program verifier. A formal model has to provide these guarantees to be *complete*, while at the same time, it should correctly model their limitations to be *sound*.

Challenge 2: Annotations Compared to other programming languages, there are two directions along which designing program annotations for Rust is challenging. First, Rust introduces *new limitations* that require *new annotations*. For example, when a Rust function uses mutable reference types in both its argument and return types, under certain conditions the reference passed as argument is *blocked* until the one returned by the function goes out of scope. This particular pattern requires designing specifications that, when expressing the postcondition of a function, can express the relation between the returned and the blocked reference in a way that is as natural as possible for Rust developers. Second, to work around some of its limitations, Rust provides *new features* that require *new annotations* too. For example, immutable references normally imply transitive immutability of their reachable values, but the language also provides a special exception, called *interior mutability*, to this rule. So, if the common annotations of a Rust program are designed based on the non-exceptional cases, describing these exceptions requires new annotations. Overall, the design of all these program annotations should be done in a way that maximizes expressivity, while at the same time keeping an eye on practical usability aspects.

Challenge 3: Automation As a further challenge, we want our verification technique to be automated to a level that is precise enough to prove memory safety. Our insight is that, since the Rust compiler is already able to guarantee memory safety with its static checks, there should be a similar way to leverage the properties of the Rust language to build automatically a memory-safety proof of safe Rust programs. The intent of achieving an automated memory-safety proof is to pave the way to the verification of unsafe Rust code, which should generate a compatible memory-safety proof, so that the two can be combined to achieve modular verification of Rust programs containing both safe and unsafe code. The difference in our aims is that for unsafe Rust the proof construction would not be expected to be fully automated.

1.3 Contributions

In this thesis, we address each of the challenges by developing novel verification techniques for two large subsets of Rust:

1. **Safe code**: programs and libraries entirely written in safe Rust code.
2. **Safe clients**: programs written in safe Rust code using trusted real-world libraries whose implementation contains unsafe code.

1.3.1 Safe Code

In Chapter 3, we address the challenges in the context of verification of safe code. To address Challenge 1, we designed an encoding of the properties of Rust types and signatures into the logic of implicit dynamic frames using Viper. To address Challenge 2, we designed a specification language for expressing the intended functional properties of Rust code, choosing a syntax and semantics that is familiar to Rust developers. This

specification language includes *pledges*: a novel specification construct for expressing the functional properties of functions returning references. To address Challenge 3, we provide an algorithm, based on a novel static analysis of Rust capabilities, to automate the encoding of Rust programs into the Viper language. We implemented our verification technique in an open-source tool called *Prusti* [27], which we use to evaluate our technique both on a large corpus of Rust code without annotations, and on a small corpus of interesting programs with hand-written contracts.

[27]: (2024), *Repository of the Prusti verifier for safe Rust*. URL: https://github.com/viperproject/prusti-dev

1.3.2 Safe Clients

In Chapter 4, we address the challenges in the context of verification of safe clients, where several static properties of Rust types are weakened by the usage of interior mutability. To address Challenge 1, we identify the novel notion of *implicit capabilities* of library types — i.e., type properties decided by the library developers — and we present a new technique that leverages them to reason about safe Rust code, even in the presence of interior mutability. To address Challenge 2, we propose a new annotation to specify the implicit capabilities of library types and several annotations to model some methods of these libraries as logical functions, some of which might even depend on the memory address of type instances reachable from their arguments. To address Challenge 3, we designed our verification technique to produce a model that can be fully expressed in first-order logic, so that it is well-suited for automation using an SMT-based verification toolchain. We implement our verification technique in an open-source tool called *Mendel* [28], showing with an evaluation that our technique supports popular types with interior mutability defined in the standard library, requiring little annotations on the client side of these libraries.

[28]: (2024), *Repository of the Mendel verifier for safe Rust clients of interior mutability*. URL: https://github.com/viperproject/mendel-verifier

1.4 Outline

Overall, this thesis is structured as follows:

- Chapter 2 introduces why Rust is a good fit for software verification, showing the language properties, verification opportunities and design principles of our Rust verification framework.
- Chapter 3 presents our novel technique for verification of safe Rust code.
- Chapter 4 presents our novel technique for verification of safe Rust clients.
- Chapter 5 presents notable design choices and solutions of our tool implementations.
- Chapter 6 concludes the thesis and presents future directions.

1.5 Publications

Part of the work in this thesis has been published in the following papers:

- Vytautas Astrauskas, Peter Müller, Federico Poli and Alexander J. Summers: *'Leveraging Rust Types for Modular Specification and Verification'* [29]
  This is the first paper in which we describe our verification technique for safe Rust code, which we implemented in the Prusti verifier. Vytautas and I contributed equally to most aspects of this paper. The notable differences are that Vytautas led the work on the borrowing DAG (described in this thesis but not in the paper), the encoding of shared references, and the functional-correctness evaluation. On my side, I led the work on the capability analysis (the algorithm that computes *place capability sets* in the paper), the large-scale core-proof, and the overflow-freedom evaluation.
- Vytautas Astrauskas, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers: *'How Do Programmers Use Unsafe Rust?'* [30]
  In this paper, we present our empirical study of how programmers use unsafe Rust code. Vytautas, Christoph and I contributed equally to most aspects of this paper. The notable differences are that Vytautas proposed the idea of the paper and led the implementation of the data-gathering tool. On my side, I performed the study of usages of unsafe code presented in Sec. 2 of the paper.
- Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers: *'The Prusti Project: Formal Verification for Rust'* [31]
  This paper provides an overview of our Prusti project for the verification of safe Rust code. Vytautas, Aurel, Jonáš, Zachary, Christoph and I contributed equally to most aspects of this paper.

Rust Verification 2

This chapter presents the reasons why Rust is a good language for doing software verification, and the design principles of our Rust verification framework on which the later chapters of the thesis are based.

2.1 Rust Guarantees

Compared to other popular programming languages, Rust stands apart for its particularly strong type system, which makes it possible to declare and enforce immutability, non-aliasing, and safety aspects of the language. As a preview, in this section, we are going to define and discuss:

- The *memory-safety* properties of the *safe* language subset of Rust.
- The *transitive immutability* property of the so-called *shared* references.
- The *linearity* properties of non-duplicable types, by which assignments *move* type instances instead of creating aliases.
- The *read-xor-write* property, by which types providing mutable and immutable access cannot have shared reachable memory locations.
- The *uniqueness* properties of *mutable* references, by which all reachable locations can be mutated only via the reference itself.
- The *library soundness* principle, which enables the safe encapsulation of unsafe Rust code, in a way that does not break the properties above.

The language design choice of using a strong type system has both advantages and disadvantages. Among the disadvantages, convincing the compiler to accept a Rust program is relatively harder than in other languages. For example, the checks regarding lifetimes (presented later) are infamously known to be difficult to learn for new Rust developers [32, 33]. Among the advantages, Rust programs that are accepted by the compiler are less likely to contain certain kinds of bugs, such as undefined behavior (UB) cases that plague C [34]. Moreover, as we show in this thesis, the strong invariants of Rust's type system make it easier for both tools and human developers to reason about many Rust programs.

[32]: Zhu et al. (2022), *Learning and Programming Challenges of Rust: A Mixed-Methods Study*
[33]: Zeng et al. (2018), *Identifying Barriers to Adoption for Rust through Online Discourse*
[34]: Wang et al. (2012), *Undefined behavior: What happened to my code?*

At a high level, Rust can be seen as having two language subsets, a *safe* and an *unsafe* one, with widely different safety properties and intended use cases. When writing in safe Rust, the compiler is able to fully check memory safety and absence of undefined behavior (e. g., writing to the target of a null pointer). This is the most popular language fragment; the one that all developers should strive to use whenever possible, especially if they are not experts. When writing unsafe Rust code, instead, the compiler relaxes some of its static checks and lets developers use C-style *raw pointers* and other potentially dangerous Rust features. The benefit is that this gives additional expressivity to developers who know what they are doing, but the main disadvantage is that it is up to those developers to carefully ensure that all strong type properties of Rust still hold. The guiding interoperability principle between these language fragments is that unsafe code should be minimized and hidden behind library

[35]: (2019), *Glossary Definition of Soundness*

abstractions that are *sound* [35], meaning that they cannot be used by safe Rust code in a way that causes undefined behavior.

The following subsections present some of the properties of Rust that are particularly useful for the verification of programs fully written in safe code, or programs written in safe code that use sound libraries.

2.1.1 Memory Safety

One of the main selling points of Rust is that the language is designed to prevent *memory errors* such as null pointer dereferences, buffer overflows, use-after-free errors, or usages of uninitialized memory. The language mainly achieves this thanks to its type system, which uses a concept of *ownership* to make sure that safe Rust code never accesses uninitialized memory locations nor dereferences invalid pointers.

In order to check valid usages of pointers, the Rust language makes a distinction between C-style *raw pointer* types and the safer pointer types called *references*. Raw pointer types have very few static guarantees and can be used only in unsafe Rust code, while reference types have stronger properties that are checked by the compiler and can thus be used in safe Rust. In particular, there are two types of references: *shared* references (also called *immutable*) and *unique* references (also called *mutable*). As the names suggest, the former provides shared immutable access to some data, the latter exclusive mutable access. A component of the type checker called the *borrow checker* takes care of ensuring correct usage of all references. For instance, it checks that there can never be usable shared and mutable references pointing at the same time to the same memory location, and that references can always be dereferenced as long as they are alive. To make it possible to conduct these checks statically, the Rust language associates with each reference a *lifetime*, which is used to describe and control what the reference might point to at runtime.

The documentation of the Rust language uses the concept of ownership to explain the checks regarding lifetimes. In the Rust terminology, every allocated memory value has a unique *owner*: a local variable that is responsible for deallocating all its owned memory when going out of scope. References are said to *borrow* the target memory location when they are created, starting a *loan*. When a reference is no longer used, all its loans *expire*, transferring the borrowed ownership back to the original owners. This is done because when an owning variable goes out of scope, the compiler checks that all the references that borrowed from it expired first. All this ownership tracking is done at compile time so that lifetimes can be elided during compilation and do not translate to runtime checks.

2.1.2 Immutability

In safe Rust, shared references are guaranteed to be *transitively immutable*. That is, as long as a shared reference can be used, it is not possible to mutate memory that is reachable by safe code through the shared reference. This property is checked by the borrow checker, which in the example of Fig. 2.1 is responsible for raising a compilation error

```rust
fn bad() {
    let mut data = 42;
    let x = &data;
    data += 1; // Compilation error
    drop(x)
}
```

Rust

```
error[E0506]: cannot assign to ‘data‘ because it is borrowed
 --> program.rs:4:5
  |
3 |     let x = &data;
  |             ----- ‘data‘ is borrowed here
4 |     data += 1;
  |     ^^^^^^^^^ ‘data‘ is assigned to here but it was
  |               already borrowed
5 |     drop(x)
  |          - borrow later used here
```

Output

Figure 2.1: An example of a Rust program with a borrow-checking error. The compiler prevents modifying the variable `data` because it is reachable from the shared reference `x`.

```rust
fn good(x: &u32){
    let first = *x;
    anything();
    let second = *x;
    assert!(first == second); // Always succeeds at runtime
}

fn anything() { /* ... */ }
```

Rust

Figure 2.2: An example of an assertion that never fails at runtime. The motivation is that the shared reference `x` guarantees immutability of its target instance, even across the `anything()` call that executes unknown safe Rust code.

when a statement attempts to modify the variable `data` while the shared reference `x` is still usable.

The rules of the borrow checker are infamously known to be (in some cases) hard to learn for Rust users [32, 33], but once they are satisfied the benefit is that reasoning about a program becomes much easier than in more permissive languages. In the program of Fig. 2.2, for example, the immutability of the shared reference `x` makes it possible to statically guarantee that no matter what the safe implementation of the `anything` function is, the target of `x` will not change and the `first == second` comparison will always succeed at runtime. Reasoning like this can be performed in a code review or can be automated in a verification tool, and is only directly possible in Rust or other languages with transitively-immutable types that cannot be circumvented by the programmer, e. g., with casts. Other popular languages such as Java, Python, C or C++ do not provide types with this kind of useful property, so verifying the equivalent of Fig. 2.2 in other languages would require a challenging program logic such as separation logic [3].

[32]: Zhu et al. (2022), *Learning and Programming Challenges of Rust: A Mixed-Methods Study*
[33]: Zeng et al. (2018), *Identifying Barriers to Adoption for Rust through Online Discourse*

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

When using unsafe code, it is possible to define library types whose *private* content can be modified via a shared reference. This *shared mutability* pattern is not in contradiction with the transitive immutability guarantee of shared references, because the *private* visibility modifier matters. In fact, safe code *cannot* directly access the private content of a library, so the transitive immutability property defined above stops at such library boundaries. The expectation is that library developers should only achieve shared mutability with unsafe code by using either (a) the special `UnsafeCell` type, whose documentation and special compiler

semantics clearly state its mutability properties under the definition of *interior mutability* [36, 37], or (b) C-style *raw pointers*, which are not subject to the invariants of reference types. In practice, developers try to avoid shared mutability whenever possible because it makes it more difficult to reason about their code.

[36]: (2024), *The Rust Reference: Interior Mutability*
[37]: (2024), *Unsafe Code Guidelines Reference: Interior Mutability*

2.1.3 Non-Aliasing

Aliasing — the situation where two expressions in a program might refer to the same instance — is one of the biggest challenges in program verification [38]. The Rust language restricts aliasing by enforcing several properties at compile time:

[38]: Dietl et al. (2013), *Object ownership in program verification*

1. **Linearity**: each use of a non-duplicable type *moves* its value, leaving the read-from place uninitialized. Because of this, assignments do not create aliases, but move values between places.
2. **Read-xor-write**: each type instance cannot be usable at the same time via both a *mutable* and an *immutable* type. Because of this, shared and mutable references cannot alias the same instance.
3. **Uniqueness**: mutable references are the *unique* way by which the reachable instances can be mutated. Because of this, multiple mutable references cannot alias the same instance.

Our insight, which we explore in this thesis, is that these properties enable *syntax-driven reasoning*, in that Rust guarantees that certain syntactically-different expressions always resolve to different memory locations. In particular, this brings two advantages to program verifiers:

- **Separation-logic reasoning**. The non-aliasing properties of standard Rust types imply that there are no mutable memory locations that are reachable from multiple type instances. As we show in this thesis, these type properties make it possible to reason about usages of the types in isolation using techniques inspired by program logics such as separation logic [3] and automated verification tools built on top of it, such as Viper [23].
- **Low-overhead specifications**. The non-aliasing properties implied by the types declared in a function signature make redundant some parts of user-provided contract annotations. As we show in our work, these type properties make it possible to lower the manual effort that users have to put in when specifying the functional behavior of a function.

[3]: Reynolds (2002), *Separation Logic: A Logic for Shared Mutable Data Structures*
[23]: Müller et al. (2016), *Viper: A Verification Infrastructure for Permission-Based Reasoning*

The following subsections further present each of the non-aliasing properties and the novel ways in which they help program verification.

2.1.3.1 Linearity (Affine Types)

The Rust language allows users to declare non-duplicable types called *non-copy* types. These types, when moved from one variable or field to another location, leave the former place uninitialized. The type-checker takes care of tracking which places might be uninitialized, raising a compilation error in case a statement attempts to access a possibly uninitialized place. In Rust, declared types are by default non-duplicable, but *duplicable* type declarations can be expressed by marking the type as `#[derive(Copy)]`.

The functions in Fig. 2.3 show the different semantics of assignments for copy and non-copy types: in the `duplicable` function, the `u32` type of `x` is Copy, so the assignment `y = x` *copies* the value of `x` into `y`. In `non_duplicable`, instead, the type `Vec<u32>` of `x` is non-copy, so `y = x` *moves* the values of `x` into `y`, leaving `x` uninitialized.

```rust
fn duplicable() {
    let mut x: u32 = 42;
    let y = x; // copies ‘x‘ to ‘y‘
    x += 1;
    println!(f"{x} {y}"); // Prints "43 42"
}

fn non_duplicable() {
    let x: Vec<u32> = vec![1, 2, 3];
    let y = x; // moves ‘x‘ to ‘y‘; uninitializing ‘x‘
    drop(x); // ERROR: value ‘x‘ is used here after move
}
```
Rust

Figure 2.3: Examples of functions using a duplicable and non-duplicable type for `x`, respectively. `non_duplicable` is rejected by the compiler because the assignment `y = x` consumes `x`.

Formally, the non-duplicable types in Rust are *affine types*, because the type-checker ensures that they can be used at most once. In the context of Rust, these types are also often informally said to be *linear*, although this name is technically incorrect because the formal definition of linear types implies that such types must be explicitly consumed exactly once [39]. This is not guaranteed by Rust, because (1) local variables can go out of scope implicitly and (2) the compiler does not check termination of the code in between the initialization of a variable and the statements that consume it. The functions in Fig. 2.4 show an example of this difference: the Rust compiler accepts all implementations, even though the latter function should be rejected in a language with linear types. The reason is that in `not_linear` the vector `x` is not explicitly consumed.

[39]: Girard (1987), *Linear Logic*

```rust
fn linear() {
    let x: Vec<u32> = vec![1, 2, 3];
    drop(x); // Explicitly consume the vector
}

fn not_linear() {
    let x: Vec<u32> = vec![1, 2, 3];
    // ‘x‘ goes out of scope without being explicitly consumed
}
```
Rust

Figure 2.4: Rust functions that demonstrate the difference between affine and linear types. The Rust compiler accepts all functions because `Vec` is an affine type. If `Vec` were a linear type, the compiler would need to ensure that `x` is always explicitly consumed, rejecting the `non_linear` function.

2.1.3.2 Read-Xor-Write

The Rust language disallows usable shared references from aliasing other usable mutable references or non-borrowed types. This is because the transitive immutability guarantee of shared references is incompatible with the mutability capabilities of the latter types. To reject programs that attempt to break this rule, the compiler checks that each place cannot be mutated as long as it might be reachable via a shared reference. As a consequence, shared references cannot alias mutable references, nor variables or fields that are not borrowed. The program in Fig. 2.5 shows an example of this. The first two functions check at runtime whether their

reference arguments are aliasing the same `u32` instance, raising a panic if they do. Because of the read-xor-write rule, mutable references cannot alias values reachable via shared references, thus the assertion in the `non_aliasing` function is guaranteed to never fail. The compiler takes care of rejecting `bad_client`, whose implementation tries to violate this property. However, the language does not prevent multiple shared references from being aliases of the same instance. So, it is possible for the assertion in the function `possibly_aliasing` to fail, for example when called from `good_client`.

```rust
fn non_aliasing(x: &u32, y: &mut u32) {
    assert!(addr_of!(*x) != addr_of!(*y)); // Never fails
}

fn possibly_aliasing(x: &u32, y: &u32) {
    assert!(addr_of!(*x) != addr_of!(*y));
}

fn bad_client(mut x: u32) {
    // Compilation error: cannot borrow ‘x‘ as mutable
    // because it is also borrowed as immutable
    non_aliasing(&x, &mut x);
}

fn good_client(x: u32) {
    possibly_aliasing(&x, &x);
}
```

Rust

Figure 2.5: Rust functions that demonstrate the non-aliasing between shared and mutable references. The `addr_of!` macro returns the memory location pointed by a reference. At runtime, the assertion in the `non_aliasing` function will never fail, because the type checker ensures that mutable references cannot alias values reachable via shared references. The function `bad_client` is rejected by the compiler because of this check. The assertion in the function `possibly_aliasing`, however, can fail because shared references can alias the same memory location, as it happens in the `good_client` function.

2.1.3.3 Uniqueness

Mutable references are guaranteed by the language to be *unique*, in that the sets of memory locations reachable from two mutable references cannot have any mutable location in common. This rule prevents any form of mutable aliasing between mutable references. For example, consider the program in Fig. 2.6, which defines two functions that split a mutable reference to a `Point` instance into two references. The first function, `bad_split`, is rejected by the compiler because it would return two mutable references from both of which `p.x` can be modified. `good_split` is instead accepted because the returned references do not have reachable memory locations in common.

```rust
struct Point {
    x: u32,
    y: u32
}

fn bad_split(p: &mut Point) -> (&mut Point, &mut u32) {
    // Compilation error: cannot borrow ‘p.x‘ as mutable
    // more than once at a time
    (p, &mut p.x)
}

fn good_split(p: &mut Point) -> (&mut u32, &mut u32) {
    (&mut p.x, &mut p.y)
}
```

Rust

Figure 2.6: Rust functions that demonstrate the uniqueness property of mutable references. The function `bad_split` is rejected by the compiler because it attempts to create two mutable references, from both of which a caller would be allowed to mutate the value of `p.x`. The function `good_split`, however, is accepted because the two returned references do not have common reachable memory locations.

2.1.3.4 Separation-Logic Reasoning

The non-aliasing properties of standard Rust types imply that given any two type instances the memory locations that, using safe Rust, are reachable and modifiable form two disjoint sets. Our novel insight that we explore in this thesis is that this disjointness property is the key to enabling and automating separation-logic reasoning. In particular, the technique of Ch. 3 relies on static non-aliasing properties to automate the construction of a separation-logic proof[1], while the technique of Ch. 4 uses the static non-aliasing information to encode separation-logic-style reasoning into first-order logic. The common ground between these techniques are the *framing* properties that they express. That is, the knowledge of which information is preserved across the execution of a statement.

1: More precisely, using the implicit dynamic frames logic [24].

For example, consider the `set_and_assert` function in Fig. 2.7, which assigns different values to its reference arguments and then checks the value behind the first argument `x`. In order to prove that the assertion never fails, a verifier has to know that the modification to the second argument `y` does not affect the memory reachable via `x`. This framing property can either be *proven*, checking that the non-aliasing claims of Rust's type system effectively hold, or can be *assumed* based on the compiler's type information. In separation logic, the proof can be done by using a *frame rule* around the `*y = 100` statement. The rule application, shown in Fig. 2.8, temporarily puts aside the knowledge about `x`'s target while reasoning about a statement that affects `y`. The separating conjunction $*$ in the proof models that the memory locations reachable via `x` and `y` are disjoint, as motivated above. In this example, the frame rule can be applied mechanically based on the syntax of the program, because the statement does not use `x` and because `x` does not alias the `y` used in the statement. Based on this idea, in Ch. 3 we develop a static capability analysis technique that we use to automate the construction of a Viper program. In Ch. 4, instead, we use the type-system information to generate the starting assumptions that a verifier can use to deduce the framing properties of Rust types with interior mutability, for which Rust's type system does not provide explicit guarantees. In both approaches, the non-aliasing properties of Rust play a key role and help to reason about a large number of programs.

```rust
fn set_and_assert(x: &mut u32, y: &mut u32) {
    // x ↦ _ * y ↦ _
    *x = 42;
    // x ↦ 42 * y ↦ _
    *y = 100;
    // x ↦ 42 * y ↦ 100
    assert!(*x == 42);
}
```

Rust

Figure 2.7: A Rust functions annotated with the separation-logic assertion that holds at each program point.

$$\frac{\dots}{\frac{}{\{y \mapsto -\}\ *y\ =\ 100\ \{y \mapsto 100\}}}{\{x \mapsto 42 * y \mapsto -\}\ *y\ =\ 100\ \{x \mapsto 42 * y \mapsto 100\}}$$

Figure 2.8: Example application of a frame rule (at the bottom) to reason about the second assignment in Fig. 2.7.

2.1.3.5 Low-Overhead Specifications

Another novel insight that we explore in this thesis is that by reusing the non-aliasing properties of Rust types, verifiers for Rust require *fewer* annotations than verifiers for other languages. Consider again for example the function in Fig. 2.7. If the arguments `x` and `y` are aliases to the same memory location, then the assertion in the function body panics at runtime. To prove absence of panics in this situation, modular verifiers typically require the user to specify as a precondition that the two reference arguments point to different memory locations. For simple types such as `&i32` one way to do so is by writing `#[requires(addr_of!(*x) != addr_of!(*y))]`, but for more complex types (e.g., structures with fields, or recursive data types) absence of shared memory regions cannot be expressed by stating disequalities. A verifier for Rust, however, can automatically extract and use the non-aliasing properties that are already implied by the types in the function signature. Thus, unlike in other popular languages, for the program in Fig. 2.7, no additional preconditions are needed to verify the function implementation.

2.1.4 Absence of Undefined Behavior

Safe Rust code is guaranteed by the language design of Rust to have no undefined behavior (UB). In the unsafe fragment of the language, however, absence of undefined behavior is a property that needs to be guaranteed by Rust developers. An official formal definition of which Rust program executions are UB is still in progress, but for the time being, the main informal (but official) description is the Rust Reference [40]. This reference presents an under-approximated list of situations that are described to surely be UB [41], as can be seen in the excerpt presented in Fig. 2.9.

[40]: (2024), *The Rust Reference*

[41]: (2023), *The Rust Reference: Behavior considered undefined*

> Rust code is incorrect if it exhibits any of the behaviors in the following list. (...)
> **Warning:** The following list is not exhaustive. There is no formal model of Rust's semantics for what is and is not allowed in unsafe code, so there may be more behavior considered unsafe. The following list is just what we know for sure is undefined behavior. Please read the Rustonomicon before writing unsafe code.

Figure 2.9: Excerpt from the Rust Reference, presenting the list of behaviors considered undefined [41].

One example of UB is *unsynchronized data races*. That is, modifying a memory location while another thread is concurrently reading or writing it, without using synchronization or atomic primitives to ensure that the concurrent operations have defined behavior. By design, safe Rust cannot have undefined behavior. Therefore, the compiler takes care of rejecting any program written in safe Rust that might have unsynchronized data races. For example, the program in Fig. 2.10 is rejected because it tries to write to the same memory location `DATA` from two threads. Using atomic integer types, it is possible to fix the program as shown in Fig. 2.11, so that the program does not have UB but only exhibits a less dangerous (from the point of view of language semantics) race condition.

[42]: Jung et al. (2020), *Stacked Borrows: An aliasing model for Rust*

```rust
use std::thread;

static mut DATA: i32 = 0;

fn main() {
    thread::spawn(|| { DATA = 1; });
    thread::spawn(|| { DATA = 2; });
    println!("Data: {}", DATA);
}
```

Rust

```
error[E0133]: use of mutable static is unsafe and requires
    unsafe function or block
 --> src/main.rs:7:9
  |
7 |         DATA = 1;
  |         ^^^^^^^^ use of mutable static
  |
  = note: mutable statics can be mutated by multiple threads:
    aliasing violations or data races will cause undefined
    behavior

error[E0133]: use of mutable static is unsafe and requires
    unsafe function or block
  --> src/main.rs:10:9
   |
10 |         DATA = 2;
   |         ^^^^^^^^ use of mutable static
   |
   = note: mutable statics can be mutated by multiple threads:
     aliasing violations or data races will cause undefined
    behavior

For more information about this error, try `rustc --explain
    E0133`.
```

Output

Figure 2.10: An example of a safe Rust program that is rejected by the compiler. The root issue of this program is that it tries to concurrently modify the same memory location from two threads.

Recently, the Stacked Borrows [42] model and its Tree Borrows evolution [43] were proposed to formally define an aliasing model for Rust in a way that can be checked at runtime by the Miri interpreter [44]. Any violation of this aliasing model would be considered UB. At the time of writing, the consensus of the community is that an evolved version of Tree Borrows is going to become in the future the official aliasing model of the language. Since the differences between Tree Borrows and the list of UB in the Rust Reference are still being discussed, we conservatively designed the verification technique in this thesis to be sound under all models, at the expense of being sometimes incomplete. We give more details in Ch. 4.

[43]: Villani (2024), *Tree Borrows: A new aliasing model for Rust*

[44]: (2024), *Miri: An interpreter for Rust's mid-level intermediate representation*

2.1.5 Soundness of Libraries

To prevent the relaxed properties of unsafe code from polluting the strong properties of safe Rust, the Rust language uses a principle of *library soundness*. According to this principle, a library is sound only if it is impossible for Rust developers to write a safe client that uses the library in a way that causes undefined behavior. Each violation of this

```rust
use std::thread;
use std::sync::atomic::{AtomicI32, Ordering};

static DATA: AtomicI32 = AtomicI32::new(0);

fn main() {
    thread::spawn(|| {
        DATA.store(1, Ordering::Relaxed);
    });
    thread::spawn(|| {
        DATA.store(2, Ordering::Relaxed);
    });
    println!("Data: {}", DATA.load(Ordering::Relaxed));
}
```

Rust

Figure 2.11: An example of a safe Rust program with a race condition, implemented using the atomic integer type `AtomicI32` to avoid UB. Depending on the scheduling by the operating system, the program might print 0, 1 or 2.

principle is treated by the Rust community as a serious security issue [45, 46].

[45]: Tolnay (2019), *Soundness bugs*
[46]: (2024), *RustSec Advisory Database: Advisories with Keyword 'unsound'*

Our insight is that the library soundness principle helps to reason about safe usages of libraries implemented with unsafe code, making it possible to deduce that the library cannot implement certain pathological behaviors. For example, consider the code in Fig. 2.12. The implementation of the `library` module is unknown and might use unsafe code or other obscure Rust features. However, we assume to know that its API is sound. The function `lib_client` calls the library passing a reference to the argument `x`, where `Box` is the Rust way to allocate memory on the heap. Then, the client checks at runtime that `x`'s value did not change across a second library call with no arguments. In this case, the assertion never fails, and the goal of a verifier is to prove so. The immutability and non-aliasing type properties presented so far would help in case all used libraries were fully implemented using safe code. However, what if `lib` contains some unsafe code? Would it be possible for the library, during the `observe` call, to store in a global variable a raw pointer to the location of `x`, so that it can later use the global variable to mutate `x`'s value during the `do_something` call? As we explore in Ch. 4, the answer is no: the value of `x` cannot change across the `do_something` call, because if it did, the library would be unsound. In particular, it would be possible to use the library from safe Rust code in a way that would cause UB. Knowing that the value of `x` cannot change is useful for a verifier, because it makes it possible to prove that the assertion in the code always succeeds.

```rust
// A sound library
mod lib {
    pub fn observe(x: &mut Box<i32>) { /* ... */ }
    pub fn do_something() { /* ... */ }
}

fn lib_client(mut x: Box<i32>) {
    lib::observe(&mut x);
    let before: i32 = *x;
    lib::do_something();
    let after: i32 = *x;
    assert!(before == after); // Always succeeds at runtime
}
```

Rust

Figure 2.12: A safe Rust function calling a sound library with an unknown, possibly unsafe, implementation.

2.2 Practical Relevance

The Rust language provides many useful guarantees. Even just the non-aliasing properties are effective, in practice, at avoiding most *data races* and *unintended aliasing* bugs that plague other mainstream languages. However, existing verification techniques are unable to benefit from these guarantees, because they were designed for programming languages with a weaker type system than Rust's. As an example, in Sec. 3.1 we compare and discuss the guarantees of programs written in C and Rust. What is needed is a new methodology that comprehends and exploits the properties of Rust types. In this thesis, we achieve so with our novel verification techniques for Rust that we present in Ch. 3 and Ch. 4.

Given the strong compile-time checks of Rust, what are the remaining Rust-specific bugs that would be helpful to eliminate, or some properties that would be useful to ensure, with a verifier? In this section, we present several real-world examples. At a high level, we divide this non-exhaustive list into cases where verification helps the *users* or the *compiler* of Rust.

2.2.1 Helping Rust Users

In this subsection, we present cases where a verifier helps developers who write Rust code. Mostly, these are cases of ensuring absence of errors in the logic of a program, but also cases where the annotations that are needed for verification can be reused for other means.

2.2.1.1 Absence of Panics

In Rust, a panic is an explicit program termination operation that is used by Rust developers to stop the program in case of unrecoverable implementation errors. Users of software written in Rust should ideally never observe a panic, and if they do, it strongly signals that there is a bug in the implementation. While it is possible to design Rust programs that recover from unexpected errors, panicking is often used as a trade-off to still prevent bad behaviors but without adding too much complexity to the codebase. Given how panics are used, a clear application of verification is to guarantee that Rust programs never panic.

As an example, consider the program in Fig. 2.13, which defines the constructor of an iterator that advances by a custom number of steps at each iteration. Intuitively, the `step` argument should never be zero, but just stating a requirement in the documentation is not enough to guarantee absence of unintended usages. So, the first statement in the implementation checks the desired property at runtime with an `assert!` statement, which panics in case `step` is zero. A necessary goal for a verifier, in this case, is to check at every call site that the `step` argument is never zero.

```rust
impl<I> StepBy<I> {
    #[inline]
    pub(in crate::iter) fn new(iter: I, step: usize)
    -> StepBy<I> {
        assert!(step != 0);
        let iter = <I as SpecRangeSetup<I>>::setup(iter, step);

        StepBy { iter, step: step - 1, first_take: true }
    }
}
```

Rust

Figure 2.13: Part of the implementation of the `StepBy` core library type [47]. Its constructor performs a precondition check at runtime using an `assert` statement.

2.2.1.2 Absence of Arithmetic Errors

Unlike C and C++, the Rust language takes care of ensuring that arithmetic errors like integer overflows do not cause undefined behavior. Depending on the compilation flags chosen, in Rust, overflows at runtime either panic or are computed using wraparound behavior. Both outcomes are considered bugs in a Rust program, and the core library exposes special functions to explicitly perform wrapped addition when a developer intentionally wants it. Thus, proving absence of unintentional overflows is a desirable feature for a Rust verifier, as well as absence of other arithmetic errors.

To see an example, consider the buggy bisection implementation in Fig. 2.14, which performs a binary search on the domain of a monotonically increasing discrete function. The assignment that computes `mid` performs an addition between two `usize` variables, `low` and `high`, which might overflow if the size of the domain is large enough. Since nothing in the documentation limits the size of the domain, the overflow is reachable and is a bug. As a fix, the developers should in this case use the `low + ((high - low) / 2)` expression, which computes the desired value without overflowing.

2.2.1.3 Reasoning About Determinism

When manually reasoning about function calls that have only shared references as arguments, an unfortunate mistake is to believe that identical sequential calls are going to produce the same results. This misconception is perhaps incorrectly motivated by the properties of shared references, which provide the necessary immutability properties across the calls, but do not ensure in any way deterministic execution. Consider for example the `main` function in Fig. 2.15, assuming for simplicity that the `Data` type has no interior mutability. Because the `data` variable is passed

```rust
/// A monotonically increasing discrete function, with domain
/// [0, domain_size)
trait Function {
    fn domain_size(&self) -> usize;
    fn eval(&self, x: usize) -> i32;
}

/// Find the `x` s.t. `f(x) == target`
fn bisection<T: Function>(f: &T, target: i32) -> Option<usize>
{
    let mut low = 0;
    let mut high = f.domain_size();
    while low < high {
        let mid = (low + high) / 2;
        let mid_val = f.eval(mid);
        if mid_val < target {
            low = mid + 1;
        } else if mid_val > target {
            high = mid;
        } else {
            return Some(mid)
        }
    }
    None
}
```

Rust

Figure 2.14: The implementation of a bisection method on a discrete function. The program contains a bug because `(low + high) / 2` might overflow for large enough domains. The fix is to use `low + ((high - low) / 2)`.

to the `compute` functions as a shared reference, it is guaranteed that both calls will be invoked with identical arguments. This is not enough to guarantee that the results `a` and `b` of the calls are equal because the execution of `compute` can still be non-deterministic. For example, it might internally perform system calls to read the current time, use randomization libraries, perform I/O operations, and so on. So, without any additional information about the implementation of `compute`, a sound verifier should point out that the final assertion might fail.

```rust
fn compute(data: &Data) -> i32 {
    // ...
}

fn main() {
    let data: Data = // ...
    let a: i32 = compute(&data);
    let b: i32 = compute(&data);
    assert!(a == b);
}
```

Rust

Figure 2.15: An example of sequential calls that only take shared reference arguments. Without knowing the implementation of `compute`, a verifier should report that the final assertion might fail. The reason is that the Rust language ensures immutability of the arguments, but not determinism of the implementation of `compute`.

To see how unexpected non-determinism leads to a real-world bug, consider the code snippet in Fig. 2.16, extracted from a library designed for developing operating systems in Rust [48]. The important expressions to focus on are the two `self.reference.len()` expressions, which are desugared by the compiler into `self.reference.deref().len()`. All method calls in this chain take a shared reference, so it might seem that the two expressions evaluate to the same values because they use identical arguments. However, since the implementation of `deref` is provided by the user of this library, by using types with interior mutability, or by performing non-deterministic operations, the data checked in the assertion

[48]: (2024), *The rust-osdev/volatile crate*

can actually be different from the data passed later to the `intrinsic::` function call [49]. Issues such as this one are called time-of-check to time-of-use errors because a property is violated after the runtime operation that was responsible for checking it. A sound verifier should identify the bug in this example by reporting that, despite the runtime check, some precondition of the non-overlapping copy is not guaranteed to hold. In this case, a fix for the bug is to call `self.reference.deref()` exactly once, storing the result in a local variable. This way the user-provided implementation of `Deref` would only be executed once, avoiding issues due to the potential non-determinism.

[49]: (2022), *Issue #26 in the rust-osdev/volatile crate*

```rust
#[derive(Clone)]
#[repr(transparent)]
pub struct Volatile<R, A = ReadWrite> {
    reference: R,
    access: PhantomData<A>,
}

impl<T, R, A> Volatile<R, A>
where
    R: Deref<Target = [T]>,
{
    pub fn copy_into_slice(&self, dst: &mut [T])
    where
        T: Copy,
    {
        assert_eq!(
            self.reference.len(),
            dst.len(),
            "(...) different lengths"
        );
        unsafe {
            intrinsics::volatile_copy_nonoverlapping_memory(
                dst.as_mut_ptr(),
                self.reference.as_ptr(),
                self.reference.len(),
            );
        }
    }
}
```

Rust

Figure 2.16: A code snippet from the `rust-osdev/volatile` library, containing a time-of-use to time-of-check bug due to a potential non-deterministic implementation of `Deref`.

There are many ways to achieve non-determinism in Rust. Almost all of them, e. g., system calls and concurrency, require some usage of unsafe code in the implementation of a library. However, even by using only safe Rust code a program can achieve non-determinism by casting a reference to an integer, revealing in this way a memory address that for security reasons is intentionally randomized by most operating systems [50]. Reasoning about determinism requires considering many special cases of the semantics of Rust, which developers can easily forget since they are relatively rare in practice. This is where a verifier becomes useful. By complementing the immutability guarantee of the shared references with some contract annotation for determinism, as we do in our thesis with the purity annotations `#[pure]` presented later, a verifier would be able to prove the cases in which assertions like the one in Fig. 2.15 are guaranteed to never fail.

[50]: Xu et al. (2003), *Transparent Runtime Randomization for Security*

2.2.1.4 Proving Rust Safety Conditions

There are multiple reasons why a developer might have to declare that a function is *unsafe*. One particular reason is to signal that the function has a precondition that must be satisfied in order for the function to be safely executed. Moreover, there are cases in which calling a function of this kind is the only reason why an unsafe block exists[2]. In this special situation, the usage of unsafe Rust can be fully checked to be correct by proving with a verifier that the precondition of the function is always satisfied. This is beneficial for Rust developers: instead of having to manually check the safety condition of all unsafe code usages, they would only need to specify the precondition of the unsafe function, annotating that the precondition is sufficient to ensure safety.

2: Calling unsafe functions is only possible from other unsafe functions or from code blocks marked as unsafe.

Consider for example the code snippet in Fig. 2.17, taken from the integer square root implementation of a library for smart contracts [51]. The `unchecked_div` function is a compiler intrinsic that is faster than a standard integer division in Rust, but unsafe because it causes UB when it is called with unsigned integer types passing a zero divisor. In the example, this condition was manually checked by a developer: as the comment suggests, proving that `xkn` is never 0 is enough to guarantee that the unsafe block used in the expression is safe. However, this manual process is error-prone and would benefit from the automation that a program verifier can offer.

[51]: (2020), *Square root implementation in the ontology-wasm-cdt-rust crate*

```rust
// div is safe since xkn will never be 0
xk = xkn.wrapping_add(unsafe { unchecked_div(n, xkn) }) / 2;
```

Rust

Figure 2.17: Example of usage of an unsafe block in the square root implementation of a library for smart contracts [51].

2.2.2 Helping the Rust Compiler

In this subsection, we present cases where a verifier could help the Rust compiler, either by double-checking some guarantees of the language or by proving program invariants that enable new optimizations.

2.2.2.1 Proving Memory Safety

One of the guarantees of Rust is memory safety, but there have been bugs in the past causing the Rust compiler to generate programs that were not memory safe. For example, consider the program in Fig. 2.18, which was reported as part of the issue #29723 of the Rust compiler [52]. Old versions of the compiler, before 2018, incorrectly accepted the program. The memory-safety bug is in the second branch of the match, where the variable `s` is used after being consumed during the evaluation of the condition of the first branch. The compiler should reject the implementation of the second branch, pointing out that `s` was consumed in the first branch. Since safety is one of the main selling points of Rust, the language would benefit from having a verifier that double-checks the guarantees of Rust. This can be seen as an instance of *dual programming*, the approach used to enhance software reliability and reduce the likelihood

[52]: (2015), *Issue #29723 in rustc: Variables moved from in match guards are still accessible in other match arms*

```rust
fn main() {
    let s = String::new();
    let _s = match 0 {
        0 if { drop(s); false } => String::from("oops"),
        _ => {
            // This should trigger an error,
            // s could have been moved from.
            s
        }
    };
}
```

Figure 2.18: An example of a Rust program with UB, which was incorrectly accepted by old versions of the Rust compiler [52].

of errors by independently implementing the same functionality in two different ways.

2.2.2.2 Identifying Redundant Runtime Checks

In order to prove absence of errors, program verifiers have to perform complex reasoning that typically exceeds the capabilities of a traditional compiler. After successfully checking a program, the verifier holds useful information that can be used to justify additional optimizations, such as dead code elimination.

For example, consider the `noop` function in Fig. 2.19. The compiler could recognize that the entire function is equivalent to a no-operation because there are no observable side effects. What is hard for the compiler to realize is that the expression in the `assert!` statement always evaluates to `true`, which is exactly what a verifier needs to prove when checking the absence of panics. By informing the compiler that the failing branch of the `assert!` is unreachable, the Rust compiler can optimize the program beyond what is currently possible.

```rust
pub fn noop(flag: bool, x: i32) {
    let y;
    if flag {
        y = 0;
    } else {
        y = x;
    }
    assert!(y == 0 || y == x);
}
```

Figure 2.19: An example of a Rust function with no side effects. The entire body of the function is dead code and could be removed as an optimization. However, using `rustc` version 1.72.0 with release optimizations enabled (the compilation flag `-C opt-level=3`) is not enough to compile the function to a single return assembly statement.

Our evaluation of Ch. 3 shows one possible way to compute redundant runtime checks using a verifier. Although in this thesis we do not study techniques to feed this optimization information back to the Rust compiler, that might be done by injecting calls to special functions, such as `std::hint::unreachable_unchecked`, which make the compiler assume that certain branches are unreachable.

2.3 Rust Verification Framework

In this section, we present the design principles of the verification framework for Rust that we developed to accommodate the verification

techniques described in Chapter 3 and Chapter 4. A description of the *implementation* of the framework is in Chapter 5. Overall, the guiding objective is to develop a verification tool for Rust that is sound and easy to use in practice.

2.3.1 Modularity

Practical verification tools need to scale well with the size of a codebase. To achieve so, we designed our verification techniques to be *modular*: the verification of a program is achieved by dividing the program into many smaller verification units, each of which is verified independently. This choice makes it possible for the verification to scale linearly with the size of a codebase. A necessary ingredient to make this possible are *contract annotations*, which describe the functional behavior of each unit. This way, when verifying a unit the implementation of other units can be ignored, by using their contracts as an approximation of their behavior. Overall, modularity leads to the following advantages:

- Since the verification of each unit does not depend on the *outcome* of the verification of other units, the verification of a large program can be easily parallelized.
- By using contracts to abstract the functional behavior of each unit, re-verifying a program after making a small change can be done efficiently. In particular, when the modification affects the implementation of a single unit, re-verifying the program only requires re-verifying the unit that changed. When the modification instead affects the contract of a unit, it is additionally necessary to re-verify all other units that were relying on the contract that changed. In both cases, the re-verification needs to verify only a small number of units; much less than the entire program in which they are contained.

To maximize modularity, we defined these verification units to be the functions and methods of Rust programs.

As an example, consider the chain of function calls in Fig. 2.20. As the name suggests, the outermost function `add_four` adds 4 units to its argument by repeatedly calling `add_two`, which in turn is implemented with calls to `add_one`. Since each addition might overflow, the contracts of all functions require their arguments to be small enough to handle the increments. Each contract also describes the functional behavior of its function, by ensuring that the result is the expected one. When verifying the program for the first time, the verifier will successfully check each function implementation against the provided contracts. Now, imagine that a developer optimizes the implementation of `add_two` by replacing its body with `y + 2`. In order to check the modified program, a verifier will only need to re-verify the implementation of `add_two`, knowing that the other two functions are still correct. In more detail: (1) `add_one` is still correct because it does not make any calls, so it cannot depend on changes to other verification units, while (2) `add_four` is still correct because its verification relies on the *contract* of `add_two` and not on its implementation. In this small example, the modularity property made it possible to avoid re-verifying two functions, and the benefit grows with the size of the program to be verified.

```rust
#[requires(x < i32::MAX)]
#[ensures(result == x + 1)]
fn add_one(x: i32) -> i32 {
    x + 1
}

#[requires(y <= i32::MAX - 2)]
#[ensures(result == y + 2)]
fn add_two(y: i32) -> i32 {
    add_one(add_one(y))
}

#[requires(z <= i32::MAX - 4)]
#[ensures(result == z + 4)]
fn add_four(z: i32) -> i32 {
    add_two(add_two(z))
}
```

Rust

Figure 2.20: A chain of Rust functions. A modular verifier can verify all functions in parallel. By reasoning about each function call using its contract and not its implementation, re-verifying a program after small changes can be done efficiently.

2.3.2 Executable Semantics of Specifications

The modularity requirement implies the need to write contract annotations describing the functional behavior of functions. When designing these annotations, we decided to match the syntax and semantics of regular Rust code whenever possible. For example, in Fig. 2.20 the contracts are written using regular boolean Rust expressions. The advantages are the following:

- Verification users do not need to learn another language in order to understand or write contracts.
- The contract annotations can be type-checked as regular Rust code, reporting familiar error messages. For example, the type checks help in case a user erroneously tries to add a boolean to an integer in a contract.
- The expressions can be easily copy-pasted between contracts and runtime checks, preserving their meaning. This, for example, makes it easy to convert existing precondition checks implemented with `assert!` statements to preconditions of a contract. Even more, this semantic equivalence makes it possible to call certain Rust functions from contracts, provided that they are marked and checked using the purity annotations of Ch. 3.

Even if it is possible to restrict the language of contracts to the point that the borrow checks are not necessary and can be disabled, we preferred to avoid that. One disadvantage of this approach is that the user might have to introduce shared references to satisfy the borrow checker. The advantages are that our language of contracts supports imperative idioms such as updates to local variables, rejecting at the same time nonsensical lifetime usages that would confuse Rust developers.

As an example of our contract annotations, consider the code in Fig. 2.21. The `IntTree` type represents either an empty tree or a node of the tree. The function `get_root_value` returns the value stored in its root if the tree is non-empty, otherwise the function panics. To annotate this function with a precondition, it is sufficient to copy-paste the runtime check done in the `assert!` statement into the `#[requires(..)]` annotation. Since the language of contract annotations supports Rust syntax, the `matches!`

Rust macro used in the copied expression does not pose a problem and is supported by the verification framework.

As an example of a contract with more complex semantics, consider the code in Fig. 2.22. The `return_true` function returns `true` by evaluating a convoluted expression that might cause many kinds of arithmetic errors: a division by zero if `x == 0`, an overflow in `42 * x` if `x` is too big and an underflow if `x` is too small. Instead of manually computing the exact range of values of `x` by which the function is correct, in this example, it is quicker for the user to just copy-paste the expression that should evaluate to `true` in the precondition annotation. By our design, the semantics of a precondition that might panic or have integer errors requires the verification of each caller to prove at the call site that the panics or integer errors in the precondition cannot happen. On the callee side, the implementation of the function will be checked to be panic-free under the assumption that the evaluation of the precondition did not encounter arithmetic errors, encoding precisely what we need.

```rust
enum IntTree {
    Node {
        value: i32,
        Box<IntTree>,
        Box<IntTree>,
    },
    Empty,
}

#[requires(matches!(tree, IntTree::Node { .. }))]
fn get_root_value(tree: &IntTree) -> i32 {
    assert!(matches!(tree, IntTree::Node { .. }));
    if let IntTree::Node { value, ... } = tree {
        return value;
    }
    0
}
```

Rust

Figure 2.21: Example of a possibly empty binary tree type, with a function that returns the value stored in the root node. In this case, the precondition matches exactly the runtime check implemented in the function.

```rust
#[requires(42 * x / x == 42)]
fn return_true(x: i32) -> bool {
    42 * x / x == 42
}
```

Rust

Figure 2.22: Example of a function that should only be called for certain values of `x`. Instead of manually specifying these values, our verification framework supports preconditions containing expressions that might cause arithmetic errors by implicitly generating the additional preconditions that are necessary to avoid the errors.

Our design of mirroring the executable semantics of Rust aims to make contracts more familiar to Rust users, but we do not want this choice to limit the expressivity of the annotations. To cover advanced cases, we decided to let the language of contracts still offer constructs with special semantics, such as existential and universal quantifications. Even if these quantifier extensions do not have realistic executable semantics[3], to maintain a uniform syntax we decided to use a Rust-like syntax that resembles a function call.

For example, consider the `binary_search` function in Fig. 2.23. The precondition of the function needs to express that the list of values is sorted, which is typically done by using a universal quantification. In the language of contracts used by the verification framework, this can be done

3: Defining quantifications over bounded integer types with loops would be possible, but iterating over all their values at runtime would not terminate in a reasonable amount of time in the case of big integer types.

```rust
#[requires(
    forall(|j: usize, k: usize|
        (0 <= j && j < k && k < values.len()) ==>
            values[j] <= values[k]
    )
)]
fn binary_search(v: i32, values: &[i32]) -> Option<usize> {
    // ...
}
```

Rust

Figure 2.23: Example of a binary search, whose precondition uses a universal quantification to require that the list of values should be sorted.

by calling the special boolean `forall` function, which accepts a Rust closure whose arguments are the quantified variables, and whose body is the quantified expression. Even if `forall` does not have a realistic implementation, the overall syntax of the contract still looks like regular Rust.

2.3.3 Automation

In our work, we decided to use SMT-based verification to automate the task of proving that each function implementation satisfies its contract. This way, the user can focus on writing the contracts rather than also writing a proof in an interactive theorem prover. Manually writing a proof would bring several disadvantages, because the user would first need to learn (1) how the Rust program is formalized in the theorem prover, and (2) how to make proofs using the logic defined in the theorem prover. Both tasks typically require expert knowledge that would make the verifier hard to use for regular Rust developers.

While SMT solvers provide powerful automation for verification tasks, they come with their own set of challenges. Mainly, the automation has limits that surface as timeouts or incompleteness errors in the verification of Rust programs. For example, the automation of non-linear arithmetic works only in trivial cases because the underlying logic is not decidable, and the search space for proofs involving quantifier instantiations is sometimes too big to be explored in a reasonable amount of time. To work around these issues when they happen, the verification technique should offer the users a manual way of guiding the proof search, so that the verification task falls back to be tractable automatically. For example, incompleteness issues might be solved by manually applying *lemma functions* in the program, at the point where the user knows that a certain proof step should be made. The proof step itself might then be small enough to be verified automatically on its own, or the lemma should be marked as *trusted* to signal that its correctness has been checked externally (i.e., manually, or using other verification tools). Regarding quantifications, a common technique to guide the proof search consists of annotating the quantifications with syntactic patterns called *triggers*, which limit the quantifier instantiations to the cases where the pattern matches a term of the proof.

2.3.4 Reusability of Specifications

Writing contract annotations is the only necessary step that a user has to perform manually. To mitigate the cost of this work, we designed the annotations of our verification framework to be easily reusable between different tools and verification techniques. For example:

- The verification techniques of both Ch. 3 and Ch. 4 consume the same core language of contract annotations, even though the techniques are different.
- The WaVe project, whose goal was to build a verified Wasm runtime, implemented a tool that consumes the contract annotations of our framework to generate property-based tests [53]. This gives more flexibility to the users, which can make trade-offs by deciding in some cases to extensively test a function against its contract instead of formally verifying its implementation. This can be useful to validate the contract of items that cannot be verified for some reason. For example, functions implemented using language features that are not supported by the verifier.

[53]: (2022), *Fuzz-gen tool implementation in the WaVe project*

2.3.5 User-Friendly Error Reporting

We designed the verification errors of our framework to be reported and phrased in a way that is natural to Rust developers, without exposing them to the complexity of the underlying verification techniques. This translates to the following requirement that the instantiations of the framework should adhere to:

- Each error message should be associated with a *position* that accurately identifies the source code item that triggered the generation of the error. For example, for a function call that does not satisfy the associated precondition the position of the error message should be the line of the call. Note that the *cause* of the error might be somewhere, but the position of the error message should be a good starting point to start from to debug the verification error. As an example, in Fig. 2.24, the `use_elevator` function call does not satisfy the precondition `people < 10`, so the verifier reports an error at the line of the call. As an additional suggestion, the verification error also reports the line at which the failing precondition is defined.
- The message of each error diagnostic should be phrased in a way that is understandable by Rust developers, who are not aware of the internals of the verifier. As an example, for a failing precondition, the error message should not be an obscure message such as "the SMT solver reported that <..> is satisfiable", where "<..>" might be some internal SMT encoding of a Rust expression. Instead, as shown in Fig. 2.24, the report should clearly describe what the error is — a failing precondition — without mentioning confusing implementation details.
- When the framework implementation encounters unsupported language features, it should report an error message explaining what the unsupported feature is and where it is located, without, e.g., crashing. An example is shown in Fig. 2.25, where the verifier reports an error because it does not support the raw pointers

```rust
#[requires(people < 10)]
fn use_elevator(people: u32) {
    // ...
}

fn main() {
    let people = 10;
    use_elevator(people);
}
```

Rust

```
error: [Prusti: verification error] precondition might not
    hold.
  --> program.rs:10:5
   |
10 |     use_elevator(people);
   |     ^^^^^^^^^^^^^^^^^^^^
   |
note: the failing assertion is here
  --> program.rs:3:12
   |
3  | #[requires(people < 10)]
   |            ^^^^^^^^^^

Verification failed
error: aborting due to previous error
```

Output

Figure 2.24: Example of a verification error, reported by the Prusti verifier, for a function call that does not satisfy a precondition.

contained in the `using_pointer` function. Bugs or failing validity checks in the implementation of the verifier should be reported in a similar way. This allows verification users to evaluate the tool even in codebases where not all function implementations are fully supported by the tool or verification techniques.

2.3.6 Command-Line and IDE Interface

To make it easy for Rust developers to incorporate verification in their workflow, we decided that our framework should offer both a command line interface and an IDE interface. The command line interface makes it possible to use the verification framework in an environment without a graphical interface. For example, the terminal of a developer, a server accessed over an SSH connection, or a script executed as part of an automated continuous integration process. The verification errors are then reported in the textual output of the tool as shown in Fig. 2.26, using the same reporting style[4] of the usual compilation errors of Rust. The IDE interface, instead, makes it possible to use the verification framework without leaving the code editor (Visual Studio Code [54] or VSCodium [55]) used for development. For example, starting the verification of a program can be done by clicking a button in an IDE or by executing one of the commands offered by the plugin. This way verification errors can be reported directly on the source code as shown in Fig. 2.27.

4: To distinguish between verification and compilation errors, all messages of the first kind have a prefix that specifies "verification error".

[54]: (2024), *Visual Studio Code*

[55]: (2024), *VSCodium*

```rust
pub fn using_pointers(ptr: *const i32) {
    assert!(ptr as usize % 8 == 0);
}
```

Rust

```
error: [Prusti: unsupported feature] raw pointers are not
    supported
 --> program.rs:2:13
  |
2 |     assert!(ptr as usize % 8 == 0);
  |             ^^^

Verification failed
error: aborting due to previous error
```

Output

Figure 2.25: Example of an error message caused by the presence of a Rust feature that is not supported by the Prusti verifier.

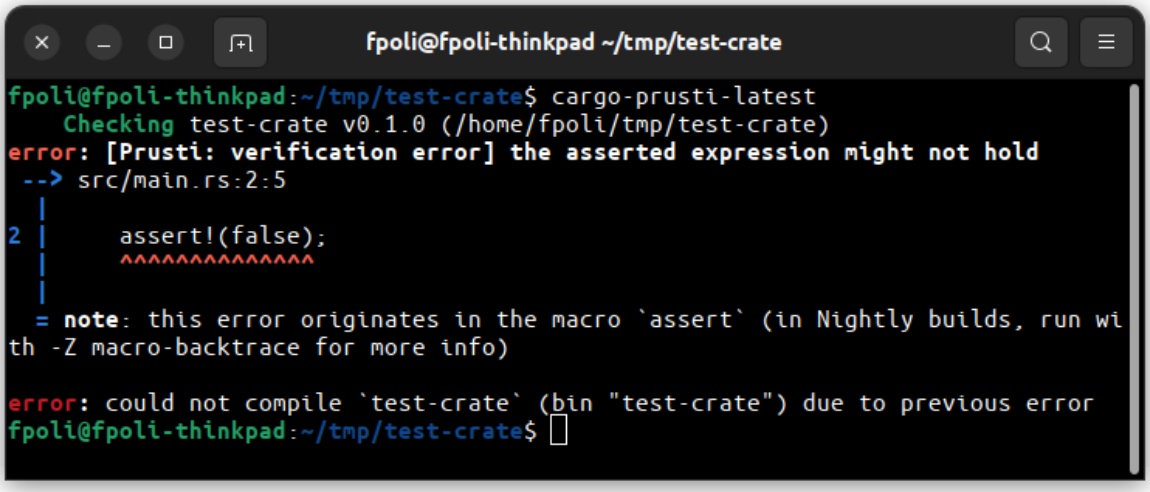


Figure 2.26: Screenshot of the command line interface of the verification framework, instantiated in the Prusti verifier.

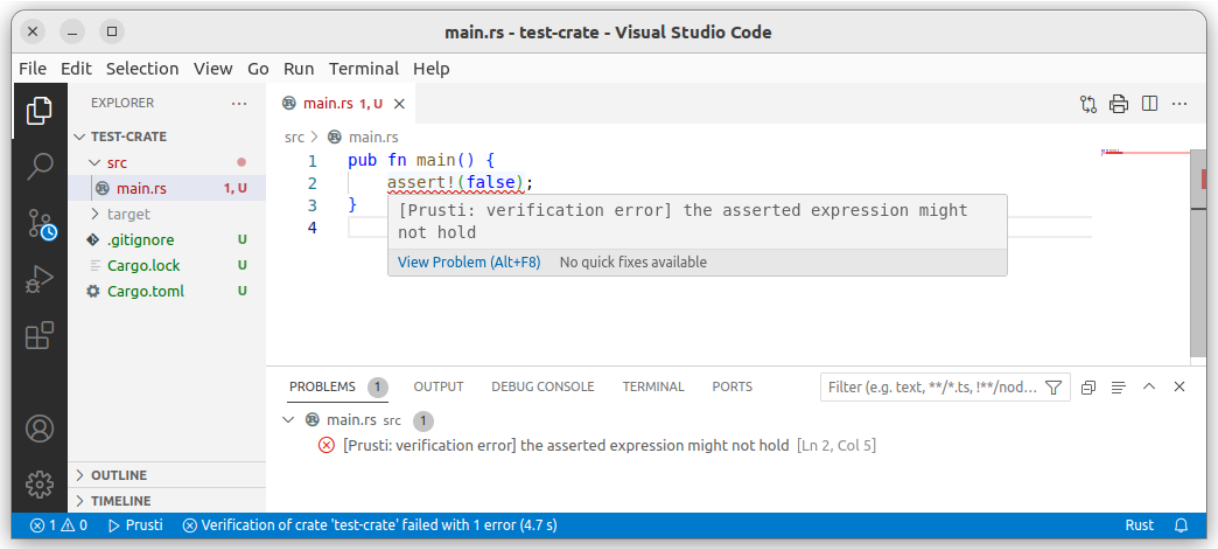


Figure 2.27: Screenshot of the IDE interface of the verification framework. The interface is implemented as a plugin for Visual Studio Code and VSCodium [54, 55]. Clicking on the "Prusti" button at the bottom left in the status bar starts the verification process using the Prusti verifier. The outcome of the verification is reported in the status bar and through the standard diagnostics handler of the IDE. The IDE reports the diagnostics in the source code (top right panel), in a list of problems (bottom right panel) and in the project structure (left panel).

3 Verification of Safe Code

In this chapter, which is based on our OOPSLA 2019 paper 'Leveraging Rust Types for Modular Specification and Verification' [29], we present our technique for the verification of safe Rust code.

[29]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*

Collaborations This work was done in collaboration with fellow doctoral student Vytautas Astrauskas. Vytautas and I contributed equally to most aspects of the work. Vytautas led the work on the borrowing DAG (Sec. 3.7), the encoding of shared references (Sec. 3.5.2), and the functional-correctness evaluation (Sec. 3.8.2.3). I led the work on the loop body invariant annotations (Sec. 3.4.2), the capability analysis (Sec. 3.7), the large-scale core-proof, and the overflow-freedom evaluation (Sec. 3.8.2.1, Sec. 3.8.2.2).

3.1 Introduction

To understand what makes Rust special in the context of software verification, let us start with a small example of verification of C code. The `client` function in Fig. 3.1, which is implemented in C, takes two linked lists by address as arguments. During its execution, it stores the length of the second list in a local variable `old_len`, appends an element of value 100 to the first list, and then checks that the length of the second list did not change. Even for such a simple function, proving that the final check never fails is challenging, because there are various possible bugs that a verifier has to rule out. First, one has to prove absence of memory errors, such as that the expression `b->len` does not dereference a null pointer and that the accessed memory location is initialized. Second, one has to prove that the two parameters are not aliasing each other. In fact, it is not enough to prove that `a != b`; it is also necessary to prove that the memory locations reachable from `a` and from `b` are disjoint. Third, one has to prove absence of data races in the implementation. Only after all these preliminary steps can one finally reason about the `append` call and prove that the length of `b` does not change.

```c
void client(List *a, List *b) {
    int old_len = b->len;
    append(a, 100);
    assert(b->len == old_len);
}
```

C

Figure 3.1: C function manipulating two linked lists.

Verifying this program in a modern program logic such as separation logic usually requires a lot of manual work. First, one needs to model the data structure referenced by the arguments by declaring a logical predicate, which represents the possibly unbounded set of memory locations of a linked list. Second, one needs to specify in a precondition that the two data structures do not have memory locations in common, which can be done using the separating conjunction operator in separation logic. Third, one needs to *define* the logical predicate of the `List` data structure,

modeling that it contains a field called `len` that can be modified by whoever holds the permission to modify the data structure. A popular technique to do so is by expressing an *ownership* relation between the data structure and the fields that compose it. Fourth, depending on the verification logic, one also needs to specify throughout the proof how to switch between an abstract representation of the data structure, where the fields are not visible, and an equivalent representation where the fields are instead visible. This can be done by using auxiliary proof annotations called *ghost code*, among which there are statements that exchange an instance of a predicate with its definition, and dual statements that exchange the fields of a definition with an instance of the predicate. Knowing that the lists `a` and `b` do not have elements in common, and knowing that `b->len` is part of the data structure of `b`, in separation logic one can immediately deduce that any modification to `a` will not affect `b->len`, which is the goal of this verification example.

The verification ingredients that we just presented are powerful, but quite difficult to use in practice. In a real-world proof, one might end up having to deal with hundreds of proof steps, each using one of these ingredients. The result is that verification can usually be done only by proof experts, who know the details of the program logic used in the proof. Avoiding these proof steps is not an option in these modern logics, because proving basic properties such as memory safety and non-aliasing is necessary in order to prove more interesting properties such as functional properties. In our work, we call a *core proof* this proof of memory safety that one has to build before proving any of the functional properties of interest.

In this chapter, we focus on the verification of Rust code. We present a technique that solves verification challenges like those described in the previous example by leveraging some of the Rust properties presented in Sec. 2.1, simplifying the functional specifications and automatically constructing a core proof. Consider for example the code in Fig. 3.2, which is an idiomatic Rust translation of the C code of Fig. 3.1. The compiler checks many more properties than in C. First, the compiler takes care of ensuring absence of memory errors in the program, which means for example that the expression `b.len()` will not dereference a null pointer nor access uninitialized memory locations. Second, the non-aliasing properties of the mutable reference types in the arguments, checked as well by the compiler, guarantee the disjointness property between the data structures of `a` and `b`. This holds because the type system of Rust uses a notion of linear capabilities for its type checks, by which mutable references can be seen as temporary holding *unique access* to the referenced data structures. Since a memory location cannot have more than one unique owner, the data structures are disjoint. Finally, the compiler also takes care of ensuring absence of data races, which are defined to have undefined behavior in a fully-safe Rust program. Following the checks of the Rust compiler, our technique automatically generates the core proof and all the auxiliary ghost code annotations that it requires. What is left is a proof where the verification user only needs to plug-in the functional specification of its functions, by writing contract annotations in the Rust source code. Our technique takes care of integrating these functional specifications in the proof, so that verification users are not required to be experts of the program logic used by our technique.

```rust
fn client(a: &mut List, b: &mut List) {
    let old_len = b.len();
    append(a, 100);
    assert!(b.len() == old_len);
}
```

Rust

Figure 3.2: Rust functions manipulating two linked lists. The types and the language guarantee non-aliasing, mutability and memory-safety properties that are not guaranteed in Fig. 3.1.

The language of contracts used by our technique is mostly standard, except for some extensions that we developed to handle challenging patterns of Rust. The functional behavior of functions is specified by pre- and postcondition annotations, in which the `result` keyword represents the returned value and logical conditions can be expressed using a subset of boolean Rust expressions called *pure* (deterministic, side-effect free, and non-diverging). To express mathematical relations, the language offers universal and existential quantifications, as well as the `old(..)` construct to evaluate an expression written in a postcondition using the pre-state of the function (i.e., just before the execution of the function). Among the specification challenges, we identify that the special *reborrowing function* pattern of Rust, which arises when a Rust function returns a reference created from some other reference-typed argument, requires the user to write postconditions that refer to a future program state where the returned reference expires. To address this case, we developed a novel contract annotation called a *pledge* to express how future modifications to the returned reference affect the original reference from which it was created. Additionally, loops in Rust can have complex control-flow graphs (CFG) because the loop guard can contain statements with side effects or early returns. For example, the `?` operator can be used in Rust expressions to conditionally perform an early return of an error value, even during the evaluation of the condition of a loop. This does not match classical loop invariant definitions, which assume loop guards to be pure. To address these cases, we developed a new annotation called *loop body invariant* that generalizes classical loop invariants, making it possible for verification users to specify an invariant even at an intermediate program point in the body of a loop.

The verification technique presented in this chapter relies on what we call the *explicit* properties of Rust types. That is, properties that are determined by the semantics of the Rust language. This is to contrast with the *implicit* type properties that developers can define when developing new libraries using unsafe code. The intuition behind the name is that *explicit* properties hold whenever a developer can *see* a type annotation in a Rust program, meaning that a verifier can programmatically discover these properties just by traversing the type declarations (e.g., tuples, `struct`, `enum`, references). The *implicit* properties, instead, are in the best case described only in prose in the documentation of the libraries. Handling the latter case is out of scope for the current chapter, but is covered in Ch. 4.

Contributions The main contributions of our work are:

- We present a specification language for expressing the intended functional properties of Rust code. This language is designed to use the same syntax and semantics as Rust in order to be familiar to Rust developers.

- We present two annotations that we developed to handle special patterns of Rust code: *pledges* for expressing the functional properties of functions returning references, and *loop body invariants* for expressing loop invariants in the presence of loop conditions with side effects or early return statements.
- We present an encoding of the properties of Rust types and signatures into the logic of implicit dynamic frames [24, 56], a variant of separation logic [3].
- We automate the encoding of Rust programs into the Viper verification language, defining an imperative and a functional translation of Rust code. A key ingredient of our automation lies in a static analysis of capabilities, which elaborates and augments the typing information provided by the Rust compiler.
- We provide an implementation of our verification technique in a tool called *Prusti*. Using Prusti, we evaluate our technique on a large-scale corpus of unannotated real-world code, as well as a hand-written collection of Rust programs annotated with functional specifications. The version of our tool used for the evaluation is available as an artifact [57], while more recent versions are available as open-source software [27].

[24]: Smans et al. (2009), *Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic*
[56]: Parkinson et al. (2012), *The Relationship Between Separation Logic and Implicit Dynamic Frames*
[3]: Reynolds (2002), *Separation Logic: A Logic for Shared Mutable Data Structures*

[57]: Astrauskas et al. (2019), *Software Artifact for the OOPSLA'19 Paper Titled "Leveraging Rust Types for Modular Specification and Verification"*
[27]: (2024), *Repository of the Prusti verifier for safe Rust*. URL: https://github.com/viperproject/prusti-dev

Outline The rest of this chapter is structured as follows. In Section 3.2, we provide an overview of the Viper verification language. In Section 3.3, we present our verification approach, based on the construction of a core proof using the Viper verification language. In Section 3.4, we introduce two annotations that we designed to handle Rust-specific code patterns. In Section 3.5, we define our encoding of Rust types into Viper predicates, and in Section 3.6, we define our encoding of Rust code into imperative or functional Viper code. In Section 3.7, we explain how we automate the construction of a Viper proof by using a static analysis of type capabilities. In Section 3.8, we implement our technique in a verifier called *Prusti*, and we evaluate it on various verification tasks. In Section 3.9, we discuss related work, then we conclude in Section 3.10.

3.2 Viper Background

In our work, we use the Viper verification language to build a correctness proof in a variant of separation logic called implicit dynamic frames [24, 56]. We chose to use Viper based on three requirements. First, we want to adhere to the design principles of our verification framework (Sec. 2.3), especially modularity and automation. We identified that the implicit dynamic frames of Viper are useful in this regard because they make it easier to automatically combine the functional specifications with a memory-safety proof. Second, as introduced in this chapter we want to encode properties of Rust types using separation logic. Third, as a practical goal, we want to minimize the engineering effort required to implement the verifier, by reusing existing automated tools and techniques where possible. Viper is one of the few verification toolchains that satisfy all such requirements. Other solutions that we considered lack in automation [12, 58], do not offer separation-logic primitives [16, 17, 20], or are not designed to be primarily used as intermediate verification languages.

[24]: Smans et al. (2009), *Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic*
[56]: Parkinson et al. (2012), *The Relationship Between Separation Logic and Implicit Dynamic Frames*
[12]: Bertot et al. (2004), *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*
[58]: Jung et al. (2018), *Iris from the ground up: A modular foundation for higher-order concurrent separation logic*
[16]: Barnett et al. (2005), *Boogie: A Modular Reusable Verifier for Object-Oriented Programs*
[17]: Filliâtre et al. (2013), *Why3 - Where Programs Meet Provers*
[20]: Leino (2010), *Dafny: An Automatic Program Verifier for Functional Correctness*

The Viper verification infrastructure is built around an intermediate verification language, called Viper as well, used as an abstraction layer between the development of frontend verifiers (e.g., our verifier for Rust) that generate Viper encodings, and backend solvers (e.g., symbolic execution, verification condition generation) that take care of performing the verification.

3.2.1 Permissions and Resources

One of the main features of the Viper language is the use of *fractional permissions* to reason about access to memory locations and usage of other kinds of resources. The Viper entities used to model imperative code blocks are called *methods*. Any method in a Viper program is allowed to read from the memory locations for which it holds a *non-zero* permission, while to also perform modifications the permission needs to be *full* (i.e., equal to one, the maximum). In particular, permissions can be declared in the *contract* of a method to state which memory locations may be accessed or modified.

Viper's resources can be declared using one of the following constructs:

- **Field permissions.** A Viper field models a memory location reachable from a reference stored in another memory location, called *object*. A field permission expression of the form `acc(x.f, p)` represents the fractional permission amount `p` for accessing the memory location of `x.f` via the field `f` of the object `x`. When `p` is omitted, its value is implicitly one. In Viper, the sum of all permissions held to the same memory location cannot exceed one, and this makes it possible to express non-aliasing properties. For example, the assertion `acc(x.f) ✻ acc(y.f)` always implies `x != y` because in the case `x == y` there would be an impossible permission of value 2 associated with the memory location of `x.f`[1].
- **Predicates.** A Viper predicate is a possibly recursive definition that describes a set of resources and related invariants. For example, predicates can be used to model data structures in Viper. To limit the proof search space and to enforce abstractions, predicates are by default treated iso-recursively [59] by the solvers, which apply predicate definitions only at the point where special proof-hint statements (`fold` and `unfold`) are used. For example, since Viper resources are affine, `fold` statements consume the definition of a predicate and produce a predicate instance, while the dual `unfold` statements consume an instance to produce its definition. A predicate does not necessarily need a definition. When that is missing, the predicate is said to be *abstract* and cannot be unfolded, meaning that the Viper proof would hold for any definition of the predicate.
- **Magic-wands.** In Viper, a magic wand of the form $P \mathrel{-\!\!*} Q$ represents an instance of a resource that enables obtaining the resource Q by consuming both the magic wand and a resource satisfying P. Creating a $P \mathrel{-\!\!*} Q$ resource – an operation called *packaging* – has the effect of consuming Q and other auxiliary resources to generate the magic wand, while *applying* $P \mathrel{-\!\!*} Q$ has the dual effect of consuming P and the magic wand to generate Q. Creating

1: Given an assertion of the shape $\langle A \rangle \ast \langle B \rangle$, the resources necessary to satisfy it are defined as the *sum* of the resources of $\langle A \rangle$ and of $\langle B \rangle$.

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

a magic wand requires proving that there exists a way to construct the Q resource starting from P and the auxiliary resources. Since this proof cannot always be constructed automatically, it can be provided by the Viper user using imperative Viper code.

Like with fields, predicate and magic wand resources can be declared in method contracts. That is, a method that unfolds a predicate, or applies a magic wand, can require a corresponding resource in the precondition of the method. For example, Fig. 3.3 shows a Viper program that declares four fields, two predicate definitions, and one method. The predicates are defined as a combination of other field and predicate resources. In order to unfold the predicates and access the fields, the method requires in its precondition a predicate instance, which has to be provided by each caller. Since the predicate instance is also declared in the postcondition, the method will return such a resource to its caller when it terminates.

```
field x: Ref
field y: Int
field start: Ref
field end: Ref

predicate Point(self: Ref) {
    acc(self.x) && acc(self.y)
}

predicate Segment(self: Ref) {
    acc(self.start) && acc(self.end) &&
    Point(self.start) && Point(self.end)
}

method align_y(s: Ref, t: Ref)
    requires Segment(s) && Segment(t)
    ensures Segment(s) && Segment(t)
{
    unfold Segment(s);
    unfold Point(s.start);
    unfold Point(s.end);
    s.end.y := s.start.y;
    fold Point(s.start);
    fold Point(s.end);
    fold Segment(s);
}
```

Viper

Figure 3.3: Viper method demonstrating the usage of predicates and permissions. All `&&` conjunctions in this program are separating conjunctions. In Viper, if a conjunct does not have resources in one of its operands, `&&` acts as a logical conjunction.

3.2.2 Memory Safety

In the Viper language, proving memory safety effectively means generating a Viper program such that every memory access respects the permission rules mentioned above. To prove so, each method should declare in its contract which permissions it might use during the execution and which permissions it returns to the caller at the end. Moreover, when a Viper method uses predicates, any manipulation of the permissions modeled by the predicate should be done by generating the correct `fold` and `unfold` statements.

As an example, consider the Viper method `align_y` in Fig. 3.3, which shows one of the Viper programs that we generate as part of our Rust

verification technique. The `s.end.y := s.start.y` statement in the method models an assignment between heap memory locations: `s` identifies one object, with fields `start` and `end`, both of which are objects with integer fields `x` and `y`. To perform the assignment, Viper checks that the method has the permissions to read `s.end`, `s.start`, `s.start.y`, and to modify `s.end.y`. These permissions are represented (as full permissions) by the `Segment(s)` predicate instance in the precondition of the method, stating that these permissions should be provided by the caller. However, the precondition is not enough to automate the proof. In order to reveal the definition of the `Segment` and `Point` predicates, the method must explicitly use the `unfold` statements. Then, in order to prove that the method gives back all these permissions to the caller, the method has to `fold` back the predicates and then declare `Segment(s)` in the postcondition. The contract also states that the method requires and ensures a predicate instance for the second argument, `Segment(s)`, even though such an argument is never used in the method's body. Overall, this Viper program constitutes a memory-safety proof, because each memory access is linked to a permission passed from the caller.

3.2.3 Functional Specification

Given the Viper memory-safety proof, proving partial-correctness properties requires additionally including in the Viper program a functional specification expressed with contracts and loop invariants. In other verification techniques, adding the functional specification may be a non-trivial task that involves changing the memory-safety proof. However, thanks to how we design the memory-safety proofs in our work, it is possible to combine the functional and the permission specifications with a simple conjunction operator (syntactically, a `&&` in Viper). Our approach leverages the implicit dynamic frames logic of Viper to achieve so.

Consider for example the method in Fig. 3.4. In order to add a functional postcondition, such as `s.end.y == old(s.start.y)`, by the way we structured the Viper program all that is needed is to conjoin a boolean expression after the predicate permission in the postcondition. Other approaches would require merging the functional specification with the predicate definition, for example by adding new parameters to the predicate and new boolean expressions to its definition. As done to build the memory-safety proof, also memory accesses in the specifications need to be supported by permissions. In our work, we chose to do this using Viper's `unfolding` expressions, which instruct the verifier where to apply a predicate definition. For example, the `s.end.y` in the postcondition should be wrapped in the `unfolding Segment(s)` and `unfolding Point(s.start)` expressions.

```
method align_y(s: Ref, t: Ref)
    requires Segment(s) && Segment(t)
    ensures Segment(s) && Segment(t) && s.end.y == old(s.start.y) && ...
{
    ...
}
```
Viper

Figure 3.4: Viper method demonstrating the simplified usage of functional specifications. The full Viper program would need to use `unfolding` expressions in the postcondition to declare where to apply the definition of the predicates.

3.3 Verification Approach

Our verification technique builds a partial-correctness proof of a Rust program in Viper, generating a Viper method for each Rust function or method, and one Viper predicate for each Rust type declaration. The proof construction is performed in two steps, starting from a *core* proof of memory safety that is then augmented with functional specifications[2]. The clean separation between these two steps is made possible by the properties of safe Rust and our design of the Viper encoding. Our motivation behind this approach is that the compiler needs to build an argument similar to a memory-safety proof when type-checking the program, so the same language restrictions that make Rust special should also make the automatic generation of Viper programs easier. When verifying unsafe Rust code, C code or other non-memory-safe programming languages, the clean separation between these proof-construction steps might not be guaranteed.

2: Proving termination – obtaining a *total-correctness* proof – is not covered in this thesis, but it is a straightforward extension of the presented verification technique. In fact, it has already been implemented in some experimental variants of our verification tool.

3.3.1 Core Proof

In our work, the core proof is a Viper memory-safety proof that is generated automatically by our technique. The purpose of this proof is manifold. First, the proof encodes the immutability and non-aliasing properties of Rust types into Viper using permissions. This way, these properties are available to the verifier and do not need to be re-stated in the contracts by the users. Second, the core proof ensures that memory accesses are performed only when they do not cause memory errors[3]. For example, this ensures that mutable references are dereferenced only when they are initialized and non-dangling. This serves as a solid ground on which to base future work on verification of unsafe code, where the Rust compiler does not ensure memory safety and it is instead up to verifiers to guarantee so. Third, the core proof checks that our permission-based formalization of how Rust capabilities flow is correct. For example, this prevents encoding bugs such as using immutable types as mutable ones, or failing to transfer back the capabilities of expiring mutable references correctly. This further validates our model and offers practical benefits during tool development.

3: When checking shared references, our technique additionally relies on the correctness of the borrow checker.

To construct the core proof, our technique encodes Rust types into Viper, modeling Rust's capabilities using Viper permissions. In particular, our insight is that Rust's write capabilities can be described as full Viper permissions, while Rust's read and immutability capabilities as small permissions strictly between zero and one. Based on this, each Rust structure is modeled as a Viper predicates, whose definition models each

non-reference field of the Rust structure with a Viper field and a predicate. For example, consider the `Segment` predicate in Fig. 3.3, which encodes the Rust type definitions of Fig. 3.5. In our Viper encoding, the predicate is parameterized by `self`, modeling the memory location at which one instance of the type is stored. In the predicate, `acc(self.start)` and `acc(self.end)` model the memory locations at which the fields of the Rust type are stored. In this example, the fields `start` and `end` are *owned* by the `Segment` instance, meaning that deallocating the latter would also deallocate the two fields. To reflect this structure, the predicate definition contains two nested predicate instances, `Point(self.start)` and `Point(self.end)`, modeling the Rust type instances that comprise the `Segment` type.

In our encoding of signatures, our technique uses Viper permissions to model the non-aliasing and immutability properties expressed by Rust. To do so, each argument is modeled as a predicate instance, whose permission amount depends on the properties of the Rust type. Owned types and mutable references are encoded using a full permission, so that those instances are known to be non-aliasing. Shared references, instead, are modeled using a small fractional permission, so that their target is modeled as being immutable. These resources encoding Rust's function arguments are then combined using separating conjunctions to construct the resources required by the precondition. For the postcondition, the encoding only needs to return the resources corresponding to capabilities that, in Rust, are transferred back to the caller. These are the resource representing the returned Rust type, and the target of Rust arguments of type mutable reference. For example, Fig. 3.6 shows the Rust function that we would encode to Viper as in Fig. 3.3. The encoded precondition requires the caller to provide full predicate instances for the two `&mut Segment` arguments, modeling that in Rust the two mutable-reference arguments `s` and `t` cannot have reachable memory locations in common. Then, the postcondition transfers back the resources that represent the capabilities of the target of the arguments. In this case, these are two `Segment` instances. If the type of the second argument had been a shared reference (i.e., `t: &Segment`), the encoding would have used a fractional permission amount for the predicate instance (e.g., `acc(Segment(t), 1/2)`) to model that `t`'s target is readable but not modifiable.

```rust
struct Point {
    x: u32,
    y: u32,
}

struct Segment {
    start: Point,
    end: Point,
}
```
Rust

Figure 3.5: The Rust types encoded to Viper in Fig. 3.3. For simplicity, `u32` is encoded not as a predicate, but as a simple `Int` type in Viper. The predicate encoding of primitive types is presented in Sec. 3.5.

```rust
#[ensures(s.end.y == old(s.start.y) && ...)]
fn align_y(s: &mut Segment, t: &mut Segment) {
    s.end.y = s.start.y;
}
```
Rust

Figure 3.6: The Rust function encoded to Viper in Fig. 3.3 and Fig. 3.4.

3.3.2 Functional Specification

The functional specifications in our Viper proofs are generated using the contracts annotated on the Rust code. Mainly, these annotations are preconditions, postconditions and loop invariants. In particular, the encoding is performed in two steps. First, the Rust expressions in a contract are automatically converted to a first-order logic expression using a technique that we present in Sec. 3.6.2. As part of this step,

the Rust expression is checked to be deterministic and side-effect-free. Second, the first-order logic expression is converted to Viper, introducing the `unfolding` constructs that Viper uses as proof hints to know when to apply the definition of predicates. This step is described in Sec. 3.7, and is based on a static analysis of Rust capabilities that we developed. Third, thanks to Viper's choice of using the implicit dynamic frame logic in combination with the design of our predicate encoding of Rust type capabilities, the boolean Viper expression representing the functional specification can be conjoined to the contracts of the core proof using a simple `&&` conjunction. As an example, Fig. 3.6 shows part of a postcondition in Rust that is encoded to the Viper code reported in Fig. 3.4. While the core proof is never expected to fail, a Viper program with user-provided functional specifications might fail with a verification error. In those cases, a back-translation step in our tool converts Viper's verification errors into user-readable diagnostics reported on the Rust program, pointing out which user annotation was involved in the error and why (cf., Sec. 5.2.1). This way, the verification user never has to interact with the low-level details of the core proof (or understand Viper at all).

3.3.3 Limitations Regarding Unsafe Code

The technique presented in this chapter relies on the explicit properties of Rust types. As a consequence, the programs that are supported cannot have unsafe code. The reason is that unsafe code can break (in special cases) the properties of types used in safe Rust. For example, consider the code in Fig. 3.7. There, the *private safe* function `unused` takes a shared reference argument. While in a fully safe program, this type annotation would imply that the target memory location is initialized, from unsafe code it is possible to call the function in a way such that the reference points to uninitialized memory. This is exactly what the `main` function does in the example, so that using the reference such as in the commented statement would cause undefined behavior. Supporting this kind of scenario, where some safe functions are called only from unsafe code, would require checking additional properties such as absence of undefined behavior and library soundness.

```rust
use std::mem::MaybeUninit;

fn unused(p: &u32) {
    // let v = *p;
}

fn main() {
    unsafe {
        let m = MaybeUninit::<u32>::uninit();
        unused(&*m.as_ptr());
    }
}
```

Rust

Figure 3.7: A Rust program where a safe private function is called with a shared reference that points to uninitialized memory.

The limitation to fully-safe Rust programs might seem quite restrictive at first, because real-world Rust programs often contain some unsafe code in their dependencies. However, the intended Rust way of working with unsafe code is to encapsulate it behind an API that exposes a safe and

sound API (cf., Sec. 2.1.5), and our technique is sound for safe clients of these libraries as well. In particular, we support annotating the libraries with *trusted* (that is, not verified) contracts, as if the implementation were entirely safe. This way, for example, it is possible to verify Rust programs that make use of common collection types of the standard library: vectors, sets, maps and so on. For example, in many libraries, the unsafe blocks in the implementation are there just for performance reasons and do not affect the overall functional behavior of the library API.

Unlike standard collections, libraries that provide interior mutability require some unsafe code in their implementation; a fully-safe implementation would not be able to provide the same API behavior. In this case, it is not possible to annotate the libraries using our technique to express the modifications that happen through shared references. Our verification technique remains sound in the presence of interior mutability, but at the cost of being incomplete. That is, each method call on a type with interior mutability would result in losing any knowledge about the content of the type. An example usage of interior mutability is in Fig. 3.8, where the `reset_cell` function modifies the content of `cell` via a function call on a shared reference. Our specification language in this chapter is not expressive enough to describe the content of `Cell`. For example, a postcondition such as `cell.get() == 0` would be rejected by our verifier with an error message stating that the `Cell::get` method is not pure and cannot be called from a contract. The only alternative, marking this method as trusted and pure, would be unsound in our technique because it would incorrectly model that the result of `cell.get()` cannot change across any call of `reset_cell(cell)`. We will describe in Ch. 4 a technique that overcomes this obstacle, introducing new annotations to model the memory locations modifiable via interior mutability and enabling verification of some of their usages.

```rust
fn reset_cell(cell: &Cell<u32>) {
    cell.set(0);
}
```

Rust

Figure 3.8: A Rust function using interior mutability to modify the content of a type passed as immutable reference.

3.3.4 Supported Rust Subset

Within safe code, the verification technique in this chapter works for a small but technically-challenging language subset. The supported types include primitive types (bool, integers, char) and the following compound types without a `Drop` implementation: boxes (for heap-allocated data), tuples, structs, enumerations, and generic type parameters. In addition, we support mutable and shared references to those types. Notably, this definition excludes structures with fields of type reference, which we do not support. Regarding usages of these types, the technique that we present — based on our OOPSLA 2019 paper [29] — supports functions with at most one lifetime and does not handle all usages of mutable references in loops, trait constraints on type parameters, nor pure functions returning non-primitive types. In addition to the paper, the technique that we present in this chapter supports loops with abrupt terminations (i.e., `break`, `continue`, and `return` statements) and non-pure loop guards.[4]

[29]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*

4: The most recent Prusti version available on GitHub [27] uses a snapshot technique not described in this chapter, which allows Prusti to support even more language features, such as generic types with trait constraints, pure functions with non-primitive return types, arrays and slices. This is described in our NFM'22 paper [31].

3.3.5 Trusted Computing Base

Our verification technique and tool build upon several assumptions:

- Our verification technique is defined on a compiler-provided CFG representation of Rust programs, called middle intermediate representation (`MIR`). Because of this, we assume that the translation from source code to MIR, referred to as "lowering" in Rust terminology and performed by the Rust compiler, is correct.
- MIR does not have a formal semantics. In our work, we rely on the available informal documentation of the semantics of MIR, as well as on tests that check the observable behavior of hand-written Rust programs.
- For the encoding of shared references (Sec. 3.6.1.3), we rely on the correctness of the borrow checker to determine when the shared references are created and when their associated lifetime ends.
- For the encoding of move assignments (Sec. 3.6.1.1), we rely on the correctness of the linearity checks of the compiler and assume that moved-out places are not read in a program. The alternative encoding proposed in Sec. 3.5.5 would remove this assumption.
- For the encoding of pure functions (Sec. 3.6.2), our verification tool assumes that the user-provided pure functions always terminate. This assumption could be removed by adding decreases-clause annotations on the Rust source code, encoding and verifying them as Viper's decrease clauses.
- The *implementation* of our verification tool has not been formally verified and is part of the trusted computing base. The tool might have bugs, independently of the correctness of the verification technique.

3.4 Rust Verification Annotations

In order to verify functional correctness, users of our verification technique need to declare the expected functional behavior of Rust functions and methods by annotating them with first-order logic contracts: preconditions, postconditions, and loop invariants. At their core, these contracts are side-effect-free boolean Rust expressions, augmented with quantifications and old expressions to refer to past program states. However, Rust has some commonly used language features that cannot be annotated, or are difficult to annotate, using just this standard specification language. In this section, we present the Rust-specific challenges and solutions that we developed.

3.4.1 Pledges

In Rust, the creation of a reference from another reference is called *reborrowing*. When the references are mutable, such as the creation of `q` in Fig. 3.9, the original reference `p` remains *blocked* until the last usage of the reborrowing reference `q`. From that point, called *expiration* of `q`, `p` can be used again. Since `p` and `q` refer to the same target memory location, any modification done via `q` will be visible via `p`.

```rust
fn reset_x(p: &mut Point) {
    let q: &mut i32 = &mut p.x;
    // p cannot be used here
    *q = 0;
    // q stops being used; from now on p can be used again
    assert!(p.x == 0); // always succeeds
}
```

Rust

Figure 3.9: Example of a reborrowing statement.

When reborrowing happens through a function boundary, meaning that the reborrowed reference is a function argument and the reborrowing reference is returned by the function, specifying the functional behavior in a modular way is not trivial. Consider for example the function `nth_mut` in Fig. 3.10, which returns a mutable reference to the nth pointer in a sequence `route` that is passed by mutable reference as well. On a call site like `let p = nth_mut(route, n);` the original `route` remains blocked until `p` expires. Until then, `p` can be used to make modifications that will persist past the expiration. Just by looking at the signature of `nth_mut`, one cannot get a precise understanding of the behavior of the function. For example, the signature does not prevent `nth_mut` from changing the length of `route`, nor prevents it from altering every point in the list. Moreover, the signature is clearly not enough to express at which position the returned reference is in `route`, and also does not express how the modifications that the caller may do via `p` will affect `route`. All of this needs to be expressed in a postcondition, but there are three main issues:

1. The standard semantics of postconditions is such that they are evaluated at the program point just *after* the function call, where the result is available and the function arguments can be expressed using `old(..)` expressions that are evaluated just *before* the function call. However, none of these program points (i.e., before and after the call) makes it possible to describe whether and how the call modified `route`. The reason is that just after the call `route` is blocked, and cannot normally be evaluated. The first program point where the value of `route` is unambiguously defined is later, at the point where `p` expires.
2. In order to express how modifications to `p` affect `route`, the postcondition of `nth_mut` needs to refer to the future program state where `p` expires. In particular, it needs to evaluate `p` just *before* the expiration point and `route` just *after* the expiration point. This is because, in order to respect the usual non-aliasing guarantees of mutable references, `p` and `route` should not appear in the same Rust expression.
3. Even if we broke the Rust rules to refer to both `p` and the blocked `route` from the same postcondition, there is one last issue: using at the expiration point the information defined in the postcondition regarding `p` and `route` is difficult. Such a postcondition would need to state precise aliasing properties, expressing that the returned `p` points to the memory location of the n-th node in the linked list. This kind of reasoning based on memory addresses is unnecessarily complex in many situations.

To solve the issues above, we designed a new functional specification annotation called a *pledge*, which makes it possible to express relations

that hold at the expiration point between the reborrowing and the reborrowed reference, without having to refer to memory addresses. In particular, such a relation is assumed to hold when verifying the caller and is verified on the callee side. To be sound, the latter verification step needs to be done considering *any* possible modification that might be done via the returned reference.

To see an example of a pledge specification, consider again Fig. 3.10. The `after_expiry` on `nth_mut` is a pledge annotation. The expression that it contains describes the relation between `route` and `result` that is supposed to hold at the program point where the result expires. Since in Rust, the result and the reborrowed `route` cannot coexist in the same expression, the `before_expiry(..)` expression can be used to refer to the program point just before the expiration, while `after_expiry` refers to the program point just after the expiration. When verifying `nth_mut`, the verifier will check that no matter how the result is used, the expression in the pledge is guaranteed to hold at the expiration point where `route` resumes being usable.

3.4.1.1 On-Expiration Condition

There are cases in which a reborrowing function requires the caller to leave the returned reference in a particular state. One example is a caller that needs to re-establish an invariant that might be temporarily broken while a borrow is alive. To make it possible to express such properties in a contract, we designed a generalization of the pledge annotation called `assert_on_expiry`.

The `assert_on_expiry(⟨A⟩, ⟨B⟩)` annotation takes two arguments. Both ⟨A⟩ and ⟨B⟩ are expressions that must hold at the point in time when the reborrowed reference expires. However, the difference is in who is responsible for guaranteeing that this is so. The first argument, ⟨A⟩, is the novelty of this annotation: a boolean expression that must be satisfied *by the caller*. The evaluation of this expression happens just after the expiration point, similarly to the evaluation of `after_expiry`. The second argument, ⟨B⟩, is the boolean expression that must be guaranteed *by the reborrowing function*. That is the same semantics of the argument of an `after_expiry`. In fact, any annotation of the form `after_expiry(⟨X⟩)` is equivalent to `assert_on_expiry(true, ⟨X⟩)`.

To see an example, consider the contract annotation of Fig. 3.11. Compared to the specification in Fig. 3.10, the contract uses an `assert_on_expiry` to require the caller of `nth_mut` to leave the route in a state where the `x` field of the returned `&mut Point` is non-negative. This condition makes it possible to re-establish the invariant that all `x` fields in the `route` are non-negative, as expressed by the additional `nth_x(route, i) >= 0` in the second argument of the `assert_on_expiry`. This way, a caller of `nth_mut` that assigns a negative value to its result will fail to verify with an error stating that, by the time the reference returned by `nth_mut` expires, the condition in the `assert_on_expiry` annotation of the reborrowing function has not been re-established.

```rust
struct Point {
    x: i32,
    y: i32,
}

struct Route {
  head: Point,
  tail: Option<Box<Route>>
}

#[pure]
#[ensures(result > 0)]
fn length(route: &Route) -> usize {
    1 + match route.tail {
        Some(ref tail) => length(tail),
        None => 0
    }
}

#[pure]
#[requires(0 <= n && n < length(route))]
fn nth_x(route: &Route, n: usize) -> i32 {
    if n == 0 {
        route.head.x
    } else {
        match route.tail {
            Some(ref tail) => nth_x(tail, n-1),
            None => unreachable!()
        }
    }
}

#[requires(0 <= n && n < length(route))]
#[ensures(result.x == old(nth_x(route, n)))]
#[after_expiry(
    length(route) == old(length(route)) &&
    nth_x(route, n) == before_expiry(result.x) &&
    forall(|i: usize|
        (0 <= i && i < length(route) && i != n) ==>
            nth_x(route, i) == old(nth_x(route, i))
    )
)]
fn nth_mut(route: &mut Route, n: usize) -> &mut Point {
    if n == 0 {
        &mut route.head
    } else {
        match route.tail {
            Some(ref mut tail) => nth_mut(tail, n - 1),
            None => unreachable!()
        }
    }
}
```

Rust

Figure 3.10: Example usage of the pledge annotation.

```rust
#[requires(0 <= n && n < length(route))]
#[ensures(result.x == old(nth_x(route, n)))]
#[assert_on_expiry(
    // Condition:
    nth_x(route, n) >= 0,
    // After expiry pledge:
    length(route) == old(length(route)) &&
    nth_x(route, n) == before_expiry(result.x) &&
    forall(|i: usize|
        (0 <= i && i < length(route) && i != n) ==>
            nth_x(route, i) == old(nth_x(route, i)) &&
            nth_x(route, i) >= 0
    )
)]
fn nth_mut(route: &mut Route, n: usize) -> &mut Point {
    ...
}
```

Rust

Figure 3.11: Example usage of the `assert_on_expiry` pledge annotation.

```rust
#[requires(0 <= n && n < length(route))]
fn bad_reset_nth_x(route: &mut Route, n: usize) {
    let p = nth_mut(route, n);
    p.x = -1; // Error: unsatisfied pledge of nth_mut
}
```

Rust

Figure 3.12: Example of a client of `nth_mut` that violates the `assert_on_expiry` annotation of Fig. 3.11.

3.4.1.2 Alternative At-Expiration Formulation

When declaring pledges, we used `before_expiry` and `after_expiry` to distinguish between the program point just *before* and just *after* the expiration. However, this syntax could potentially be simplified. An expiration is a ghost operation that is used only to explain the behavior of the borrow checker. That is, it does not compile to any runtime operation. There are no differences between the values observable at these two program points; what changes are just the expressions that can be used to refer to them. Based on this observation, and also noting that contract expressions are by definition side-effect free, we designed an alternative formulation of the pledge annotation by which there is no distinction between the program point before and after the expiration. Instead, the pledge is evaluated *at* the expiration point assuming that, for a brief moment, both the reborrowing and the reborrowed references are usable in a read-only mode. The example in Fig. 3.13 shows what the new pledge annotation would look like. Overall, it is syntactically simpler and potentially easier to grasp. Reasoning about immutable references at the same program point may require less mental overhead than reasoning about references evaluated at more distant program points. However, there are also downsides. To support this new annotation the tool implementation has to do more work to recognize what in the expression should be type-checked before and what after the expiration point. Moreover, mixing expressions that in Rust are valid only before or after the expiration point would violate the principle by which contracts should contain valid Rust expressions.

As an additional simplification one can observe that, since the reborrowed arguments are unusable just after the call, any usage of them in a postcondition should implicitly be evaluated as if they were in an

```rust
#[requires(0 <= n && n < length(route))]
#[ensures(result.x == old(nth_x(route, n)))]
#[at_expiry(
    length(route) == old(length(route)) &&
    nth_x(route, n) == result.x &&
    forall(|i: usize|
        (0 <= i && i < length(route) && i != n) ==>
            nth_x(route, i) == old(nth_x(route, i))
    )
)]
fn nth_mut(route: &mut Route, n: usize) -> &mut Point {
    ...
}
```

Rust

Figure 3.13: Example usage of the `at_expiry` syntax of pledges.

`at_expiry(..)` annotation. Syntactically, this makes pledges look exactly like postconditions when they do not refer to `result`, but without causing ambiguities. The reason is that since the reborrowed reference is blocked and unusable on the call site until the expiration point, and since the pledge is checked to hold no matter how the reborrowing reference is used, a pledge that does not refer to `result` can be seen as holding *at any point* between the end of the reborrowing function and the expiration point. The postcondition, evaluated just after the reborrowing function, is just a special case of the pledge. So, unifying the syntax of pledges and postconditions is possible. The example in Fig. 3.14 shows what the new pledge annotation would look like.

```rust
#[requires(0 <= n && n < length(route))]
#[ensures(result.x == old(nth_x(route, n)))]
#[ensures(length(route) == old(length(route)))]
#[ensures(nth_x(route, n) == at_expiry(result.x))]
#[ensures(forall(|i: usize|
    (0 <= i && i < length(route) && i != n) ==>
        nth_x(route, i) == old(nth_x(route, i))
))]
fn nth_mut(route: &mut Route, n: usize) -> &mut Point {
    ...
}
```

Rust

Figure 3.14: Example of a pledge specification expressed in a postcondition.

In this chapter, we evaluated and implemented only the former pledge definition, based on `before_expiry` and `after_expiry`. However, we think that these alternative `at_expiry` formulations can be beneficial to future work.

3.4.2 Loop Body Invariants

When verifying code that contains loops, the standard approach in Hoare logic is to build an inductive proof using a *loop invariant*, which is a property that holds when entering the loop and after each loop iteration. The classical definition, in Fig. 3.15, requires the guard B of the loop to be a boolean expression with a side-effect-free evaluation. This makes it possible to directly embed B in the program logic used for the proof, by assuming that $\neg B$ holds after the loop in addition to the invariant.

The classical definition of loop invariants works well for simple Rust

$$\frac{\{P \wedge B\}\, S\, \{P\}}{\{P\}\ \texttt{while}\ B\ \texttt{do}\ S\ \texttt{done}\ \{\neg B \wedge P\}}$$

Figure 3.15: Hoare-logic rule for loops.

loops such as the one in Fig. 3.16, where the guard is `i < 10` and the loop invariant sufficient to verify the assertion after the loop is `i <= 10`. In this case, the CFG representation is quite simple: the block ① is the loop head, is side-effect free, and is also the origin of the *exit edge* corresponding to the negation of the loop guard. The loop invariant, in particular, holds on all edges that arrive at the loop head ①.

```rust
fn main() {
    let mut i = 0;
    // Loop invariant: i <= 10
    while i < 10 {
        i += 1;
    }
    assert!(i == 10);
}
```

Rust

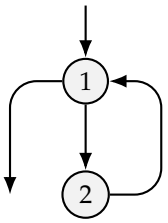


Figure 3.16: Simple while loop.

However, loop guards in Rust are not always side-effect free and the loop may contain `break`, `continue` or `return` statements. A common example of guards with side effects are iterators, for which we provide an example in Fig. 3.17. The `main` function first declares an iterator that generates values from 0 to 9, which are counted in the loop. The special syntax `while let Some(..) = ..` is Rust's syntax sugar to break the loop when the result of the `next` call is `None`. A loop invariant alone, such as the conjunction of `i == range.from`, `range.from <= range.to`, and `range.to == 10`, is not enough to verify `i == 10` after the loop. It is also necessary to know that after the loop `iter.from >= iter.to`, which is implied by the postcondition of `next` when returning `None`. Since the loop guard is not pure, we cannot directly use the standard Hoare rule in Fig. 3.15. Manually rewriting the loop so that the guard is side-effect free or so that the loop does not contain `break`, `continue`, or `return` statements is possible, but it would make both the source code and the loop invariant more complex. A better option is to perform the rewriting automatically in the verifier, transparently to the user. However, for a given loop there are many different possible rewritings, depending on what the new loop guard should be and at what point in the loop the invariant should hold. Since this choice affects the semantics of the invariant, the user should have control over it. As we are going to present, our technique performs the loop rewriting automatically, based on a user choice of where to place the invariant. For example, before the evaluation of the loop guard, immediately after the guard's evaluation, or even at an intermediate state during the execution of the loop body. The novelty of our work is not in rewriting the loop but in allowing the user to flexibly choose where to place the invariant, which in turn implicitly determines the loop rewriting used by the verifier. Recent work on verification of C programs use loop invariant annotations with a similar degree of flexibility [60].

[60]: Zhou et al. (2024), *VST-A: A Foundationally Sound Annotation Verifier*

To handle the general case of loops, we designed a *loop body invariant* annotation, `body_invariant!`, that makes it possible to flexibly declare

```rust
struct IterRange {
    from: usize,
    to: usize,
}

#[requires(iter.from <= iter.to)]
#[ensures(iter.to == old(iter.to))]
#[ensures(old(iter.from < iter.to) ==> iter.from == old(iter.from) + 1)]
#[ensures(old(iter.from < iter.to) == matches!(result, Some(_)))]
fn next(iter: &mut IterRange) -> Option<usize> {
    if iter.from < iter.to {
        let v = iter.from;
        iter.from = v + 1;
        Some(v)
    } else {
        None
    }
}

fn main() {
    let mut range = IterRange { from: 0, to: 10 };
    let mut i = 0;
    while let Some(_) = next(&mut range) {
        i += 1;
        body_invariant!(i == range.from && range.from <= range.to && range.to == 10);
    }
    assert!(i == 10);
}
```

Rust

Figure 3.17: While loop traversing an iterator.

an invariant at any non-conditional point in a loop body. The semantics is similar to the one of classical invariants, but generalized. First, the loop body invariant should hold when reaching the invariant for the first time. Second, assuming that an invariant already holds at a generic loop iteration, it should also hold when the invariant is reached again in the next loop iteration. Third, the invariant should be specified in a *non-conditional* point that cannot be avoided when executing a loop iteration. As a result, when reasoning about the code that comes after the loop, the verifier can assume that the loop body invariant was either holding the last time it was reached, or that the loop body was never executed at all. Note how this definition does not restrict in any way the expression of the loop guard and does not limit the usage of `break` and `return` statements. This semantics of the invariant closely matches the evaluation of a runtime assertion, meaning that in the case of simple invariants, it is possible to switch between verified and runtime-checked invariants by replacing `body_invariant!` with `assert!`, or the other way around.

To see an example, consider again the loop in Fig. 3.17. Instead of rewriting the loop to have a side-effect-free guard, which would require manually desugaring Rust's `while let` syntax, the invariant is specified with a `body_invariant!` annotation placed at the end of the loop body. First, the verifier checks that the body invariant holds the first time it is reached, when `i == 1`. Then, it checks that the invariant is preserved across one loop iteration. That is, after assuming the invariant and executing `next(&mut range)`, if the result is `Some(_)`, after incrementing `i` by

one the invariant should hold again. The position of the invariant is legal because there are no `if` or `continue` statements that can skip over the `body_invariant!` going to the next loop iteration. As a result, the verifier can prove the assertion after the loop, by using the information that the invariant was holding before the last execution of `next(&mut range)`, which returned `None`.

The choice of where to place the loop invariant is up to the user, who can choose what is most convenient case by case. In Fig. 3.17 the invariant is placed at the end of the loop, but an equivalent invariant could also be expressed at the beginning of the loop body by shifting the value of `i` by one: `body_invariant!(i == range.from - 1 && ...)`. As an extreme, the loop invariant can even be placed at the beginning of the loop head, just before the evaluation of the loop guard. This way, demonstrated in Fig. 3.18, the semantics of the loop body invariant matches exactly the semantics of the classical loop invariant, although the syntax of the resulting program may seem unusual.

```rust
fn main() {
    let mut i = 0;
    while {
        body_invariant!(i <= 10);
        i < 10
    } {
        i += 1;
    }
    assert!(i == 10);
}
```

Figure 3.18: While loop a body invariant annotation. This is the annotated version of Fig. 3.16.

3.5 Viper Encoding of Types

The approach of our verification technique requires modeling several properties of the Rust type system in separation logic, which we do using the Viper verification language. This section presents our encoding of Rust types in Viper.

At a high level, we want our technique to model memory safety, non-aliasing and immutability properties of the Rust types. These properties should be checked during verification so that our technique is ready to be extended to cases of unsafe code, making it possible at the same time to rely on these properties in the proof when reasoning about function calls, aliasing and mutations. Since the term *ownership* in Rust is already used to describe the memory locations that must be deallocated when the local variable that owns them goes out of scope, in this chapter we use the term *capability* to define an abstraction that captures the properties of the Rust language that we are interested in. In particular, we use two kinds of capabilities, *shared* and *exclusive*, which intuitively correspond to the properties of shared and mutable references, respectively. A shared capability guarantees that the reachable memory locations are immutable and can safely be read, while an exclusive capability ensures that the reachable memory locations can be safely read and written, but only via the Rust expression that is holding such capability. In fact, capabilities are statically associated with syntactic *places*, which are Rust expressions that

can be constructed starting from local variables by repeatedly accessing a field, following a reference, or downcasting an enumeration. Like mutable references in Rust, an exclusive capability is non-duplicable and can be temporarily split into duplicable shared capabilities. In this chapter, we use a capability definition that is transitive, meaning that if the local variable `x` has an exclusive capability for the place `x.f`, and if `x.f` has an exclusive capability for `x.f.g`, then `x` has an exclusive capability for `x.f.g`. Moreover, shared capabilities have precedence over exclusive ones when they are chained. For example, if `y` has a shared capability for `y.f` and `y.f` has an exclusive capability for `y.f.g`, then `y` has a shared capability for `y.f.g`. Overall, these capabilities are enough to express the type properties that we are interested in:

1. Memory safety: holding a capability guarantees that the corresponding memory location is initialized, safe to read and free of data races. If the capability is exclusive then the memory location is also safe to write to.
2. Immutability: holding a shared capability guarantees that the corresponding memory location is immutable.
3. Non-aliasing: holding an exclusive capability guarantees that there are no usable places that alias the same memory location.

In Viper, capabilities can be modeled as resources with fractional permissions. An exclusive capability corresponds to a full fractional permissions, which implies absence of aliases and provides read and write access to the holder of the permission. A shared resource, instead, can be modeled as an infinitesimal non-zero fractional permission, which can be aliased and provides read access and immutability. This is the core on which the whole encoding is built upon.

3.5.1 Non-Reference Rust Types

One challenge of modeling Rust types is that the capability of a type instance might have to describe an unbounded number of memory locations. Take for example the linked-list definition in Fig. 3.19, which is a recursive data type composed of a generic field *value* and a field `next` pointing to the next node in the list. The `Option` in the type definition is an *enumeration*; a sum type akin to the `Maybe` type in Haskell. Finally, the `Box` type is an owning pointer used to allocate the target instance on the heap. When holding an instance of `List`, there is no static limit to the number of memory locations that might be reachable from there.

```rust
struct List<T> {
    value: T,
    next: Option<Box<List<T>>>,
}
```

Rust

Figure 3.19: Type declaration of a linked list in Rust.

Viper makes it possible to model an unbounded set of memory locations using recursive predicates. We model the Rust types after monomorphization, so the Viper encoding contains one predicate definition for each monomorphized Rust type definition. Each predicate is parameterized by a Viper reference type, `Ref`, which models in Viper the base address in Rust of an instance of the type. To encode Rust structures, the corresponding Viper predicate is defined as containing several Viper fields

of type `Ref`, one for each field of the Rust structure. This models the memory address of each field. For each field, the predicate declares (1) to have full permission to the field, so that it can be read and even modified when encoding moves, and (2) to hold another predicate instance that encodes the content of the field. This way, each predicate definition can be constructed using only the information available in a single Rust type declaration. Tuples are encoded following the same recipe, as if they were anonymous structures.

As an example, consider the Rust type `List<u32>`, whose Viper encoding is shown in Fig. 3.20. The figure just shows a part of the encoding; what is not yet visible is the encoding of the `Option` field, which internally contains another instance of the `ListU32` predicate to encode the next node in the list. As defined earlier, the `ListU32` predicate contains the full permission for two fields, expressed by the `acc(..)` expressions, as well as a predicate instance of `U32` or `OptionBoxListU32` to encode the Rust type of the fields.

```
field value_field: Ref
field next_field: Ref

predicate ListU32(self: Ref) {
    acc(self.value_field) && U32(self.value_field) &&
    acc(self.next_field) && OptionBoxListU32(self.next_field)
}
```
Viper

Figure 3.20: Viper encoding of the `List<u32>` type.

The `Option` type is defined in the standard library as an enumeration composed of two variants, as shown in Fig. 3.21: a `Some` variant that contains an instance of the type parameter `T`, and a `None` variant that is empty. When compiling enumerations, the Rust compiler automatically generates a *discriminant* field under the hood to represent at runtime which variant is stored in an instance of the enumeration[5], similar to the tagged unions idiom in C. When modeling enumerations in Viper, the encoding uses a `discriminant` integer field constrained to the same range of values of the Rust discriminants. Then, depending on the value of the discriminant, the predicate definition declares to hold the permission and predicate instances corresponding to the fields of each variant.

5: In some cases, the Rust compiler takes advantage of invalid bit patterns (*niche values* [61]) to represent the discriminant using the existing fields of the type. In our work, this optimization does not matter because we always model the discriminant as a separate field.

```
enum Option<T> {
    None     // discriminant: 0
    Some(T), // discriminant: 1
}
```
Rust

Figure 3.21: Rust definition of the `Option<T>` type.

As an example, Fig. 3.22 shows the Viper encoding of `Option<Box<List<u32>>>`. Since the enumeration has two variants, the Viper discriminant is constrained to be either 0 or 1. Then, in case the discriminant is 1, an implication in the predicate definition encodes the fields that compose the `Some` variant in the same way they were encoded for structures. Since the `None` variant has no fields, the predicate definition does not need to define anything more for the case where `self.discriminant == 0`.

To complete the definition of the encoding of `List<u32>`, we still need to define the ending of `Box` and of primitive types such as `u32`. `Box<T>` is

```
field discriminant: Int
field some0: Ref

predicate OptionBoxListU32(self: Ref) {
    acc(self.discriminant) &&
    0 <= self.discriminant && self.discriminant <= 1 &&
    (self.discriminant == 1 ==>
        acc(self.some0) &&
        BoxListU32(self.some0))
}
```

Viper

Figure 3.22: Viper encoding of the `Option<Box<u32>>` type.

a type that represents a pointer to some heap-allocated data. Its definition internally uses raw pointers, but Rust special-cases the compilation of this type so that from safe Rust code, the type looks like a simple wrapper around an instance of `T`. In our Viper model, we do not distinguish between data structures on the stack or on the heap. So, like the Rust compiler, our technique special-cases the `Box` type, encoding it as if it were a regular Rust structure with just one field. An example for `Box<List<u32>>` is shown in Fig. 3.23. Note that by using `ListU32`, the predicate definition becomes recursive, passing through all the predicates used in the examples above. For primitive types, our encoding defines a predicate that contains one integer Viper field, as well as an encoding of the value range of the Rust integer. In the case of `u32`, a 32-bit unsigned integer, the Viper encoding is shown in Fig. 3.23. Other integer primitive types are encoded in a similar way, but using different value ranges. As for the Rust `bool` type, its `Boolean` encoding shown in Fig. 3.23 does not need any value range.

```
field target: Ref
field val_int: Int
field val_bool: Bool

predicate BoxListU32(self: Ref) {
    acc(self.target) && ListU32(self.target)
}

predicate U32(self: Ref) {
    acc(self.val_int) &&
    0 <= self.val_int && self.val_int < 2^32
}

predicate Boolean(self: Ref) {
    acc(self.val_bool)
}
```

Viper

Figure 3.23: Viper encoding of `Box<u32>` and primitive Rust types.

3.5.2 Reference Types

As introduced earlier, mutable and shared Rust references can be modeled as exclusive and shared capabilities, which in Viper correspond to full and infinitesimally small fractional permissions, respectively. Since in Rust it is possible to create references that point to other references, in our Viper predicate encoding we still model the address of the reference instance with a Viper `Ref` parameter, the address of its *target* instance with a `target` field, and the content of the target with another predicate

instance. For the case of mutable references, Fig. 3.24 shows the encoding of a `&mut List<u32>` type.

The encoding of shared references is similar, but slightly complicated by the fact that Viper does not natively support infinitesimally small fractional permissions. Instead, in our technique we model them by using an uninterpreted function `rd()`, axiomatized to be strictly between zero and one. By using `rd()`, we can then model that a shared reference holds only a fractional permission amount `rd()` for its target predicate instance, modeling a shared and not an exclusive capability. This is expressed by the Viper syntax `acc(ListU32(self.target), rd())` in the example of Fig. 3.25, which shows the encoding of the `&List<u32>` type. The field `self.target` that models the address of the target instance is still owned with a full permission by the predicate of the shared reference, because in Rust it is legal to change which target a shared reference points to, without modifying the content of the old or the new target instance.

When building on top of the axiomatization of `rd()`, reasoning with usages of `rd()` sometimes hits incompletenesses in the Z3 SMT solver used internally by Viper. The reason is that non-linear arithmetic is undecidable, and sometimes the solver is not able to prove simple properties such as `0 < rd() * rd()` or `rd() * rd() < rd()`. To mitigate those incompletenesses, in our experiments we found it useful to provide these two properties as axioms, as shown in Fig. 3.25.

```
predicate MutRefListU32(self: Ref) {
    acc(self.target) && ListU32(self.target)
}
```
Viper

Figure 3.24: Viper encoding of the `&mut List<u32>` type.

```
domain RdPerm {
    function rd(): Perm
    axiom { none < rd() && rd() < write }
    axiom { none < rd() * rd() && rd() * rd() < rd() } // Hint
}

predicate ShrRefListU32(self: Ref) {
    acc(self.target) &&
    acc(ListU32(self.target), rd())
}
```
Viper

Figure 3.25: Viper encoding of the `&List<u32>` type. In Viper, `none` = 0 and `write` = 1.

3.5.3 Borrowed Types

When a reference is created, it borrows its capabilities from an existing place. This leaves a *hole* in the capabilities of the data structure from which the reference is borrowing, meaning that the predicates that we defined above are not suitable for describing the remainder capabilities "with a hole". For example, consider the `value_mut` function in Fig. 3.26. The capabilities that the function receives from the caller can be modeled by a `MutRefListU32` predicate instance, but what about the capabilities that the function *returns* to the caller? Returning just a `MutRefU32` instance for the return type is not enough, because in Rust a client such as `set_all` can resume using the borrowed argument after the returned reference expires, like it happens in the example after the `*v = new_val`

statement. This means that we need a way to model, and return to the caller, the capabilities that are left after the creation of a reference such as `&mut x.value`.

```rust
fn value_mut(x: &mut List<u32>) -> &mut u32 {
    &mut x.value
}

fn set_all(x: &mut List<u32>, new_val: u32) {
    let v = value_mut(x);
    *v = new_val;
    if let Some(ref mut tail) = x {
        set_all(tail, new_val)
    }
}
```

Rust

Figure 3.26: Example of a reborrowing function. The implementation of `value_mut` returns a reference created from a reference argument.

When the reborrowing involves mutable references, as in the case of the argument and return type of `value_mut`, we model the remainder capabilities using the *magic wand* connective of separation logic. Informally, a magic wand $A \mathrel{-\!\!*} B$ is a resource that testifies that there is a way to obtain the resource B by consuming the resource A and the magic wand itself. This matches what we need to express in the postcondition of reborrowing functions such as `value_mut`. In Viper, we can express a magic wand that represents the capabilities that, if conjoined with the content of the `MutRefU32` instance representing the returned reference, can generate a `ListU32` instance that models the capabilities that should be restored when the mutable reference argument expires. The encoding of the capabilities in the contract of `value_mut` can be seen in Fig. 3.27.

```
method value_mut(x: Ref) returns (res: Ref)
    requires MutRefListU32(x)
    ensures MutRefU32(res)
    ensures U32(res.target) —* ListU32(x.target)
```

Viper

Figure 3.27: Viper encoding of the capabilities of a reborrowing function.

3.5.4 Generic and Unsupported Types

Predicates, in Viper, are not required to have a definition. In that case, the predicate is called *abstract* to suggest that any proof involving them would work for any possible definition of the abstract predicates. This is a convenient feature that we use in our technique to verify functions with type parameters, or functions that use some type (e.g., `String`) whose declaration uses unsupported Rust features (e.g., raw pointers) that can be hidden as implementation details without affecting the specification of the public API. In the case of instances of type parameters, our technique encodes them as abstract predicates when verifying the Rust function where they are declared. When reasoning about calls of such functions, where the type parameters are instantiated by concrete Rust types, our technique replaces the abstract predicates with the predicates encoding the concrete Rust types. In the case of unsupported types, our technique encodes them as abstract predicates, so that the proof around them can proceed with the guarantee that it will not depend on the content of the unsupported types.

3.5.5 Alternative Allocation-Based Encoding

The encoding technique described in this chapter essentially uses permissions to model initialized memory locations. That is, a statement that deinitializes some memory, such as a move assignment, is encoded so that after the statement there are no permissions in Viper for the deinitialized memory location. This choice is necessary in order to verify full memory safety, which has safety advantages but also technical disadvantages. For example, to encode a move assignment such as `a = b` our technique has to handle the case where it is statically not known whether `a` was initialized before the statement or not. So, it is more difficult to generate an encoding (as we describe in Sec. 3.6.1.1) that is correct in both cases.

There exist alternative encodings that may be easier to generate. One such encoding is based on the idea of using permissions to model *allocated* memory locations, instead of initialized ones. Under this model, the guarantees of the verification are weaker because accessing a non-initialized memory location would raise no verification errors, while trying to access deallocated memory locations would still be rejected. The expected advantage is that the encoding should be easier to define, because gaining and losing permissions would essentially correspond 1:1 to allocation and deallocation machine operations (e.g., pushing an element on the stack, or allocating a new region on the heap) that the compiler represents in the CFG of a function.

3.5.6 Alternative Instance-Identity Based Encoding

When designing the encoding of move assignments of non-copy types, we observed an interesting invariant: across a move, the *address* at which the type instance is stored might change, but the type instance is intuitively still *the same*. Because of this, linear types can be considered as having an intrinsic *identity* characteristic, which copy types do not have. Being able to refer to the identity of a non-copy type instance from specifications has been shown to enable specification techniques that use Rust types to manage separation-logic resources [62], making the specification language more expressive. Moreover, leveraging this instance identity, it might be possible to design a Viper encoding that is particularly concise and efficient in modeling move assignments.

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

Modeling an instance identifier can be done in several ways. The first way, which builds on top of the technique presented in this chapter, is to implicitly add a special `id` field to all Rust types during the encoding to Viper, so that this special field cannot be used in Rust code but can be mentioned only in the specifications. This field should be initialized with an unconstrained value at the moment of the creation of an instance, and should always be moved together with the type instance to a new memory location when modeling a move assignment. This would make it possible to expose the identity of non-copy instances in the specifications, but would not bring performance advantages to the verification.

The second way is to change the conceptual meaning of the `Ref` values in Viper, so that they model the identifier of a type instance instead of memory locations. The encoding of types into predicates would mostly remain the same, because interpreting the `Ref` parameter of the

predicates as being the identifier of an instance mostly works, and would only require changes to the magic wands that encode borrowed types and to the encoding of Rust assignments (presented later, in Sec. 3.6.1.1). An advantage of this approach is that move assignments would be easy to model, because they would only need to assign a Viper `Ref` from one place to another instead of using the encoding described in Sec. 3.6.1.1. Because of this, the Viper backend that uses symbolic execution might have a greater chance of verifying the Viper program based on syntactic properties instead of having to resort to using a (powerful but slow) SMT solver, potentially greatly increasing the performance of the verification. A disadvantage is that in this encoding it would not be possible to reason about memory locations, making it more difficult to tackle verification of unsafe code in future work.

A third option is to keep the current encoding that uses `Ref` types to model memory addresses, but insert a new `.instance_id` field in between the `self` parameter and the content of each predicate to model the identifier of the instance. This way, considering a particular program point, the content of the type is modeled as being a function of the instance identifier, which is in turn a function of the address of the reference. The `.instance_id` field would model allocation, while the existing Viper fields would still model initialization. The advantage of this approach is that in this encoding move assignments can be easily (and efficiently) encoded as a single assignment between the `.instance_id` field of the source and target instance as before, while still being able to reason about memory locations from specifications. An example of the predicate encoding a `List<u32>` type with this technique is in Fig. 3.28.

```
field instance_id: Ref
field value_field: Ref
field next_field: Ref

predicate ListU32(self: Ref) {
    acc(self.instance_id) &&
    acc(self.instance_id.value_field) &&
    U32(self.instance_id.value_field) &&
    acc(self.instance_id.next_field) &&
    OptionBoxListU32(self.instance_id.next_field)
}
```

Viper

Figure 3.28: Alternative Viper encoding of the `List<u32>` type.

3.6 Viper Encoding of Procedures

In this section, we describe how our technique models Rust procedures (i.e., functions and methods) in Viper. At a high level, our technique models Rust functions in two ways: a functional *pure* encoding and an imperative *impure* one. All procedures are modeled using the imperative style, but only the procedures that are guaranteed to be deterministic and side-effect free can additionally be modeled using the functional style. To opt-in into this second kind of encoding, a verification user has to mark the procedure using the `#[pure]` attribute, which checks the desired properties by conservatively enforcing several language limitations. For example, pure functions cannot have mutable reference parameters, loops, or calls to other non-pure functions.

3.6.1 Imperative Encoding

The main goal of the imperative encoding is to generate a Viper program that verifies only if the source Rust program respects all properties checked by the verification technique (e. g., absence of panics, absence of integer overflows and divisions by zero, functional correctness). A first challenge when working with a real-world programming language is that the grammar of the language is *large*, and defining an encoding starting directly from the syntax of the language might not be feasible, or not possible if the grammar has not been formalized. Moreover, the encoding is trusted and not formally verified, so there might be the risk of misinterpreting the semantics of a Rust program. To address both challenges, we designed our encoding to remain as close as possible to the compiler-provided CFG representation of Rust programs, called middle intermediate representation (MIR), whose definition is much simpler than the grammar of the language or its abstract syntax tree representation. In order to make the technique easier to maintain, modify, and reuse, a second goal of our design is to be as modular as possible, meaning that all statements in the MIR representation should be encoded independently. This way, a Rust procedure can (almost entirely) be modeled as the concatenation of the simpler encoding of its statements.

At a high level, the MIR representation of a function is a CFG where the nodes are called basic blocks. Each basic block contains a sequence of consecutive statements, the last of which can be a goto or a switch statement, whose jumps lead to the beginning of basic blocks. Among the statements, the most important are assignments. The left-hand side of an assignment must be a `Place`, which is a local variable followed by a sequence of projection operations such as field accesses, dereferences and downcasts of enumerations. The right-hand side can be either a function call, a usage of a `Place` (a *move* or, if the type allows it, a *copy*) and other operations such as integer arithmetic. This list is not exhaustive and overall this MIR definition is simpler than the real one, but it is nevertheless sufficient to present the core of the encoding.

Our technique encodes Rust procedures into Viper methods, modeling the structure of the CFG using `if` and `goto` Viper statements. At the beginning of the Viper method, the encoding models the capabilities that the caller passes to the callee. This is done by assuming the predicate instance that corresponds to the capabilities of the Rust arguments. After that, the encoding assumes the precondition of the procedure and then encodes the body of the procedure. Finally, the method checks at its end that the postcondition of the procedure holds, and that the method actually has the permission for the predicate instances of the capabilities that should be returned to the caller.

As an example, the diagram in Fig. 3.29 shows a simplified version of the encoding of the `force_inc` function. There, we can see that the parameters are encoded as `Ref` arguments for which the proof *inhales* the resource of a predicate, modeling its creation: `MutRefRoute` for `&mut Route` and `I32` for Rust's `i32`. The encoding from types to predicates follows what presented in Sec. 3.5. Since `force_inc` does not have a precondition, the encoding uses `true` as default precondition. The CFG of the procedure contains a single switch statement, which is directly modeled in Viper with an `if` statement. In this case, the

branch condition depends on the discriminant of the Rust expression `r.tail`, which Rust internally desugars to the `Place` `(*r).tail`. Rust dereferences and field accesses are encoded to Viper field access, such as `.target` for dereferencing `r` and `.some0` for accessing the only element of the `Some` variant of the expression `*r`. Note that Viper, based on the predicate definition of the `Option` type of `*r`, automatically checks that the `.some0` field is only accessed in a program path where the discriminant of the enumeration is known to be 1, i.e., the `Some` variant. Moreover, every field access such as `.tail` is checked to be only performed when the method holds a permission for the field, which in our encoding technique models that the Rust field has been initialized.

3.6.1.1 Encoding of Assignments

One of the most common kinds of MIR assignment is those whose right-hand-side is a usage (move or copy) of a `Place`. Since our verification technique does not aim to verify absence of memory leaks nor functional properties of Rust addresses (e.g., equality between the target addresses of two shared references), in our encoding we have the freedom of modeling new allocations whenever we want. This gives an advantage when modeling initialization using Viper permissions, because whenever a statement assigns to a place that cannot statically be determined to be surely initialized, the encoding can model that the statement allocates a new memory location with a new address instead of overwriting the existing one. Consider for example the encoding in Fig. 3.29. The assignment to `q` and the temporary variables introduced by the compiler `arg1` and `arg2` are modeled by assuming with an `inhale` statement that the statement allocates a new memory location, instead of potentially reusing an existing but no longer used memory location on the stack. When encoding the assignment to `r.head.x`, however, the left-hand-side place is statically known to be already initialized, so our technique can model the assignment more efficiently, simply overwriting the memory location represented by the Viper expression `r.target.head.x`.

Another common kind of MIR assignment is the one whose right-hand side is a function call. Our encoding differs depending on whether the called function is pure or not. In case it is pure, the encoding of the call is a single Viper assignment which calls on the right-hand side a functional encoding of the callee, which we present in the next section. This is the case of the call to `max` in Fig. 3.29. In case the called function is not pure, the encoding is done based on the signature and contract of the callee. More precisely, the encoding checks that it holds the predicate instances modeling the capabilities of each of the arguments of the call, and that the precondition of the callee holds. Then, it consumes all those predicate instances to model that they were passed to the callee. These three steps correspond to the `exhale` statement in the encoding of `force_inc`. Then, the encoding models that the callee transfers back some capabilities to the caller, and that the postcondition in the contract holds. These two steps correspond to the `inhale` statement in the encoding of `force_inc`. The additional `label call_pre` Viper statement in the encoding is used by the functional encoding of contracts. This way, the `old(..)` expressions in the contracts can be encoded

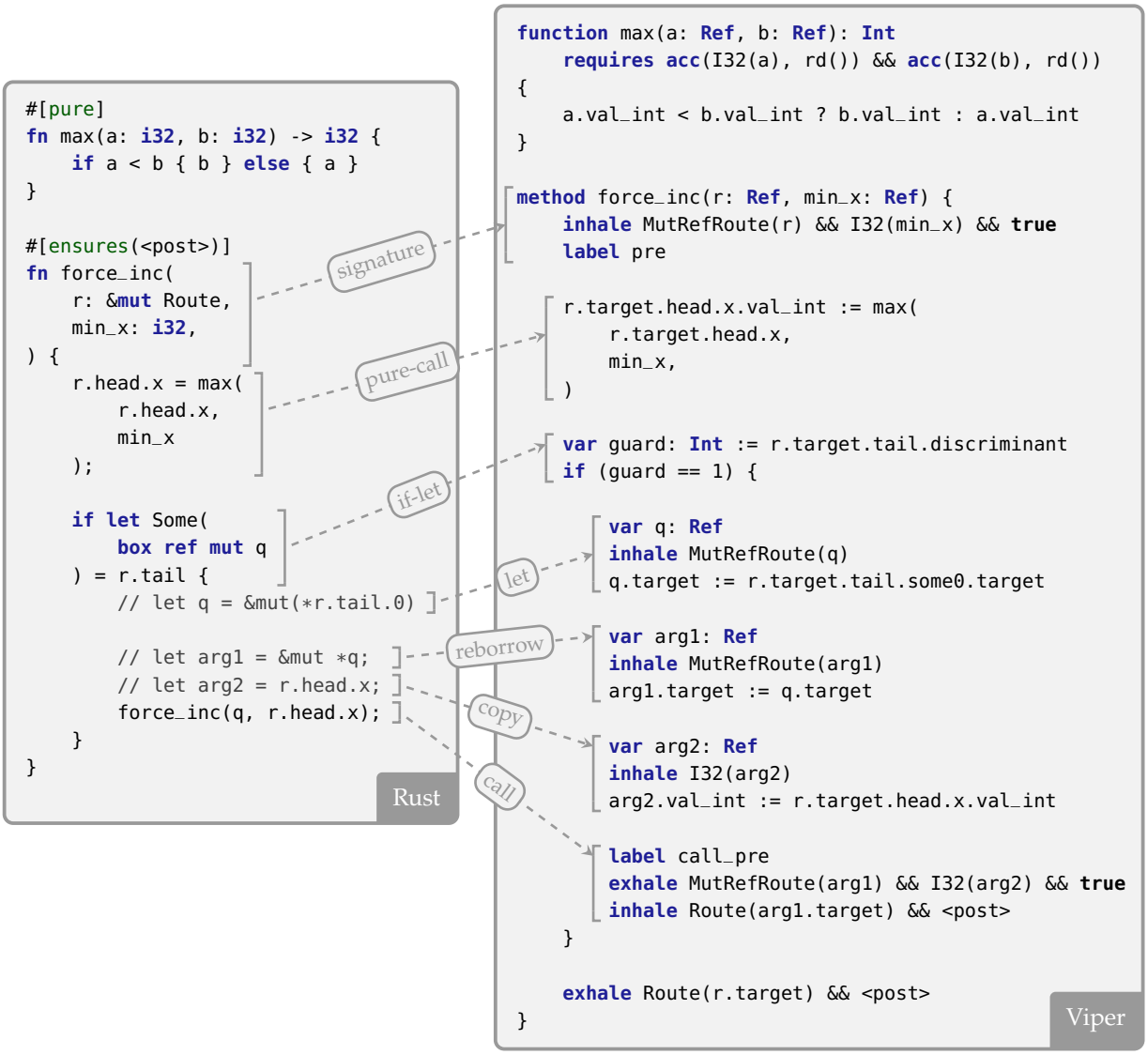


Figure 3.29: Imperative Viper encoding of a Rust function.

to a Viper `old[call_pre](..)` expression, which evaluates a Viper expression before the call, where the `call_pre` label is declared.

3.6.1.2 Encoding of Loops

The main idea behind our encoding of loops is that the loop invariant does not need to be placed at the head of the loop, because at its core an invariant is just a property that has to hold whenever the execution reaches the program point of the invariant annotation[6]. What is important for verification models is to encode, in some way, an unconstrained number of loop iterations using a finite number of modeling statements in loop-free code. Our finding is that this can be done even when the invariant is placed at various places in the middle of the loop body, using a single semantics-preserving CFG transformation. The only requirement is that the invariant annotation should be in a non-conditional path of the execution of the loop body, so that it is not possible to complete a loop iteration without passing through the invariant annotation. With this technique, we can verify loops using the loop body invariant annotation presented in Sec. 3.4.

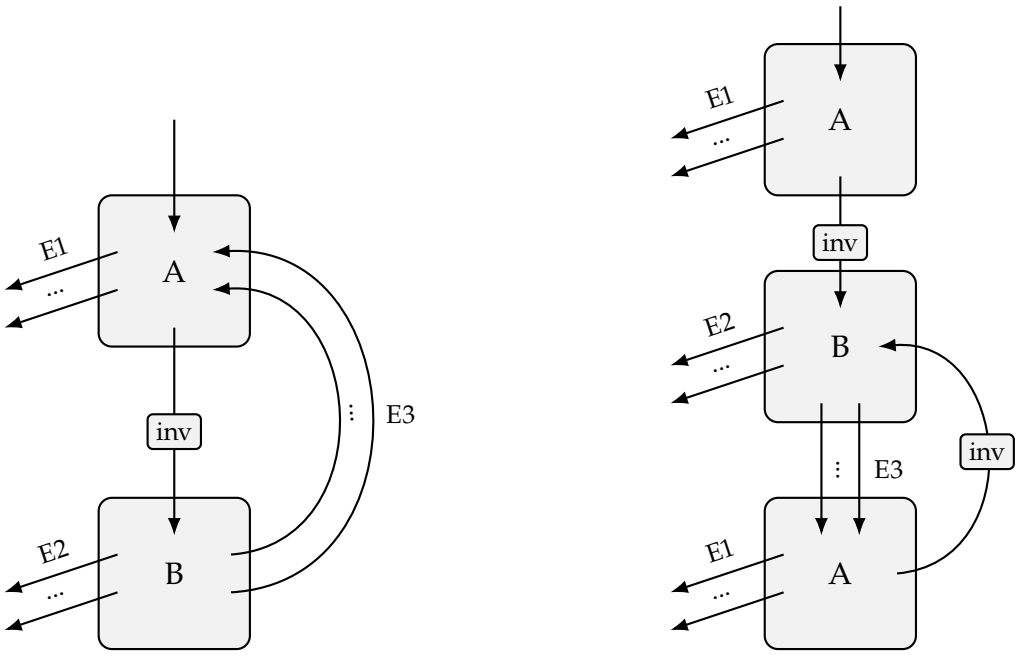
6: The loop invariant should also be strong enough to be used as an inductive argument in the correctness proof of the loop, but it is not necessary to focus on this aspect to understand our encoding.

To understand how the technique works, consider the diagram on the left of Fig. 3.30, which represents the generic shape of *any* loop in Rust. The blocks "A" and "B" are composed of any number of CFG nodes, and the "inv" edge between them is the point where the loop body invariant annotation is specified. The edges "E1" represent the exit edges of the loop that start from a statement placed inside the loop, but before the loop body invariant. Similarly, "E2" represents the exit edges placed in the loop, but after the loop body invariant. These two kinds of edges are, for example, `break`, `return` and error-handling statements. Finally, the edges "E3" are CFG edges that start in one of the statements in "B" and end up in "A". These are typically caused by `continue` statements or by the end of the loop body.

The first and most important step in our encoding is the semantics-preserving CFG transformation that makes the entry point of "B" become the new head of the loop. This is done by duplicating the group of CFG blocks in "A" and its exit edges "E1", moving the targets of "E3" to the new group "A" and creating a new edge between the end of the new "A" and "B". The resulting CFG is shown in right of Fig. 3.30.

The advantage of the new CFG is that, after the transformation, the loop body invariant holds at the beginning of the loop head. This is a much simpler kind of loop, which can be verified using standard techniques. For example, the loop and its invariant can be modeled as a `while (true) invariant <perms> && <inv> ...` loop in Viper and the exit edges as `goto` Viper statements.

The only missing step is to generate the permissions `<perms>` and the functional specification `<inv>` of the loop invariant. The latter is done by using the functional encoding presented later, while the permissions to be used in the invariant are determined by a static (syntax- and type-based) analysis of the loop body. At a high level, this analysis computes the Rust places that, during a loop iteration, are required to hold an exclusive capability, or for which a shared capability is enough. This is done in a few passes. First, the analysis computes which Rust places are definitely



(a) Original CFG of a loop. The loop head is in "A".

(b) Modified CFG. The loop head is in "B".

Figure 3.30: CFG transformation for the encoding of loops. The rectangles "A" and "B" represent two generic groups of CFG nodes. For each group, the diagram represents only the CFG edges that exit from the group.

initialized when the loop is reached for the first time. Second, the analysis collects all the Rust places that are used for mutations, i.e., those that are in the left-hand side of an assignment, are moved or are mutably borrowed during the loop. This can be performed by pattern-matching on the syntax of Rust statements and querying the type system to determine whether the assigned types are duplicable. Finally, the analysis collects all Rust places that are ever used in the loop. The permissions of the loop invariant are then determined by generating an exclusive permission for the places that are initialized and used for mutations, and by generating a shared permission for the places that are initialized and used, but not for mutations. These two sets of places associated with a permission are not yet ready to be used in the loop invariant, because there might be duplicate permissions when, e.g., both `x.y` and `x.y.z` are used for mutations. Since capabilities (and permissions) are transitive, `x.y` already implies the permission for `x.y.z`. To solve this, the actual permissions used in the invariant are computed by removing from the two sets of places all the places that are an extension of some other place in the same set.

Note that the soundness of our verification technique does not depend on the correctness or completeness of this algorithm that computes the permissions of a loop invariant, because Viper will nevertheless check that the permissions are correct and sufficient to verify the loops. In fact, our encoding is actually incomplete when the loop body reborrows a reference or calls a reborrowing function. In those cases, we believe that the permissions of the invariant can be encoded with a magic-wand resource, to model the capabilities that should be restored when the reference expires. In our automation technique presented in Sec. 3.7,

we do already compute a static representation of the information of all references at a given program point. We think that this representation should be sufficient for future work to additionally generate the magic-wand resources that should be placed in a loop invariant.

As an example of the computation of the permissions of the invariant, consider the loop in the `main` function of Fig. 3.31, which iterates over the values in the range between 0 and 10. At the point of the loop invariant, the surely initialized places are three: `upper`, `i`, and `range`. During the loop, `new_i`, `i` and `range` are used for mutations because `new_i` and `i` appear in the left-hand side of some assignment, while `range` is mutably borrowed in the `next(&mut range)` call. The loop uses many places in a read-only way: `range.from`, `range.to`, `upper`, `new_i` and `i`. Note how several of these are also used mutably or are an extension of some place used mutably. To compute the exclusive capabilities that should go in the loop invariant, it is enough to take the places used mutably, except for `new_i` which might be uninitialized (e. g., the first time that the execution reaches the invariant). In the resulting set of places {`i`, `range`}, there are no places that are an extension of other places, so no further steps are to be performed. Regarding the shared capabilities, many of the places used in a read-only way are already covered by the exclusive capabilities that we just computed: `range.from`, `range.to`, and `i`. After removing them and the `new_i` which might be uninitialized, the resulting set of places just contains `upper`. Thus, the algorithm encodes in the loop invariant a full permission for `i` and `range`, and a fractional infinitesimal permission for `upper`. In Viper, this can be expressed as `Usize(i) && IterRange(range) && acc(Usize(upper), rd())`.

```rust
struct IterRange {
    from: usize,
    to: usize,
}

#[requires(...)]
#[ensures(...)]
fn next(iter: &mut IterRange) -> Option<usize> {
    ...
}

fn main() {
    let upper = 10;
    let mut new_i: usize;
    let mut range = IterRange { from: 0, to: upper };
    let mut i = 0;
    while let Some(_) = next(&mut range) {
        body_invariant!(
            i == range.from - 1 && range.from <= range.to &&
            range.to == upper
        );
        new_i = i + 1;
        i = new_i;
    }
    assert!(i == upper);
}
```

Rust

Figure 3.31: While loop with a body invariant.

3.6.1.3 Encoding of Shared References

So far, we always described that shared capabilities are encoded as infinitesimal fractional permissions in Viper. Fractional permissions model the immutability and the potential of having aliases, but not yet the duplicability of shared capabilities. To model the latter, our technique relies on an elaboration of the compiler information to determine when a shared capability should be created, duplicated and when it should expire at the end of a lifetime. Each of these three cases is encoded in a different way. When a shared capability is created from a place that has an exclusive capability, the Viper encoding generates a `exhale acc(T(...), write - 2 * rd())` statement to model that the permission amount associated with the predicate instance of `T` should be lowered until it is equal to `2 * rd()`, where one `rd()` is associated with the place of the original exclusive capability and the second to the place of the newly created shared capability. When a shared capability is duplicated, the Viper encoding generates an `inhale acc(..., rd())` statement to generate a new `rd()` permission associated with the duplicated place. When the last shared capability associated with a place expires and unblocks the exclusive capability that was being borrowed, the Viper encoding generates an `inhale acc(..., write - rd())` statement to raise again the permission to the amount used to model exclusive capabilities. Only in these last two cases (duplicating or expiring a shared capability) the correctness of the generated Viper code relies on the correctness of the compiler. In particular, the encoding relies on the borrow checker component to make sure that the shared references are not duplicated after their expiration, and that shared references only expire once. In both cases, the Viper encoding would end up assuming that the fractional permission of the capability is greater than one, which contradicts one of the axioms of Viper. Because of this, in the presence of shared references and only for this reason, our technique does not fully verify memory safety.

To see an example, consider the encoding diagram in Fig. 3.32. In the `shared_references` function, the exclusive capability of the `x: &mut u32` argument is first downgraded to a shared capability to initialize `y1`. Then, the initialization of `y2` and `y3` duplicates an existing shared capability either from `x` or from `y2`. Finally, when the references `y1`, `y2`, and `y3` expire, all borrows of `x` expire and the original exclusive capability for `x` is restored. Along these steps, the Viper fractional permission associated with the place `x.target` changes as follows. Initially, before initializing `v1`, the permission amount is `write`, i.e., full. After the "create" encoding block the permission amount is `2 * rd()`, where one `rd()` can be seen as being associated with `x.target` and the other to `y1.target`. Then, after initializing `y2` it becomes `3 * rd()` and after initializing `y3` it becomes `4 * rd()`. That is, one `rd()` for the targets of `x` and of the other three local variables. During the "expire" encoding block, the three `exhale` statements consume a `rd()` permission each, bringing the permission amount to just `rd()`. Then, the last `inhale` upgrades the `rd()` permission of `x.target` back to the original `write` amount, effectively restoring the original capability.

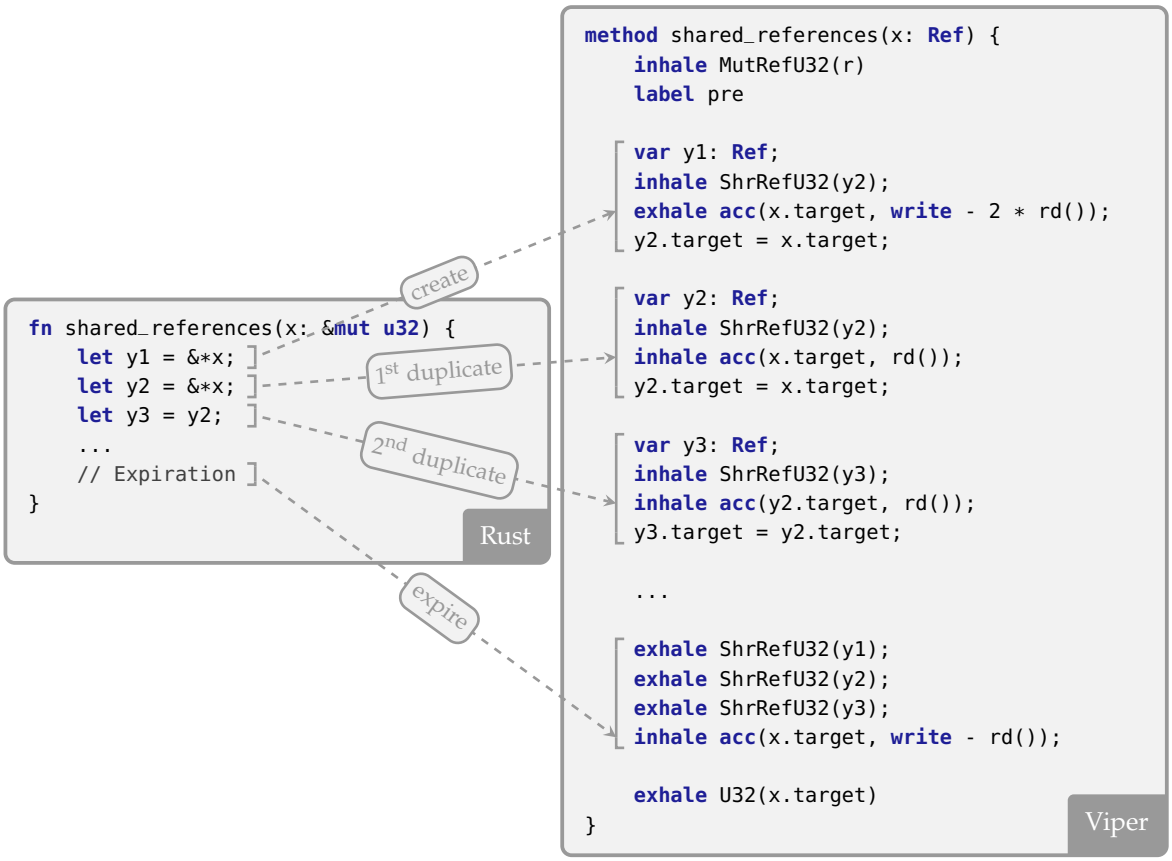


Figure 3.32: Imperative Viper encoding of a Rust function using shared references.

3.6.2 Functional Encoding

Our functional encoding has two goals. The main goal is to encode the pure expressions used in contracts and invariants into a formal logic, such as Viper or first-order logic. A second goal is to model in the proof that the evaluation of Rust expressions and Rust functions marked with the `#[pure]` attribute is deterministic. This is desirable when reasoning about code where a pure function is called multiple times with the same arguments, and is also necessary for soundness when modeling pure functions as mathematical functions in our encoding. The imperative encoding presented so far is not suitable for these goals. This is because the Viper language requires contracts and invariants to be written using only a restrictive pure subset of Viper, which does not include, e.g., assignments to local variables or fields. Moreover, to express that the result of a Rust function is a deterministic function of its parameters, the verification user would have to model that the result is a pure function of its arguments, thus requiring the existence of a pure encoding of such function anyway. For these reasons, our technique defines a functional encoding, composed of two steps: a Rust functionalization and a 1:1 translation to Viper. Overall, the functional encoding generates pure Viper functions and expressions starting from a subset of Rust that we designed to be deterministic and side-effect free.

At the Rust level, our technique requires all Rust expressions written in contracts, invariants and in all functions marked with `#[pure]` to be *pure*, meaning they are deterministic and side-effect free. Expressions that do

not satisfy these requirements are conservatively rejected by the verifier[7]. For example, calling non-pure functions from a pure expression is not allowed, as well as declaring a pure function with a mutable reference argument.

Expressions in Rust are quite flexible, because they can contain blocks[8] with `if` and `match` statements, local variables, overflow checks and other kinds of assertions. To take advantage of this expressivity, our functional encoding uses as input the CFG representation of the evaluation of a Rust expression. The only requirement is that the code should not contain loops and that all assignments should have a local variable in their left-hand side (i.e., `x = ...` is allowed but `x.y = ...` is not). The functionalization step of our technique converts a MIR body to a single Rust expression that does not contain any expression blocks. For example, the functionalization of `{ let x = y; x + 2 }` where `y` is a function parameter is `y + 2`; all imperative-style statements such as local variable assignments disappear during the conversion.

At a high level, the functionalization is a symbolic interpreter of MIR statements that computes a pure Rust expression representing the *result* of the evaluation of the MIR body. This interpreter works *backward*, i.e., starting from the final blocks in the CFG and following the CFG edges backward, and is based on syntactic *substitutions*, meaning that each assignment is interpreted by searching the left-hand-side variable in the result and replacing all occurrences with copies of the right-hand side of the assignment. At the beginning of the interpretation, the expression representing the result is a fixed local variable called `result`[9], which, substitution after substitution, ends up being an expression where the only free variables are the arguments of the original pure Rust function or expression that was to be encoded. What we described works only for straight-line code; for code with branches at each join point in the CFG the state of the interpreter has to be duplicated so that each branch can perform its substitutions independently. The important step is the encoding of a branch point, where the state of the interpreter from the branches is unified generating an `if` expression.

To see an example of the execution of the symbolic interpretation, consider the code in Fig. 3.33. The interpreter starts at ① with a result expression that is simply `result`. The interpreter proceeds to evaluate the statement `return tmp`, and with a [`result` → `tmp`] substitution the result expression becomes `tmp`. At the join point ②, the state of the interpreter is duplicated to follow both branches independently. To go from ③ to ④, the interpreter replaces [`tmp` → `b`], so the result becomes `b`. To proceed to ⑤, the two branches need to be unified with an `if` expression, so the result becomes `if b > a { b } else { tmp }`. To go from ⑤ to ⑥, the interpreter replaces [`tmp` → `a`], so the result becomes `if b > a { b } else { a }`. This is the end of the symbolic interpretation. At this point, each free variable in the expression of the result is an argument of the function. This is guaranteed because any other local variable in such an expression is removed when symbolically interpreting the first statement that initialized it. For example, when going from ⑤ to ⑥ the interpreter replaces *all* usages of `tmp` with `a`. Since local variables cannot be initialized cyclically, this is sufficient to guarantee that by the time the interpretation terminates the expression of the result will only mention the arguments of the function.

7: Sometimes, to show that a pure function is well-defined, the syntactic checks are insufficient and the verifier would also need to prove termination. This could be done by adding decreases-clause annotations on the Rust source code and checking them with an encoding into Viper. Our verifier does not do this and, instead, relies on the user to check termination of its pure functions.

8: A block expression in Rust is an anonymous namespace scope that can be used as an expression and that can be defined using imperative statements. For example, the block expression `{ let x = 40; x + 2 }` is an expression that evaluates to 42.

9: In MIR, the result corresponds to the local variable named `_0`.

```rust
#[pure]
fn max(a: i32, b: i32) -> i32 {
    // ⑥ result := if b > a { b } else { a }
    let mut tmp = a;
    // ⑤ result := if b > a { b } else { tmp }
    if b > a {
        // ④ result := b
        tmp = b;
        // ③ result := tmp
    }
    // ② result := tmp
    return tmp
    // ① result := result
}
```

Rust

Figure 3.33: Functionalization of a Rust function, generated with a backward interpreter.

After the functionalization step generates a single Rust expression containing no imperative statements, the translation to Viper is defined as a 1:1 mapping from Rust expressions to Viper expressions:

- Rust function arguments are translated to Viper function arguments with the same name.
- Primitive values in Rust are translated to primitive values in Viper.
- Rust operations between boolean or integer values are translated to the equivalent Viper operations.
- `if a { b } else { c }` Rust expressions are translated to the equivalent `a ? b : c` Viper conditional operator.

Given that we designed the language of pure expressions to have the same syntax and semantics as Rust, one possible outcome of their evaluation is for the expressions to panic. To prevent such cases, our technique generates well-definedness checks in parallel to the functional encoding. In particular, for the case of Rust function marked as `#[pure]`, the verifier uses the imperative encoding to verify that the precondition of the pure function is sufficient to guarantee absence of panics during its evaluation. In the case of pure expressions written in contract or loop invariant annotations, we identified two alternative solutions. The first solution is to verify that all panicking cases of contract expressions are unreachable. For example, given a possibly-panicking precondition such as `a + b == c`, where `a`, `b` and `c` are of type `u32`, the verifier should reject the contract because `a + b` might cause an integer overflow. To fix this example, the user would have to strengthen the precondition to something like `a < 10 && b < 10 && a + b == c`. This way, when `a` or `b` is too large the evaluation of the expression is well-defined to be `false`. However, this approach requires users to manually specify verbose conditions, potentially making the overall contract less readable. So, in our verification technique, we instead opted for a second solution, which automatically generates the conditions under which the expression does not panic, so that contract annotations evaluate to `false` when such conditions are not met. While this could be done by using existing techniques to compute the weakest precondition of the expressions used in the contracts, we found that to be unnecessary. Instead, when encoding contract annotations, we instruct the functional encoding to treat every panic call as if it was a `return false;` statement. This way, our encoding achieves the desired semantics in a way that is both lightweight for the user (in terms of the number of annotations

that must be written) and easy to implement. One downside of this approach is that symbolic manipulations of the specifications are as difficult as symbolic manipulations of the Rust code. For example, in a precondition `a + b < c || a + b >= c` is not equivalent to `true` because, in the case where `a + b` causes an overflow, the expression evaluates to `false`. As another example, since boolean operations have short-circuiting semantics, `true || 1/0 == 1` evaluates to `true` while `1/0 == 1 || true` evaluates to `false`.

3.6.2.1 Alternative Functionalization: Forward Interpretation

While designing the functionalization, we explored several other possible techniques. One of them is a substitution technique that works forward, instead of backward. The state of the interpretation, in this case, is not just one expression representing the result, but instead is a map from local variables to expressions that represent their value. Instead of generating a single `if` at the branching points, the interpretation generates one `if` for each local variable at each join point. This approach, compared to the backward interpretation, tends to generate expressions that are larger, because each branch can generate many more `if` expressions. Moreover, the algorithm uses more memory because it needs to store an expression for each local variable, even when the local variable does not influence the result. Among the advantages, by working forward the algorithm is usually easier to follow step-by-step and debug. Moreover, the algorithm computes a pure encoding of each local variable at each program point, which might be useful for other techniques. For example, an optimization pass of a compiler might use this technique to detect which local variables correspond to expressions that are syntactically equal, so that their evaluation can be deduplicated.

The code in Fig. 3.34 shows an example of the execution of our forward-interpretation algorithm. Starting from the top, where the only local variables that are defined are the arguments, the execution defines `tmp := a` and, inside the then branch, re-defines it as `tmp := b`. To unify these representations, when joining the branches the technique introduces an `if` expression.

```rust
#[pure]
fn max(a: i32, b: i32) -> i32 {
    // ① a := a, b := b
    let mut tmp = a;
    // ② tmp := a, a := a, b := b
    if b > a {
        // ③ tmp := a, ...
        tmp = b;
        // ④ tmp := b, ...
    }
    // ⑤ tmp := if b > a { b } else { a }, ...
    return tmp
    // ⑥ result := if b > a { b } else { a }, ...
}
```

Rust

Figure 3.34: Functionalization of a Rust function, generated with a forward interpreter.

```rust
#[pure]
fn max(g: bool, x: i32, y: i32) -> i32 {
    // ③ result := def x1 = y + z in g ? x1 + 1 : x1 - 1
    let x = y + z;
    // ② result := g ? x + 1 : x - 1
    if g { x + 1 } else { x - 1 }
    // ① result := result
}
```

Rust

Figure 3.35: Functional Viper encoding of a Rust function, generated with a backward interpreter using syntactic definitions.

3.6.2.2 Alternative Functionalization: Syntactic Definitions

Both functionalization approaches described above have the downside that, in the worst case, the size of the generated expression is exponential in the number of branches contained in the imperative code, which causes a large memory consumption to represent the AST of the expression. The bottleneck is the encoding of the join and branch points, which in the worst case can duplicate each time the size of the encoded expression. One way to solve this problem, in both approaches, is to extend our language of functional-style expressions to include syntactic definitions. To distinguish these from regular Rust local variable declarations, we use the syntax `def` ⟨`variable`⟩ `=` ⟨`definition`⟩ `in` ⟨`expression`⟩. With this, the functionalization can introduce a new syntactic definition whenever a large expression needs to be duplicated at a join or a branch point. This allows large expressions to be used just once in the definition of a fresh variable, which can then be used multiple times without causing a worst-case exponential growth of the size of the AST. Apart from the reduced memory usage and all the advantages that derive from it, another advantage of this approach is that the generated pure expression tends to be easier to read and debug for humans because the overall size is smaller and the syntactic definitions correspond almost exactly to the let assignments that were in the original Rust code. The only disadvantage is during the translation to Viper. Rust syntactic definitions can be translated 1:1 to Viper let expressions, but Viper does not permit using certain expressions inside the definition of let expressions. For example, this is the case of the `unfolding` permission-manipulating expressions that will be described in Sec. 3.7. This let-based encoding cannot be used in the presence of such expressions.

The code in Fig. 3.35 shows an example of this functionalization technique applied on a backward interpretation. When going from ② to ③, instead of repeating `y + z` twice in the expression representing the result, the technique generates a new unique variable name `x1` and uses `y + z` just once to define `x1`. The `x1` variable is then used multiple times in the expression of the result, but that does not increase the size of the expression. Note the similarity between the generated syntactic definitions and the corresponding Rust assignment that generated it.

3.7 Automatic Generation of the Core Proof

So far, we presented our verification technique with one big simplification: the generated Viper program does not contain any of the proof steps that Viper needs in order to know where to apply the definition of

predicates. Since in Viper, predicates can be defined recursively and aliasing is unrestricted, this is in general an undecidable problem [63]; the search space for the proof is too big to be explored exhaustively. In fact, automation of predicate reasoning is a known problem in the automated verification community. Existing solutions rely on triggering heuristics [16, 17, 19, 20, 23, 64] or require the verification user to manually specify when a recursive definition should be applied. The latter, for example, is done with `fold` / `unfold` / `unfolding` operations in Viper and in verifiers based on it [25, 26, 65], with `open` / `close` operations in VeriFast [64], and with `reveal` operations in Dafny [20].

Our key observation is that, despite the known challenges of recursive definitions, a memory-safety proof of a Rust program can be constructed automatically and deterministically using an approach that associates capabilities with syntactic places. What is special about Rust is that the language has been designed around several restrictions, such as aliasing, linearity, and mutability, that make it possible for the compiler to check memory safety statically. In other words, a Rust program that compiles is a testimony that the compiler internally constructed the equivalent of a memory-safety proof. In this section, we present our technique — inspired by the compiler — to reasoning statically about Rust capabilities. This technique can be seen as a *certified* borrow checker implementation, which, as a certificate, generates a separation-logic memory-safety proof of the supported programs[10]. The resulting proof serves as the basis for verifying functional properties, whose encoding has been presented in Sec. 3.6. In the proof, all recursive predicate definitions are explicitly applied where necessary, addressing the automation challenge presented above.

3.7.1 Capability Analysis

At a high level, our technique works as a static analysis that follows the CFG of the program while tracking in its state a set of capabilities associated with syntactic places. Whenever a Rust statement requires a particular capability, the state is modified using state operations that apply the required predicate definitions and make the capability available. Whenever the CFG joins two branches, the state from the branches is unified using operations until the two states are identical. These state operations, which are saved and later encoded to Viper, are the proof hints that are needed to complete the separation-logic proof. In a way, the computed information is a representation of the definitely-initialized places of a Rust program, but augmented with step-by-step checkable explanations of what the analysis computed. In this subsection, we present the analysis of programs without references, and we later expand the analysis to also cover usages of references.

The outcome of our analysis is a program annotated with state operations written in an intermediate Rust-like language that we call a *VIR* (i.e., verification intermediate representation), and with a description of the capabilities available at each program point. The benefit of this representation, compared to Rust, is that the state operations describe explicitly the hidden steps that the Rust compiler performs internally when checking a Rust program. This is useful for visualization, debugging and implementation purposes. As an example, Fig. 3.36 shows a

[63]: Becker et al. (2019), *The Axiom Profiler: Understanding and Debugging SMT Quantifier Instantiations*

[16]: Barnett et al. (2005), *Boogie: A Modular Reusable Verifier for Object-Oriented Programs*
[17]: Filliâtre et al. (2013), *Why3 - Where Programs Meet Provers*
[19]: Cohen et al. (2009), *VCC: A Practical System for Verifying Concurrent C*
[20]: Leino (2010), *Dafny: An Automatic Program Verifier for Functional Correctness*
[23]: Müller et al. (2016), *Viper: A Verification Infrastructure for Permission-Based Reasoning*
[64]: Jacobs et al. (2011), *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java*

[25]: Eilers et al. (2018), *Nagini: A Static Verifier for Python*
[26]: Wolf et al. (2021), *Gobra: Modular Specification and Verification of Go Programs*
[65]: Heule et al. (2013), *Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions*
[64]: Jacobs et al. (2011), *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java*
[20]: Leino (2010), *Dafny: An Automatic Program Verifier for Functional Correctness*

10: We remind that, in the presence of shared references, our technique relies on the correctness of the compiler's borrow checker implementation and, as such, does not entirely prove memory safety.

representation of the outcome for the analysis of `get_values`. In there, the only state operation is an `unpack curr` that opened the definition of the unique capability `List<T>` for the memory location of `curr`. The capabilities at each program point of this example are described by the comments that start with `// Unique`.

```
struct List<T> {
    value: T,
    next: Option<Box<List<T>>>,
}

fn get_values(list: List<u32>) -> Vec<u32> {
    // Unique: { list }
    let mut curr = list;
    // Unique: { curr }
    let mut values = vec![];
    // Unique: { curr, values }
    loop {
        // Unique: { curr, values }
        unpack curr;
        // Unique: { curr.value, curr.next, values }
        values.push(curr.value);
        // Unique: { curr.value, curr.next, values }
        match curr.next {
            Some(box tail) => {
                // Unique: { curr.value, tail, values }
                curr = tail;
                // Unique: { curr, values }
            }
            None => {
                // Unique: { curr.value, values }
                break;
                // Unique: { curr.value, values }
            }
        }
        // Unique: { curr, values }
    }
    // Unique: { curr.value, values }
    values
    // Unique: { result }
}
```

VIR

Figure 3.36: Result of a capability analysis. The program is augmented with a description of the capabilities available at each program point, and in this case one `unpack` state operation introduced to satisfy the requirements of the `values.push(curr.value)` statement. For simplicity, the temporary mutable borrow of `values` in that statement is not represented in this example.

3.7.1.1 Capability State

The state of our analysis describes which capabilities are available at a specific program point. Each capability is associated with a syntactic place[11], which in our approach *holds* the capability corresponding to the memory location identified by the place. The type of each place is uniquely determined by the compiler, so there is no need to keep track of it in our analysis.

11: A place in this context is an expression that starts with a local variable, followed by a sequence of field accesses, downcasts of enumerations, or dereferences.

The initial state that describes the capabilities at the beginning of a function simply associates a unique capability to each function argument. As an example, consider Fig. 3.36. The state at each program point is described by the `// Unique` comments. In this case, there is no state that contains a shared capability, so the states only define the set of places

that hold a unique capability. Since the function has a single argument, the initial state contains only one unique capability, held by `list`.

Since capabilities are transitive, not all combinations of places and capabilities are legal. For example, at any program point, it would be illegal to have a state where there is a unique capability held by `curr` and *also* a unique capability held by `curr.next`. This is because the capability for `curr` already provides the capability for `curr.next`. In our analysis, all states are legal, because the initial state is legal by construction and because all state transformations (that we are about to describe) preserve the legality of a state.

3.7.1.2 State Operations

It is possible to have different capability states that represent the same capabilities. For example, in Fig. 3.36 the state `curr` and the state `curr.value, curr.next` describe the same capabilities, because the capability for `curr` is defined as being the union of the capabilities for its two fields. To reshape a state into a different but equivalent one, we define two state operations called **`pack`** and **`unpack`**. To reshape a state losing information, we define an **`archive`** operation.

The first two operations, **`pack`** p and **`unpack`** p are an application of the definition of the capability of the type of the place p. The effect of the pack operation is to add the capability for p to the state, by consuming the capabilities of its direct fields. The dual unpack operation performs the opposite: it consumes the capability for p, generating the capabilities for its fields.

The last operation, **`archive`** p, removes the capability for the place p from the state. Contrary to the other operations, the effect of an **`archive`** operation is that the resulting state is not equivalent to the initial one. In fact, the resulting state describes a weaker set of capabilities. This operation is needed to unify the capability states of some CFG join points. That is, if the execution of a branch consumed a capability while the second did not, the capability states before joining the branches are not equivalent. To make them equivalent, the second branch needs to archive its remaining capability. This unification step, presented in more detail later, is necessary to make sure that after the join all capabilities in the state are unconditionally available, as opposed to path-sensitive capabilities that are available only for certain path conditions. The archived path-sensitive capabilities are not completely lost; the analysis remembers the program point where a place was archived. We will see later another operation to restore an archived capability in the state of another program point of the program, provided that the path condition of the restoration and archival are the same.

3.7.1.3 Statement Requirements

During our analysis, each Rust statement is modeled to have some *requirements* on the capabilities of the state and an *effect* that describes how the statement changes the capability state. The requirements consist of two kinds: requirements to have a certain capability *contained* in the

state, or requirements to have some capability *unpacked* enough in the state.

As part of the first kind, a statement can require the state to explicitly contain a certain capability associated with a certain place. This is typically the case of places that are mentioned on the right-hand side of assignments and function calls[12]. As an example, the `let mut curr = list` statement in Fig. 3.36 requires a unique capability for `list`, because such capability is then transferred by the statement to the left-hand side. Similarly, the call `values.push(curr.value)` requires a unique capability for `curr.value` to be in the state. This is not immediately satisfied, because the state contains a capability for `curr`. However, by introducing an unpack operation the state is transformed in an equivalent representation where `curr.value` and `curr.next` are explicitly available.

12: Not all places on the left-hand side determine a capability requirement. For example, `let x = y` requires a capability for `y`, but cannot require one for `x` because before the assignment that place is uninitialized.

Just requiring certain capabilities to be *contained* in the state might be sufficient to express all statements in our analysis, but it would not be practical. The reason is that by not knowing whether these capabilities are packed or not, the definition of the *effect* of a statement would have to handle a much larger number of cases of combinations of capabilities in the initial state, including cases where before the statement the capabilities are contained one in another. As a solution, we introduce a second kind of requirement that simplifies the definition of the effect of statements: a statement can require *unpacking* certain capabilities if they are obtainable. This is typically the case of assignments that have at least a field access in their left-hand side, which require the base of the field access to be unpacked in the state. For example, consider the assignment `x.f.g = y`. The requirement of the statement is to unpack `x` and `x.f`. Thanks to this, the definition of the effect of the assignment can assume that the capability for `x.f.g` is either already in the state or cannot be obtained at all. That is, the definition of the statement does not need to consider the case where a capability for `x.f.g` is obtainable by unpacking other capabilities.

Note that requirements coming from the left and right-hand side of an assignment may be conflicting. Consider for example `(*x).f = foo(*x)`, whose right-hand side requires a capability for `*x`, while the left-hand side requires `*x` to be unpacked. These conflicts can be solved by desugaring the assignment to two simpler statements using a fresh temporary variable. For example, `let tmp = foo(*x); (*x).f = tmp`. A similar rewriting can be used when the requirements coming from different places in the right-hand side are conflicting. For example, the call `bar(x, x.f)` requires both the capability for `x` and for `x.f`, but the two are never available in the same state. This conflict can be resolved by introducing more local variables, automatically rewriting the call to `let a = x; let b = x.f; bar(a, b)`.

3.7.1.4 Statement Effect

Each Rust statement has an *effect* that describes how the statement changes the capability state. In our analysis, we make sure that the *requirements* are satisfied before the statement affects the state. This way, when modeling the effect we can assume that the requirements already hold. In general,

the execution of an assign statement (1) *consumes* the capabilities for places in the right-hand side that correspond to non-copy types, (2) *archives* the remaining capabilities for extensions of the left-hand-side place, and (3) *generates* a new capability for the left-hand-side place. The intermediate archive step is essentially a clean-up operation that removes remaining capabilities that would potentially overlap with the newly generated capability. Note that this model of the effect of statements covers the case of calls without a return value, because they can be seen as assignments of a unit value to an anonymous temporary variable, but is not yet expressive enough to describe the case of assignments that create borrows. To handle those cases, as we present later, it is necessary to track additional borrowing information in the capability analysis.

The function in Fig. 3.36 contains several examples of assignments. First, `curr = list` consumes a capability for `list` and generates one for `curr`. Then, `values = vec![]` removes no capabilities and generates one for `values`. Later, `values.push(curr.value)` does not consume a capability for `curr.value` because its type is copy, not generates any capability because the call does not return any value. In the match statement, `curr = tail` consumes a capability for `tail`, archives the remaining permission for `curr.value` and then generates one for `curr`. Finally, returning `values` removes all capabilities from the state and generates one capability for the distinguished symbol `result` that represents the memory location of the returned value.

3.7.1.5 Unification

When the analysis reaches a branch in the CFG, the capability state is duplicated for each branch, so that the analysis can explore them independently. However, when the analysis reaches a point where multiple branches are joined there is a problem: the capability states at the end of the branches might not coincide, or might not even be equivalent. To solve this, our technique performs a unification in three steps. First, the unification generates `unpack` operations if another branch has a capability that can be obtained that way. Second, the unification tries to generate `pack` operations to unify the capabilities that originated from different enumeration variants. Third, if the previous steps were not enough, the remaining differences between the states are removed by generating `archive` operations. Since the last step always succeeds, in the worst case the states can be joined by archiving all capabilities. However, the intention of the analysis is to compute which capabilities are unconditionally available, in contrast to path-sensitive capabilities that are available only for some path conditions. This way, the analysis models faithfully what the compiler knows to be definitely initialized. While in theory the difference between two capability states might just be in the capability kind, e. g., `x` is unique in one branch but shared in the other, this is only possible when a branch created a reference, which we cover later.

To see an example of unification, consider the states `a, b` and `a.value, b` where `a: List<u32>` and `v: Vec<u32>`. The `a.value` capability of the second branch can be obtained with an `unpack a` in the first branch. The resulting state for the first branch is `a.value, a.next, b`. Now, the `a.next` in the first branch cannot be

obtained by unpacking capabilities in the second branch. The two states cannot be unified by packing `a`, because the `a.next` capability does not originate from a difference in the enumeration variants unpacked in the two branches. Another way to see it is that the states cannot be unified by packing `a` because the second branch lacks the `a.next` capability necessary to do so. So, the remaining option is to generate a `archive a.next` in the first branch, so that the resulting unified state is `a.value, b`. These steps correspond to the unification of the two if branches in Fig. 3.37. Note how this reflects the initialization analysis of the compiler, because after the `if` statement `a.value` and `b` are usable in Rust, while `a.next` is not.

```
fn random_drop(mut a: List<u32>, b: Vec<u32>) {
    // Unique: { a, b }
    if rand() {
        // Unique: { a, b }
        unpack a;
        // Unique: { a.value, a.next, b }
        archive a.next;
        // Unique: { a.value, b }
    } else {
        // Unique: { a, b }
        unpack a;
        // Unique: { a.value, a.next, b }
        drop(a.next);
        // Unique: { a.value, b }
    }
    // Unique: { a.value, b }
}
```
VIR

Figure 3.37: Unification of two capability states. To unify `a, b` with `a.value, b`, the analysis generates an `unpack` and a `archive` at the end of the first branch.

3.7.2 Capabilities of References

In order to extend our automation technique to handle reference types, we need to model the implicit flow of capabilities that happens when a reference is created or *expires*. The expiration of a reference does not have any effect at runtime, but is only used to describe the point where a reference stops being used and the borrowed place regains the capabilities that make it usable. When that happens, we model that the capabilities for the target of the expiring reference go back to the syntactic place that was used in the right-hand side of the assignment that created the reference. We note two challenges when reasoning about these capabilities. First, depending on their borrowing relations, many references may expire at the same time. Second, modeling how expiring references restore the borrowed places might require temporarily referring to path-sensitive capabilities, in case a reference was created in some branch of the CFG. To handle these cases, we extend our analysis to generate a representation of the borrowing information that we call a *borrowing DAG*, which we then use to model the rearrangements of capabilities at expiration points.

3.7.2.1 Borrowing DAG

A borrowing DAG is a directed acyclic graph that represents for a certain program point the borrowing relations between its live references, and

```rust
fn branching<T>(mut a: T, mut b: T, cond: u32) {
    let y: &mut T;
    y = &mut a; // ①
    match cond {
        0 => { y = &mut a; } // ②
        1 => { y = &mut a; } // ③
        _ => { y = &mut b; } // ④
    }
    let z = &mut *y; // ⑤
    drop(z);
}
```

Rust

Figure 3.38: Rust function with reborrowing and conditional borrowing statements.

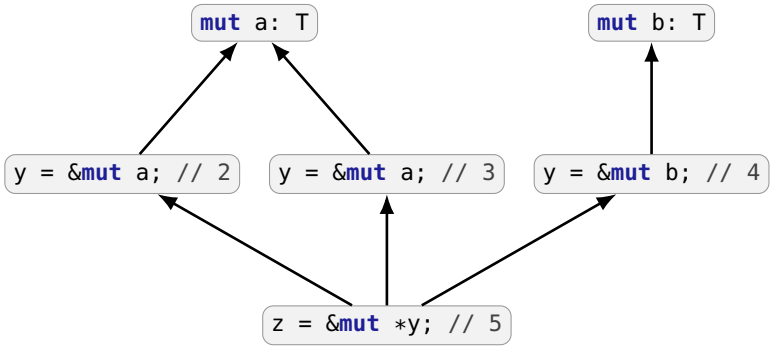


Figure 3.39: Borrowing DAG after the creation of `z` in Fig. 3.38.

between live references and function arguments. The nodes of the graph represent Rust places or local variables of type references, identified by the statement that initialized them. In particular, each sink[13] node represents a Rust place that is being borrowed, while each non-sink node represents a statement that creates a reference. Each directed edge in the graph, from the creation of reference a to a place b, represents that a borrows b (e.g., `let a = &mut b`) or the target of b (e.g., `let a = &mut *b`). A key property of the graph is that it describes the capability flow of an expiration point: when a reference expires the capabilities for its target "flow" through the borrowing DAG and reach the statement that borrowed them. In general, there can be more than one assignment that created a reference, which means that there is more than one way by which the capabilities of the reference can flow back to other places.

13: A node with no outgoing edges.

As an example, consider the function in Fig. 3.38. The borrowing DAG after statement ⑤ is represented in Fig. 3.39. The reference created at ① is not in the graph, because that borrow expires in the branches of the `match` statement. One way to visualize the expiration of capabilities is to imagine that when a reference expires the capabilities for its target "flow back" through the CFG up to the statement that borrowed them. Because of this, the shape of the borrowing DAG is always going to be a simplified version of the CFG of the function, but with its edges going in the opposite direction. For example, the reference `z` created at ⑤ has three outgoing edges, because in the CFG the creation of `z` happens after joining the three branches that initialize `y`. When `z` expires, the capability for `*z` goes back to `*y`, but then the capability for `*y` can follow ②, ③ or ④, depending on a path condition that we will model later.

3.7.2.2 Expiring References

When a reference expires, its capabilities go back to the original borrowed place. In our automation technique, we make this flow explicit, by introducing a new state operation kind[14] `transfer` $a \to b$, which explicitly transfers the capability from the syntactic place a to b. We also introduce a new `restore` p state operation to reverse the effect of a `archive` p performed at a past program point with the same path condition. This archive-restore pair is useful to remove some capabilities from the state, as long as those capabilities are path-sensitive and cannot be represented in the capability state. In particular, for each `archive` p operation that was executed along the way, the encoding generates a dual `restore` p operation that restores the capability for that place at an expiration point. Restoring a path-sensitive capability is needed to reconstruct the step-by-step flow of the capabilities for our memory-safety proof. The Rust compiler is only interested in knowing which references become available at the very end of an expiration, but in our analysis, we want to reconstruct which capabilities are temporarily restored in the intermediate steps that explain a flow of capabilities between distant nodes.

14: In addition to `pack`, `unpack`, `archive`.

As an example, Fig. 3.40 shows the result of the capability analysis on the function in Fig. 3.38. In this VIR program, `y` expires immediately after `z`, so after the `drop(z)` statement the analysis models their expiration in sequence, in reversed order of initialization. Since `y` was initialized in three different ways, the analysis models its expiration by generating a `match cond` statement with three branches, inside which the analysis restores either `a` or `b`, reversing the `archive` operation that the analysis generated in the original `match` statement. As a result, after the expiration of `y` and `z`, all original arguments `a`, `b` and `cond` are as usable as they were at the beginning of the function. This is reflected in the capability state, which at the end of the function is equal to the capability state at the beginning of the function.

3.7.3 Viper Encoding

The output of the capability analysis, composed of a VIR program and the capability state associated with each of its program points, is the last ingredient necessary for the generation of the core proof in Viper. The key property of the encoding is that for each capability computed by the analysis, the Viper proof holds a corresponding resource at the same program point. For example, if the capability state states that there is a unique capability for `x` and for `y`, then it means that `P(x) ✻ Q(y)` holds at the same program point in the Viper encoding, where `P` and `Q` are the predicates encoding the types of `x` and `y`.

To complete the Viper proof of the imperative encoding, the capability state operations of VIR need to be encoded to the corresponding Viper statement. Each `pack` p operation is encoded to a Viper `fold P(`p`)` statement, where `P` is the predicate corresponding to the encoding of the type of p. Respectively, each `unpack` p operation is encoded to a Viper `unfold P(`p`)` statement. All the remaining `transfer`, `archive`, and `restore` operations are encoded to a no-op in Viper, because resource

```
fn branching<T>(mut a: T, mut b: T, cond: u32) {
    // Unique: a, b, cond
    let y: &mut T;
    y = &mut a;
    // Unique: y, b, cond
    unpack y;
    // Unique: *y, b, cond
    transfer *y → a // Expiration of y
    // Unique: a, b, cond
    match cond {
        0 => {
            y = &mut a;
            // Unique: y, b, cond
            archive b;
            // Unique: y, cond
        }
        1 => {
            ...
        }
        _ => {
            y = &mut b;
            // Unique: y, a, cond
            archive a;
            // Unique: y, cond
        }
    }
    // Unique: y, cond
    unpack y;
    // Unique: *y, cond
    let z = &mut *y;
    // Unique: z, cond
    drop(z);
    // Unique: *z, cond
    transfer *z → *y; // Expiration of z
    // Unique: *y, cond
    match cond {
        0 => {
            transfer *y → a; // Expiration of y
            // Unique: a, cond
            restore b;
            // Unique: a, b, cond
        }
        1 => {
            ...
        }
        _ => {
            transfer *y → b; // Expiration of y
            // Unique: b, cond
            restore a;
            // Unique: a, b, cond
        }
    }
    // Unique: a, b, cond
}
```

VIR

Figure 3.40: Intermediate representation of the program in Fig. 3.38, augmented with capability-state operations that show the flow of capabilities. The comments in the code show the capability state at each program point.

reasoning in Viper is already path-sensitive and because assignments are not destructive. Even if these state operations do not need a Viper encoding, they are necessary for two reasons. First, the VIR language is based on Rust, so it technically needs new statements to express path-sensitive semantics that Rust does not have. Second, these state operations break down complex resource reasoning in very simple steps, providing this way valuable debugging information that would be harder to reconstruct in Viper. For these reasons, the state operations and our VIR representation might be useful for more techniques than our Viper-based verification. For example, they might be used for pedagogical purposes to explain how the Rust compiler internally performs its capability checks, or might be used as a starting point for a formalization of Rust that aims to define exactly how capabilities flow during the execution of a Rust program.

An important part of our technique consists of the verification of pledge annotations, which describe functional properties of functions returning references. In Sec. 3.5, we described that the capabilities of borrowed Rust types are encoded as magic wands resources in separation logic, whose creation is modeled by the Viper construct `package` $A \mathrel{-\!\!*} B$ `{ ... }` that we generate at the end of the encoding of functions returning references. In particular, given a pledge annotation of the form `#[after_expiry(`⟨F⟩`)]`, our technique places the functional property ⟨F⟩ on the right-hand side of the magic wand of the borrowed type, conjoining it with B, to express that the property of the pledge holds after applying the magic wand when the borrow expires. The package statement, in its body, requires showing with Viper code how the resources for B can be generated by consuming the resources in A. Since in our encoding from Rust both A and B represent the capabilities of the target of some references, our key idea is that the Viper code that we generate for the body of the package statement corresponds to the encoding of the expiration of the resources associated with A. Concretely, this means that to model an expiration the verifier should encode the expiration of the edges of the borrowing DAG that separate the resources of A from the resources of B. In fact, the effect of the expiration is to make the capabilities for B available again, exactly as the package statement requires. For more details regarding the generation of the body of package statements, refer to the thesis of Vytautas Astrauskas [66].

[66]: Astrauskas (2024), *Leveraging Uniqueness for Modular Verification of Heap-Manipulating Programs*

3.7.4 Analysis of Pure Expressions

When analyzing pure Rust expressions, i.e., the implementation of `#[pure]` functions or the expressions in contract annotations, the capability analysis described above can be performed in a much simpler way on the result of the functionalization step of Sec. 3.6. The advantage is that such a result is an expression that does not contain any imperative statement, nor borrow definitions or non-copy types. So, there is no need to model the effect of statements that might mutate the capability state like in Sec. 3.7.1.4, and there is no need to handle borrow expirations like in Sec. 3.7.2. For this kind of capability analysis on functionalized expressions, instead of the statement-like `pack` and `unpack` operations, we define a single expression-like state operation called `unpacking`, which can be used to temporarily unpack a capability in the context of

a subexpression. The syntax is `unpacking` p `in` e, where p is a place and e the Rust expression where p needs to be unpacked. Instead of following the CFG of the function, this capability analysis follows the AST of the expression, from the root to the leaves. In this representation, each internal node can be either a pure function call, an operation between primitive types, or an `if` expression, while the leaves are Rust places using one of the arguments of the function. When a node requires some capability to be unpacked, the analysis wraps a subtree in an `unpacking` p `in ...` expression, and then proceeds to analyze the subtree using a capability state in which p is unpacked. Lastly, for the translation to Viper, each `unpacking` operation is directly encoded as an `unfolding` Viper expression, which has an analogous semantics.

3.7.5 Alternative Viper-to-Viper Formulation

The capability analysis that we presented in this section works entirely at the level of Rust, generating a VIR program such that the encoding to Viper can be defined in a simple way with 1:1 translations from VIR to Viper. An alternative automation technique that we explored consists of performing the capability analysis not at the level of Rust, but on an intermediate Viper-like language that is generated before the analysis runs. In this intermediate layer, Rust programs would be modeled in a Viper-like language extended with the `transfer`, `archive` and `restore` state operations, but without Viper's `fold` or `unfold` statements, nor `unfolding` expressions. These fold-related operations that are necessary to translate the Viper-like program into a complete Viper one would then be generated with a Viper capability analysis that works analogously to the one that we presented for Rust.

The benefit of this approach is that by moving the automation technique closer to the Viper language, the automation can potentially be reused by other verifiers based on Viper. For example, to verify non-Rust programming languages by requiring to annotate their source code with Rust-like lifetime annotations. The disadvantage of this approach is that in order to guarantee automation the intermediate Viper-like language would need to be more restrictive than Viper. For example, it would need to reject Viper features such as quantified permissions, because such features manage permissions in a way that is difficult to track statically. So, evolving the encoding of this approach would be more constrained and difficult.

3.8 Implementation and Evaluation

[29]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*

In this section, we report the implementation and evaluation that we presented in our OOPSLA 2019 paper [29]. Since then, the technique and implementation have evolved in multiple ways:

- Prusti now supports more language features. Thus, Rust programs can now be verified with less manual rewriting. The main examples of this are loops with `return` and `break` statements, which no longer need to be rewritten to avoid these statements.

- We made our specification language more expressive, incorporating features such as loop body invariant annotations (which we describe in this thesis) and a deep value equality based on a new snapshot-based technique [31]. As a result, we expect that the number of annotations needed to verify existing programs should roughly remain the same.
- We upgraded the Rust compiler and Viper version used in our implementation. Since the encoding and verification techniques are largely still the same, we do not expect major changes in the verification time.

[31]: Astrauskas et al. (2022), *The Prusti Project: Formal Verification for Rust*

We have implemented our work as a Rust compiler driver, and evaluated it on a wide variety of crates (Rust packages) from the Rust package repository. The evaluation shows that our technique can generate core proofs fully automatically and verify interesting correctness properties without the need for complicated specifications. The version of our tool used for the evaluation is available as an artifact [57]; more recent versions are available as open-source software [27].

[57]: Astrauskas et al. (2019), *Software Artifact for the OOPSLA'19 Paper Titled "Leveraging Rust Types for Modular Specification and Verification"*

[27]: (2024), *Repository of the Prusti verifier for safe Rust.* URL: `https://github.com/viperproject/prusti-dev`

3.8.1 Implementation

We implemented a tool called Prusti as a Rust compiler driver, usable with Cargo, the official package manager for Rust. Working with Prusti provides a similar experience to existing tools used by Rust developers, such as the Rust linter Clippy [67]. Prusti performs its main work after the type-checking pass of the Rust compiler. We extract the compiler's MIR along with type and borrow-checker information, construct the corresponding Viper program, and verify it with Viper's symbolic execution verifier; verification results are translated back from Viper to Rust and reported using the Rust compiler's error reporting mechanisms. In addition to proving user specifications, Prusti optionally checks absence of panics and overflows.

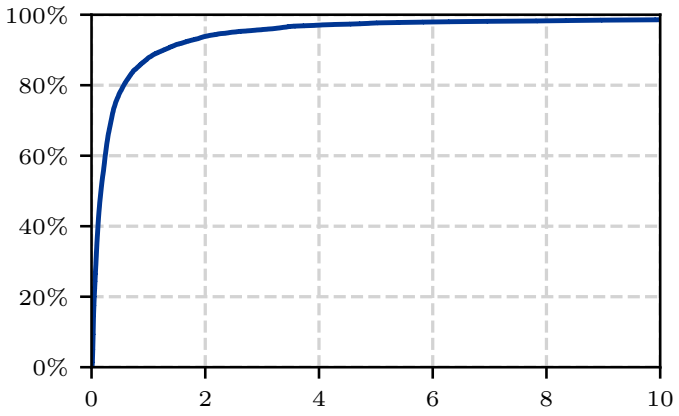
[67]: (2024), *Clippy: A collection of lints to catch common mistakes and improve your Rust code*

During the development of our tool, we built a test suite of more than 300 correct and incorrect Rust programs (annotated with expected verification errors) to check that we model corner cases of Rust's semantics correctly; these are provided with our implementation.

To support libraries, our tool provides a `#[trusted]` annotation, allowing us to equip functions with contracts used by callers but *not* checked against the function's implementation.

3.8.2 Evaluation

We evaluate our work in three ways: (1) we evaluate the construction of core proofs on all functions from the top 500 Rust crates that fall within our supported language subset; (2) we evaluate the ability to verify panic-freedom by proving the absence of overflows in examples that check for overflow at runtime, to determine whether these runtime checks may ever fail (without any user-provided specifications); (3) we evaluate the use of user-provided specifications by verifying panic freedom and richer functional correctness properties of existing implementations of well-known algorithms. All timings were performed using a clean Ubuntu



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

Figure 3.41: Left: cumulative distribution of the verification time (horizontal axis, in seconds) required for the core proof verification of each of the 11,791 supported functions (177 functions required between 10 and 120 seconds; 11 required between 120 and 888 seconds). Right: distribution of error messages for the overflow freedom evaluation on 519 functions.

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.

3.8.2.1 Core Proofs

15: On November 2nd, 2018.

[68]: (2024), *The Rust community's crate registry*

To test the automation of our core proof construction, we took the 500 most popular Rust community crates[15] [68], and applied three simple filters: firstly, we discarded any crates (148) which did not compile successfully within 15 minutes using the standard compiler configured and the *Polonius* borrow checker[16] (without our tool); secondly, we filtered all remaining 56,257 functions (top-level, `impl` and trait functions) with a simple syntactic check for unsupported language features; thirdly, we manually discarded ten unusually large functions that would have taken more than one minute just for the encoding, due to the large number of local variables used (five implement 4×4 matrix operations; the other five contain huge match expressions with up to 2,000 cases). This left us with 11,791 functions (21% of the total) to evaluate our work on. We re-ran the compiler with Prusti on the unmodified source code of these functions to generate and verify core proofs.

16: `rustc nightly-2018-06-27`; flags `-Zborrowck=mir -Zpolonius -Znll-facts` and using the reference Polonius algorithm ("Naïve").

The verification of these 11,791 functions succeeded as expected, without any need for manual intervention. This shows that we generate sufficient annotations to automate the core proofs in Viper. These annotations are non-trivial: we generated a total of 1,140,384 lines of Viper code, of which 138,499 are `fold`, `unfold`, `package` or `apply` statements necessary to automate the proofs.

We measured how much time is required by Viper to verify each function, reporting results (averaged across three runs) in Fig. 3.41 (left). We observe that the average verification time per function is 1.2 seconds, that only 0.16 seconds are enough to verify 50% of the functions, and that almost all of the functions (98.6%) are verified in less than 10 seconds each. A small fraction of the functions take more than 10 seconds to verify[17]. Upon close inspection, we found that these functions are similar in style to the ten unusually-large functions that we discarded (see above).

17: These functions were not manually discarded because their *encoding* takes less than one minute.

3.8.2.2 Overflow Freedom

We automatically identified all 519 supported functions in our evaluation crates which contain runtime checks for integer overflows or divisions by zero. We re-ran Prusti on these, enabling checks for panics and overflows (again, without specifications). Interestingly, 52 of these functions verified; on manual inspection, this was due to expressions that cannot overflow (e.g., `x-x/4`), or that were guarded by range checks. Since our tool proves soundly that these checks can never fail, one could eliminate them to improve performance without compromising safety.

For each of the remaining 467 functions, Prusti reported a verification error, listed in Fig. 3.41 (right). Manual inspection showed that these are mostly due to implicit assumptions on argument ranges; our technique makes it possible for developers to make these assumptions explicit as preconditions, and verify them at each call site. In eight cases, Prusti failed to prove that Rust's dynamic overflow checks actually imply that an operation does not overflow. Our inspection revealed that these verification failures are due to the handling of non-linear arithmetic in the underlying Z3 SMT solver. Increasing Viper's timeout for each Z3 query from 10 to 60 seconds results in "divide by zero" verification errors in all eight cases, which is the expected result.

Overall, this evaluation shows that even on a code *without specifications* our verification technique is already useful, proving automatically that several of Rust's runtime checks are redundant. In those cases where the verification fails, users have still the option to trade manual effort for stronger guarantees, adding preconditions. Still, we identify that SMT solver limitations remain an issue. These can be worked around, but there is still a need for better automation or mitigation techniques.

3.8.2.3 Specifications and Functional Behavior

In the third part of the evaluation, we investigated the specification and verification of both absence of panics and richer functional properties, using examples from the programming chrestomathy site Rosetta Code [69], a Rust tutorial on linked lists [70], and from Matsakis' blog posts on Rust's language design [71, 72]. From Rosetta Code, we manually selected a diverse list of eleven examples that either fall into the supported subset of the language or can be adapted without major changes. In order to handle examples using standard library types, we wrote wrappers marked with `#[trusted]` for these types (as explained above); we also rewrote `for` loops as `while` loops, and restructured some code to avoid `return` and `break` statements.

Table 3.1 gives an overview of the verified examples (we provide the code including specifications in the accompanying artifact). Before any manual modifications, the Rosetta Code examples had between 10 to 89 lines of code (excluding blank lines and comments) and between 1 and 6 functions. The average total verification time (averaged over 3 runs) is typically less than 30 seconds, which we consider reasonable for our so-far unoptimized encoding and tool. The slowest examples "Knight's tour" and "Knapsack Problem/0-1" take less than two and a half minutes

[69]: (2024), *Rosetta Code, Category Rust*

[70]: (2024), *Learn Rust by writing Entirely Too Many Linked Lists, First Final Code*

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

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

Table 3.1: An overview of the examples verified in the third part of the evaluation. The column "LOC" indicates 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" indicates the time in seconds required to encode and verify the example; "Viper Time" is just the time needed by the Viper symbolic execution back-end verifier to verify the encoding. "No Panic"/"No Overflow" shows whether we verified absence of panics/overflows ("–" means that the example contains no operations that could panic/unchecked arithmetic). The first two groups of examples are taken from the Rosetta Code website [69], except the "Linked List Stack" example which we took from [70] because it is more complete than the one in Rosetta. The second group differs from the first one in that we verified some functional properties. For example, for the "Ackermann Func." and "Fibonacci Seq." examples, we showed that multiple implementations all compute the correct result. We had to monomorphize "Binary Search" and "Selection Sort" for proving stronger functional properties because Prusti does not yet support intrinsic trait properties such as transitivity of the equivalence operator. As reported by Xavier Denis, in [29] we incorrectly marked one selection sort example as generic; we fix the mistake here. The preconditions we chose for the Ackermann functions do not prevent overflow and, thus, this aspect could not be verified (indicated by "×"). In "Knapsack Problem/0-1", we verified correctness of all intermediate computations; correctness of the result and absence of overflow would require sum comprehensions, an advanced specification feature not yet supported in Prusti. The two examples in the last group are from Matsakis' blog posts about non-lexical lifetimes in Rust [71, 72]. For one of them, proving panic freedom failed because the program does not handle all IO errors.

Example	LOC	#Fns	Spec. LOC	Time (s)		No Panic	No Overflow	Verified Additional Properties
				All	Viper			
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 (monomorphic)	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 behavior
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	×	-	

(each of them contains one large function that takes most of the time). In all cases, standard deviations were around 1 second.

For most examples, we verified the absence of panics and overflows, by adding specifications where necessary. In some cases, for example for "Binary Search", this required adding only a simple invariant that the indices are no larger than the vector's length, which allowed the verifier to prove not only the absence of out-of-bounds accesses, but also the absence of overflows. In other cases, for example for "Knight's tour", we had to add ghost code to encode object invariants. The most interesting specification for proving panic-freedom is for "Langton's Ant", which required not only quantifiers to specify an invariant of the grid on which the ant walks, but also a pledge to specify how changes made via borrows affect the invariant of the grid. Via our evaluation, we found a bug in the source code of this example, which causes an integer overflow during execution. We fixed this error by correcting existing boundary checks and types.

For seven examples, we also verified properties that go beyond basic safety. For two of them, we had to monomorphize the generic parameters to integers in order to use integer comparisons instead of a trait function. Functional correctness of the binary search example initially failed to

verify; closer inspection revealed an off-by-one bug in the source code (a fixed version verifies with our tool). We encode other properties such as sortedness ("Selection Sort"), functional correctness of recursive and iterative implementations ("Fibonacci Seq." and "Ackermann Func."), functional correctness of a data structure ("Linked List Stack"), correctness of intermediate computations ("Knapsack Problem/0-1"), and validity of parameter values in function calls ("Towers of Hanoi").

These seven examples require on average 1.3 lines of annotation per line of code. While this overhead is not negligible, it is lower than the overhead required by many existing verifiers for heap-manipulating programs. Moreover, our annotations are conceptually much simpler since they are expressed in terms of Rust expressions rather than complex program logics. Another core advantage of our approach is that the user is not forced to provide all of them from the beginning, but can add them gradually to strengthen the verified properties. For instance, proving safety for "Binary Search" requires only two lines of annotations. To additionally prove that the returned index is correct if `Some` is returned, the user needs to add two additional straightforward assertions. Finally, proving correctness for the case that `None` is returned is slightly more involved because it requires writing a quantifier that expresses that the vector is sorted. Nevertheless, none of these assertions expose the complexity of program logics for concurrent, heap-manipulating programs.

We also evaluated our tool on two examples from Matsakis' blog [71, 72], designed to illustrate difficult borrowing patterns. The support for the first example was added to stable Rust only recently, while the second one still requires a nightly-build version of Rust. Both examples are already supported by our tool (using the corresponding new borrow checker implementation).

[71]: Matsakis (2018), *MIR-based borrowck is almost here*
[72]: Matsakis (2018), *MIR-based borrow check (NLL) status update*

Overall, in our evaluation of Prusti, we demonstrated on a large collection of real-world code that our automated construction of the core proof works well. Additionally, we showed that our technique enables users to *incrementally* verify Rust code. In fact, even without specifications, our verifier is already useful and proves or reports possible overflows. Developers can then invest manual effort and add annotations using our *rich* specification language, obtaining in return the verification of stronger properties. Even in this setting, our technique is *lightweight* in that it requires a relatively low average number of annotations per lines of code.

3.9 Related Work

As a general point, we believe our tool was the first deductive verifier for Rust with source code contract annotations, and our implementation was the first verification technology to operate directly on the Rust compiler's analysis results and representations of source programs; there is no gap between the Rust programs and notions and the starting point for our work.

Capability-Based Type Systems Many other type systems can also be understood to associate capabilities with reference types [2]. Some extend pre-existing languages (e.g., C♯ [73] and Scala actors [74]); more recently, several programming languages have built these in (e.g., Pony [75], Æminium [76], Verona [77] and Rust itself [78]). Such built-in type systems can be used by the compiler: e.g., for memory management in Rust, or to enable the distributed garbage collection in Pony. While these systems provide programmers with stronger guarantees than traditional type safety, *functional correctness* of programs cannot be expressed: our work shows how to layer such verification concerns on top, while exploiting the benefits provided by the type system.

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

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

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

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

[76]: Stork et al. (2014), *Æminium: A Permission-Based Concurrent-by-Default Programming Language Approach*

[77]: Arvidsson et al. (2023), *Reference Capabilities for Flexible Memory Management: Extended Version*

[78]: (2024), *The Rust Programming Language*

Type Systems for Verification Liquid Types [79] equip types with logical qualifiers prescribing value properties; their extension to Alias Refinement Types [80] applies to mutable heap data structures. Type-checking is decidable, and loop invariants can be inferred. Flux [81] shows how liquid types can be used to verify lightweight correctness properties of Rust programs, taking advantage of ownership types to reason about imperative updates. Flux makes it possible to specify decidable invariants attached to types, verifying efficiently that they are respected. In particular, Flux introduces a new kind of mutable reference called strong reference (`&strg`), which can be used to modify the invariant attached to the target type, while regular mutable references cannot. Similar invariant checks can be expressed using our `assert_on_expiry` annotation. In comparison, the specification language of Prusti is more expressive because it is not limited to a decidable logic fragment. This makes it possible to specify more advanced correctness properties, such as sortedness and other relations between elements within the same collection.

[79]: Rondon et al. (2008), *Liquid types*

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

[81]: Lehmann et al. (2023), *Flux: Liquid Types for Rust*

SYMPLAR [82] targets formal verification for Java, employing a notion of permission (unique or immutable) to separate reasoning about aliases from functional properties concerning values. These permissions are checked using an SMT solver, while in Rust they are checked by the borrow checker using a decidable fixpoint technique. The verification of functional properties is performed in a second step, which leverages the aliasing restrictions of the permissions to generate verification conditions that are simpler than those of other approaches. For example, in our work, both aliasing and functional properties as encoded as part of the same program proof, making the job of the SMT-based verifier harder. The advantage of our approach is that our proof is more powerful, because it is ready to be extended to verification of code, such as unsafe Rust, in which the type system does not provide useful non-aliasing guarantees. The advantage of SYMPLAR's approach is that, within a codebase in which all types have been annotated with permissions, automation is easier because the SMT solver has to prove simpler properties.

[82]: Bierhoff (2011), *Automated program verification made SYMPLAR: Symbolic permissions for lightweight automated reasoning*

SPARK, a subset of Ada designed for formal verification, was extended in [83] by introducing Rust-like borrowing restrictions to enable reasoning about pointers. From 2019 to 2020, their approach used a `Pledge` annotation inspired by Prusti's pledges to describe the borrowing relation across function boundaries and loops [84]. While the syntax and semantics of these annotations are similar, they are verified in different ways. Prusti checks the memory safety of borrowing relations as part of the core

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

[84]: Dross (2019), *Pointer Based Data-Structures in SPARK*

proof, while SPARK relies on the borrowing restrictions of the language and only needs to verify the functional properties of their pledges. Since then, SPARK upgraded its internals to verify `Pledge` annotations using the *prophecy variables* technique of RustHorn [85], and later replaced the relation-based `Pledge` syntax with a value-based `At_End_Borrow` syntax. The resulting simpler borrowing annotations are closer to the `^` syntax used in Creusot [86], and the `at_expiry` alternative syntax that we presented in Sec. 3.4.

The Move prover [87] is a verification tool for Move, the smart contract language for the Libra/Diem blockchain [88]. The Move language is inspired by Rust, but is also much more restrictive. For example, Move has linear resources and references that can be either mutable or immutable, but does not allow developers to declare references in structures stored in global memory. Similarly to Prusti, the Move prover leverages these language restrictions to greatly simplify reasoning about aliased mutable data. In particular, the Move prover uses them to eliminate heap reasoning while encoding a program into Boogie [16], by transforming, e.g., a function with an argument `&mut T` into a function that takes and returns `T` values. This is done based on a *borrow graph* representation that tracks when references are released and how they relate to each other, similarly to our borrowing DAG presented in Sec. 3.7.

Flowistry [89] is a static analyzer for safe Rust, which shows that ownership types can be used to soundly and precisely reason about information flow in an ownership-based language like Rust. For example, the lifetime constraints in a Rust function signature soundly describe from which arguments a returned reference may be reborrowing from, and a static analysis can benefit from that to approximate the behavior of a function call. Similarly to Prusti, Flowistry taps into the type system information of the Rust compiler to perform its analysis of borrow information.

Automated Rust Verifiers CRUST [90], adaptations of SMACK [91], and KLEE [92] are *bounded* verification tools for Rust (including unsafe code); these tools allow user checks to be added as Rust expressions. They work on C/LLVM code where Rust's type information is absent. By contrast, we exploit this information for modular *unbounded* (sound) verification, and support richer functional specifications via `old` expressions and pledges.

Kani [93] is a bounded model-checker for Rust that internally encodes programs to CBMC [94]. Unlike other model-checkers, this tool is designed to be a Rust compiler backend, thus it has first-class access to Rust's type information. By default, Kani attempts to unroll all loop and function calls, and terminates only after having exhaustively checked all possible program executions. When this is not possible, the tool offers several annotations to the user to limit or simplify part of the proof. Among them, Kani offers annotations to limit the exploration to a certain number of loop unrollings, and stub functions to replace problematic code (e.g., using unsupported features) with a more verification-friendly version. Recently, Kani even added function contract annotations [95], based on CBMC's code contract, that enable users to benefit from the scalability advantages of modular verification techniques. However, they do not have annotations for loop invariants and their contract language

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

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

[87]: Dill et al. (2021), *Fast and Reliable Formal Verification of Smart Contracts with the Move Prover*

[88]: Amsden et al. (2020), *The Libra Blockchain*

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

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

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

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

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

[93]: Kani (2024), *Kani Rust Verifier*

[94]: Kroening et al. (2023), *CBMC: The C Bounded Model Checker*

[95]: (2023), *Kani RFC Book: Function Contracts*

is strictly more limited than Prusti's pure expressions, because it does not include `old(..)` expressions, pure functions, nor pledges.

RustHorn [85] proposes an encoding technique of safe reference-manipulating Rust programs into constrained Horn clauses [96], based on the key idea of using *prophecy variables* to describe the target value of a mutable reference at the moment of its expiration. The novelty of the prophecy variable approach is that the functional behavior can be modeled entirely in first-order logic by relying on the properties of the Rust language. In contrast, Prusti encodes to separation logic in order to verify the core proof, relying less on the Rust language guarantees.

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

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

Creusot [86] uses the prophecy encoding of RustHorn [85] to build a deductive verifier for safe Rust code by encoding Rust programs with contracts into Why3 [17]. Like RustHorn, Creusot relies on the Rust language properties and does not verify memory safety like Prusti. By avoiding the burden of the core proof, the tool can support more language features such as nested references, and verify similar programs in less time. Instead of pledges, Creusot lets verification users specify borrows in contracts using the `^` operator to refer to the target value of a mutable reference at the moment of its expiration. We believe that the semantics of the annotations are largely compatible, and that the syntax of pledges can benefit from the expressivity of the `^` approach as observed in Sec. 3.5.6. However, the prophecy variable technique might also bring some challenges, because so far they have not been used to verify invariants of mutably borrowed types that should hold by the time the borrow expires — a use case for which we designed the `assert_on_expiry` specifications. One possible direction for future work is to further investigate the differences between pledges and the prophecies annotations, bringing them closer in syntax and semantics. A unification of the two could be of great value for verification users.

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

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

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

Verus [62] is a deductive verifier for a language close to Rust, but with non-Rust syntax for proofs, contract specifications and ghost types. Compared to regular Rust code, libraries in Verus can be marked as safe under the assumption that all clients will be verified. This assumption is then used to avoid runtime checks in the implementation of the library, with a performance gain. The specification language of Verus offers a way to refer to the identifier of a type instance, which is preserved across moves of linear types. This annotation, combined with ghost types, makes it possible to split data from permissions in the specifications of the libraries, enabling reasoning about some cases of concurrency and interior mutability since this notion of identity is standard in separation-logic reasoning. We believe that Prusti can benefit from the same expressivity by using one of the techniques to encode object identifiers presented in Sec. 3.5.6.

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

Rust Verifiers Based on Interactive Theorem Proving Ullrich [97] encodes safe Rust programs into functional programs, to be interactively verified in Lean [98]. Reborrows are supported via lenses [99]. Recent work at Galois similarly reduces reasoning about a subset of safe Rust to proofs about functional programs in Saw [100]. In contrast to these works, our technique does not require the manual construction of proofs

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

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

[99]: Foster et al. (2015), *Combinators for Bi-Directional Tree Transformations: Linguistic Approach to the View Update Problem*

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

or verifier directives; in addition, we designed our underlying separation-logic formalism to provide a suitable (imperative-style) model for future extension to unsafe code.

RustBelt [101] provides a formalization aimed at proving that unsafe library implementations encapsulate their unsafe behavior, and defining formally what this notion should mean for Rust. RustHornBelt [102] adds functional specifications to RustBelt predicates, formalizing the prophecy variable technique used by RustHorn. The goals of our work are very different, and this led to different technical choices and contributions. Whereas RustBelt aims to formalize metatheory for the language and to construct proofs using the Coq proof assistant, we do not address Rust semantics, and aim primarily at the functional (automatic) verification of safe Rust code, and to equip programmers with specification mechanisms at the level of abstraction of such code, largely shielding them from the complexity of formal logics.

[101]: Jung et al. (2018), *RustBelt: Securing the foundations of the Rust programming language*

[102]: Matsushita et al. (2022), *RustHornBelt: A semantic foundation for functional verification of Rust programs with unsafe code*

RustBelt handles the expiry of borrowed references with a combination of a *lifetime logic* for determining when lifetimes can be known to safely end, and rules that restore full capabilities to a borrowed-from place once the corresponding lifetime is over. Restoring capabilities using this indirection via the lifetime logic has the advantage that this solution works consistently for mutable and for shared borrows, and for both safe and unsafe code (in the latter case, more work is required in the lifetime logic itself). Our handling of shared borrows (and the corresponding upgrade to full permissions) was inspired by this approach, but we rely on Rust's borrow checker to determine when borrows expire.

One key difference in our technical approach is that our logical encoding of borrows (using magic wands) reflects the flow of capabilities from the re-borrowed reference to the place borrowed from. It is this modeling which enables our pledges specification to be layered on top, since we can relate the two states of the accessible memory before and after the borrow's expiry in one assertion. In this way, we can directly express how changes made via a borrow affect the borrowed data; this is a fundamental difference in the two models. It would be interesting future work to integrate the two approaches, using RustBelt's lifetime logic to justify expiry of references in unsafe code in place of the (currently trusted) borrow checker we use for safe code.

There are also other technical differences that were motivated by the differing goals of the projects. RustBelt focuses on a formal program model usable even in the presence of unsafe Rust, designed for Coq-based proofs. Because of this, their logical foundation is a powerful and complex separation logic based on Iris, and their language formulation is a continuation-passing-style intermediate representation, convenient to work with in Coq. Translations from Rust to this representation have to be performed manually by experts. For our goals, it is essential that the translation to the language in which the proof is performed is automated, and that the input specifications written by users match the features and abstraction level of Rust's source code.

Aeneas [103] translates Rust programs into a value-based functional language that can be used to prove functional properties of the original program using interactive theorem provers. To reason about modifications via mutable references, instead of RustHorn's prophecies or Prusti's

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

pledges Aeneas proposes a new *backward function* translation technique and a *borrow graph* static abstraction that represents the dependencies between borrows. Compared to Prusti and other deductive verifiers, the Rust language support of Aeneas is a bit more limited because of some limitations regarding nested loops and disjunctions in the control flow. It would be interesting to explore in future work the relation between Aenea's borrow graph and Prusti's borrowing DAG representation presented in Sec. 3.7.

Rust Semantics and Formalizations A number of formalisms for subsets of Rust have been designed, focusing on type soundness results [104–107]. It would be interesting to compare these formal models with the capability analysis and borrowing DAG that our work produces from the compiler.

[104]: Reed (2015), *Patina: A formalization of the Rust programming language*
[105]: Kan et al. (2018), *K-Rust: An Executable Formal Semantics for Rust*
[106]: Wang et al. (2018), *KRust: A Formal Executable Semantics of Rust*
[107]: Weiss et al. (2018), *Rust Distilled: An Expressive Tower of Languages*

Oxide [108] is an in-progress work that formalizes Rust's borrow checker using standard type-checking rules that are proven sound using progress and preservation. Their approach follows the view that a lifetime is a set of loans, as proposed by Matsakis [109]. This is reflected in Oxide's typing rules, which track which reference might point to which others in a way that looks similar to the borrow graph that we compute in our work.

[108]: Weiss et al. (2019), *Oxide: The Essence of Rust*
[109]: Matsakis (2018), *An alias-based formulation of the borrow checker*

In the context of unsafe Rust, an important property is whether a program has undefined behavior or not. While there is not yet an officially accepted formal definition of UB, Stacked Borrows [42] and its follow-up Tree Borrows [43] propose a formal aliasing definition of UB that can be checked at runtime using the Miri interpreter [44]. By assuming that the verified safe code is not called by unsafe code, our encoding of Rust types is based on safety assumptions that are stronger than just absence of UB. One notable difference is that the current consensus is that in Rust, it is not UB to have private reference fields that point to uninitialized memory, as long as such memory locations are not read, while our work assumes that such a case cannot happen. To extend Prusti to verification of unsafe code, it would be necessary to introduce, e. g., new annotations to specify to which extent the usual guarantees of Rust types are weakened in unsafe code.

[42]: Jung et al. (2020), *Stacked Borrows: An aliasing model for Rust*
[43]: Villani (2024), *Tree Borrows: A new aliasing model for Rust*
[44]: (2024), *Miri: An interpreter for Rust's mid-level intermediate representation*

3.10 Conclusions

In this chapter, we presented our verification technique that leverages the properties of Rust's type system to simplify program specifications and automate heap reasoning, making verification accessible to non-experts. The technique in this chapter is based on the *explicit* properties of Rust types; those defined by the Rust language. As a consequence, verification is incomplete when reasoning about types with *implicit* library-defined properties. For example, types with interior mutability or concurrent semantics. In the next chapter, we are going to see how to express some of these library-defined properties with lightweight annotation.

Verification of Safe Clients of Interior Mutability

4

In this chapter, we present our technique for the verification of *safe clients*: safe Rust programs that make use of trusted libraries that may be implemented with unsafe code. These libraries pose several verification challenges due to their ability to implement mutable shared data structures, including concurrent data structures. The examples in this chapter focus on the popular *interior mutability* pattern of Rust to ease the presentation, although the presented verification technique can be applied to any case of shared mutability because it abstracts over the implementation details of the libraries.

Existing automated verification techniques for safe Rust code rely on the strong type-system properties to reason about programs, especially to deduce which memory locations do not change (i.e., are framed) across function calls. However, these type guarantees do not hold in the presence of *interior mutability* (e.g., all concurrent data structures) and other behaviors made possible by implementing safe libraries using unsafe code. As a consequence, existing automated verification techniques for safe code such as Prusti [29] and Creusot [86] are either unsound or fundamentally incomplete if applied to this setting. In this work, we present the first automated technique to verify some safe usages of real-world Rust libraries, including those implemented with unsafe code, that does not require changing the signature of the existing library methods[1]. At the core of our approach, we identify a novel notion of *implicit capabilities*: library-defined properties that cannot be expressed using Rust's types. We propose a new annotation to specify these capabilities, with a meta-theory soundness proof for their semantics and a first-order logic encoding suitable for program verification. We implemented our technique in a verifier called Mendel and used it to prove absence of panics in simple Rust programs that make use of popular standard-library types with interior mutability, among which `Rc`, `Arc`, `Cell`, `RefCell`, `AtomicI32`, `Mutex` and `RwLock`. Our evaluation shows that these library annotations are *useful* for verifying usages of real-world libraries, and *lightweight* enough to require zero client-side annotations in many of the verified programs.

[29]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*

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

1: An example of this would be adding ghost arguments and return values to describe the effect of the mutations.

4.1 Introduction

Rust's ownership type system offers strong guarantees, such as memory safety, absence of data races, absence of dangling pointers and, in general, absence of undefined behavior (UB). In the safe language fragment of Rust, these properties are statically guaranteed by the compiler, making it possible for verification techniques and tools to build upon them [29, 62, 86, 103]. This is achieved by associating an *exclusive capability* [2] to all mutable references and non-borrowed types, and a *shared capability* to all immutable references, as done in the capability analysis of Sec. 3.7. The former capability guarantees write access and non-aliasing, while the latter read access and immutability. In our work, we call these type

[29]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*

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

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

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

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

capabilities *explicit*, because they are fundamental properties defined by the Rust language.

The explicit capabilities of Rust types are too restrictive to implement certain behaviors: cyclic data structures such as doubly linked lists, concurrency, shared mutable state such as a global cache, and so on. To overcome this, many Rust libraries use unsafe Rust in their implementation to offer *interior mutability* in their API. This is the only way by which safe Rust libraries can implement shared mutable state. If it were not for the unsafe code in the library implementation, the shared references would imply transitive immutability, preventing any mutations. However, the additional expressivity for the library developers comes at the cost of losing static guarantees. In fact, in the presence of interior mutability, it is no longer statically known whether shared references may be used for mutations or not, because the explicit capabilities of these library types do not describe the memory locations where the mutations may happen. This is an inherent limitation for existing static analysis and automated verification tools, none of which is able to reason about basic usage of real-world unmodified libraries with interior mutability. For example, the program in Fig. 4.1 shows a simple usage of the `Cell` type of Rust's standard library, which is a container with interior mutability. The `cell_client` function receives a shared reference `c` pointing to an instance of `Cell`. Thanks to interior mutability, despite the immutability of shared references, the function is allowed to increment the cell's content by one with a call to `c.set(..)`. The two `c.get()` calls around the increment simply return a copy of the cell's content. At the end, with an `assert!(..)` statement, the function checks at runtime whether the value read after the increment is exactly one more than the value read before. The assertion never fails at runtime, because the design of the `Cell` library guarantees that its content cannot be modified concurrently as long as there is a `&Cell` instance like `c`, so the only modification happens as a result of the `set` call. The compiler would generate an error as soon as a program tries to send `&Cell` to a different thread. Naively treating `Cell` like any other type would lead to unsoundness in existing automated verifiers, because they would use the immutability properties of shared references to incorrectly prove that `before == after` holds at the point of the assertion. To soundly model the interior mutability of `Cell`, they can only conservatively assume that other threads might interfere at any moment and modify the cell's content, losing all precision. As a consequence, existing automated verifiers are fundamentally incomplete and cannot prove the last assertion, for which they report a verification error.

```rust
fn cell_client(c: &Cell<i32>) {
    let before = c.get();
    c.set(before + 1);
    let after = c.get();
    // Goal: prove that the following never fails
    assert!(before + 1 == after);
}
```

Rust

Figure 4.1: Example of a client of the `Cell` library.

In this work, we provide a technique to reason about safe clients of *real-world unmodified*[2] libraries with interior mutability in an *automated* verifier. The focus on real-world unmodified libraries and automation is

2: That is, without changing the signature of the existing library methods.

aimed at minimizing the manual effort needed to verify existing software. At the core of our work, we identify that Fig. 4.1 and many other usages of interior mutability can be verified by introducing a new notion of library-defined capabilities, which we call *implicit* to oppose them to the regular explicit capabilities guaranteed by the Rust language. In fact, not all Rust types provide explicit capabilities. Raw pointers, for example, are C-style pointers that can be dereferenced only in unsafe code because the type system does not ensure their validity. When using raw pointers, e.g., to implement an API with interior mutability, library developers can still decide to guarantee immutability or uniqueness properties if they wish. When that happens, we say that the library has implicit capabilities associated with its types. While explicit capabilities can be automatically discovered by traversing Rust type declarations (as we exploited extensively in Ch. 3), implicit capabilities cannot, because they are in the best case only described in prose in the documentation of the libraries. For example, the design of the `Cell` library is such that any `&Cell` instance is a witness that the content of the cell is thread-local. This means that every `&Cell` implicitly provides a non-interference capability with other threads that can be exploited for verification, even though the Rust types used in the private declaration of `Cell` are too weak to guarantee this property inherently. Knowing this implicit capability is a key step to verifying the example in Fig. 4.1. Since the instance `c` is available (i.e., not moved-out nor mutably borrowed) throughout the entire implementation of `cell_client`, a verifier can deduce that the cell's content is always thread-local and can be modified only as an effect of the `c.get()` and `g.set(..)` calls. To prove the last assertion, it is then sufficient to rely on contract annotations that specify the functional behavior of the `get` and `set` methods.

Contributions The main contributions of our work are:

1. We identify the notion of implicit capabilities and propose new annotations to specify them on library types and APIs.
2. We present a new verification technique that uses capabilities to reason about safe Rust code in the presence of interior mutability.
3. We present a novel proof technique that relies on basic UB properties to prove what the unsafe implementation of a sound library cannot do. Using this proof technique, we prove the correctness of our model of a core subset of our capabilities.
4. We show an encoding of our reasoning technique to first-order logic, suitable for automation using an SMT-based verification toolchain.
5. We implement our reasoning technique in a deductive verifier for Rust called Mendel, showing with an evaluation that our technique is (1) *useful*, as it supports popular types with interior mutability defined in the standard library, and (2) *lightweight*, because it requires near-zero annotations on the client side of the simple programs in our evaluation.

Outline The rest of this chapter is structured as follows. In Section 4.2, we present the verification challenges of our setting and we identify the core notion of implicit capability. In Section 4.3, we present our implicit capabilities, introducing library annotations to specify them and semantic

rules to reason about them. In Section 4.4, we prove the correctness of the semantics of a core subset of our capabilities, introducing our *proof by semantics-preserving transformation* technique to derive properties of sound Rust abstractions from well-established properties of safe Rust. In Section 4.5, we present a first-order logic encoding of our capability-based reasoning technique. In Section 4.6, we implement our work in a deductive verification tool and evaluate it on simple clients of popular standard-library types. In Section 4.7, we discuss related techniques, then we conclude in Section 4.8.

4.2 Problems

In this section, we present the key problems of verifying Rust clients that make use of libraries exposing interior mutability.

4.2.1 Shared Mutable State

Shared references in Rust normally provide a transitive immutability guarantee, in that every value reachable with safe code[3] starting from the shared reference cannot be modified as long as the reference is alive. In this work, we call those memory locations *stable*. When a library is implemented with unsafe code, this immutability property is not always transitive, posing a verification challenge presented in the following example.

3: i.e. without dereferencing raw pointers or entering the special unsafe cell types

Consider the two functions `option_client` and `cell_client` in Fig. 4.2, which are written in safe Rust and only use libraries with a safe API. Both functions take as parameter a shared reference to either an `Option` type or to a `Cell` type, where the `Option` type represents an optional value and the `Cell` type can be seen as a container of size 1 with interior mutability. The implementations of `option_client` and `cell_client` follow a similar structure. They start by calling what we call a *query method* on `x`: both `is_some` and `get` are side-effect free, deterministic, non-diverging, and make it possible to observe the state of their receiver. Then, a copy of the (duplicable) shared reference `x` is passed to a function call whose implementation is unknown, and finally the state of `x` is queried and compared with other observations.

```rust
fn use_option(x: &Option<i32>) { /* ??? */ }

fn option_client(x: &Option<i32>) {
    let a = x.is_some();
    use_option(x);
    let b = x.is_some();
    let c = x.is_some();
    assert!(b == c); // Succeeds
    assert!(a == b); // Succeeds
}
```

Rust

```rust
fn use_cell(x: &Cell<i32>) { /* ??? */ }

fn cell_client(x: &Cell<i32>) {
    let a = x.get();
    use_cell(x);
    let b = x.get();
    let c = x.get();
    assert!(b == c); // Succeeds
    assert!(a == b); // Might fail
}
```

Rust

Figure 4.2: Example of a client of the `Option` library (left) and of the `Cell` library (right).

Because of the transitive immutability property of shared references, a naive expectation might be that `use_option` and `use_cell` cannot

modify the state of `x`, so that all assertions in the code should always succeed. With the `Option` type, this is indeed guaranteed by the type system and library API design. This property is already exploited in the verification technique used by existing Rust verifiers, which allow a developer to mark the query method `is_some` as, e.g., `#[pure]` to express that its result is a mathematical function of the stable value reachable from the arguments [29]. This implies that the result of `x.is_some()` remains the same as long as `x` is alive. It is then easy for a verifier to prove that the results `a`, `b` and `c` of the query are equal even if there is an unknown usage of `x` in between.

[29]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*

The same reasoning, however, does not apply to the `cell_client` function. Since the `Cell` library internally uses unsafe code to provide interior mutability, `use_cell` can modify the state of the cell using the API method `x.set(42)`. The challenge for verification techniques is that query methods such as `get` can now depend on *unstable* values (as opposed to stable) reachable from their arguments. Contrary to the example with `Option`, the correct verification outcome is now to report that the final runtime check `a == b` might fail, but that `b == c` succeeds because concurrent threads cannot access the `Cell`. How can this be achieved? If a verifier naively ignores interior mutability and treats `get()` analogously to `is_some()`, it would be unsound. This is because the verifier would believe that `a == b` holds, contradicting the behavior of `use_cell`. Alternatively, if a verifier treats `get()` non-deterministically, it would be unable to prove that `b == c` holds. Neither of these two options is sufficient to verify the `cell_client` function. What is missing is a technique to soundly and precisely model the framing properties of these libraries: whether and when the results of query methods are preserved or not. Hard-coding that the `Cell` type has special framing rules in a verifier might be sufficient to verify this example, but such an approach would not be general and would face identical limitations when reasoning about other types with interior mutability, including cases where a `Cell` type is part of a larger data structure.

Key Problem K1: How to model framing properties of types with interior mutability?

4.2.2 Library Properties

In order to verify clients of libraries exposing interior mutability, a key requirement is to know the library-defined immutability and uniqueness properties of its types. Consider for example the signature of the `refcell_client` function in Fig. 4.3. `RefCell` is a type with interior mutability, which allows clients to access the cell's content using flexible runtime aliasing checks instead of rigid compile-time borrow checks. When a client asks for a reference to the data of the cell, for example with a `borrow` or `try_borrow` method call, the library returns a `Ref` or `RefMut` auxiliary type, where `Ref` provides a read-only view of the data and `RefMut` also provides write access. A key property of the library is that its API (as opposed to the Rust language) ensures that at any point, there can be either multiple `Ref` instances associated with a certain cell or, alternatively, at most one `RefMut`. As a consequence, `Ref` implies

```rust
fn use_refcell(x: &RefCell<i32>) { /* ??? */ }

fn refcell_client(x: &RefCell<i32>, y: RefMut<i32>) {
    let Ok(a /*: Ref */) = x.try_borrow() else { return; };
    let before: i32 = *a;
    use_refcell(x);
    let after: i32 = *(x.borrow());
    assert!(before == after); // Succeeds
    assert!(x.as_ptr() as *const _ != y.deref() as *const _); // Succeeds
}
```

Rust

Figure 4.3: Example of a client of the `RefCell` library. At runtime, both assertions never fail, so a complete verifier should be able to verify them. The first line in the implementation, `let Ok(..) = ..`, makes the function return in case the `try_borrow` method fails due to `x` being already borrowed.

immutability of the cell's content while `RefMut` ensures non-aliasing with other `Ref` or `RefMut` instances. These properties can be modeled as capabilities of the `Ref` and `RefMut` types.

In order to prove that the two assertions in `refcell_client` never fail at runtime, a verifier must be aware of the implicit capabilities of the `Ref` type. The first assertion checks whether the content of `x` changed across the `use_refcell` call, whose implementation is unknown, while the second uses a pointer comparison to check whether `x` and `y` are aliasing the same memory location. At the beginning of the function body, the `try_borrow` call tries to obtain a `Ref` instance to access the data of `x`. If it fails, for example because there already exist some `RefMut` instances associated with the data of `x`, the function enters the `else` branch and terminates due to the explicit `return` statement. If it succeeds, then `try_borrow` returns an instance that remains alive until the end of the function. The implicit capabilities of `a` are enough to ensure that (1) across the call to `use_refcell`, the content of `x` does not change, proving that the first assertion always succeeds, and (2) that `a` and `y` do not alias the same data. Since `try_borrow` ensures that the returned `a` surely aliases the data in `x`, it follows that `y` does not alias `x`. These two deductions are sufficient to prove that the last two assertions in the program do not fail. The reasoning is high-level enough to match what a Rust developer would think when analyzing this function. In particular, it is modular in that it does not rely on implementation details of the `RefCell` library, but just on immutability and non-aliasing properties described in its API.

Without knowing the implicit capabilities of the `RefCell` library, a verifier would have to conservatively assume that any usage of any type with interior mutability might potentially alias and modify the content of any other type with interior mutability. That is, `use_refcell(x)` should be conservatively modeled as possibly modifying `x`, and `y` as possibly sharing the same content of `x`, forcing the verifier to report a verification error for the two assertions in the program. Such a verifier would be sound, but unable to prove any interesting property regarding the content of types with interior mutability. Instead, to be useful a verifier should be able to leverage on the guarantees provided by these libraries.

Key Problem K2: How can developers specify the implicit capabilities of libraries?

Key Problem K3: How can implicit capabilities be used by automated verification tools?

4.2.3 Conditional Library Properties

The library properties used so far to reason about Rust programs hold for *all* instances of certain library types, independent of the values that the instance contains. Certain programs, however, show that it is possible for some library properties to hold only for certain instances of a library type.

Consider for example the function in Fig. 4.4. The parameters are of type `Arc`, a popular thread-safe reference-counted pointer that makes it possible to share data between threads. At the beginning of the function, the two parameters `x` and `y` might share the same memory, and might even share the same memory of `Arc` instances used by other threads. The `Arc::strong_count` function makes it possible to read the current value of the reference counter. The `Arc` type has a second counter for a weak pointer but the API does not provide a way to atomically observe both at once. So, in this work, we assume that the weak counter is always at zero[4]. If the counter of `x` is exactly 1, then it means that `x` does not share its content, neither with `y` nor with any other `Arc` instance used in other threads. In other words, when the counter is 1, an `Arc` instance has unique access to its own counter and the contained data. This implies that the two assertions in the first branch always succeed, because uniqueness provides the necessary immutability and non-aliasing properties to deduce so. When the counter is not 1, however, multiple `Arc` instances can access and modify the same counter, even from other threads. Thus, a verifier should report the first assertion in the second branch as potentially failing, because nothing prevents `x` and `y` from sharing the same memory. Moreover, the last assertions should also be reported as potentially failing, since other threads might suddenly bring the counter to 1 just after the evaluation of the branch condition.

4: This can be verified by adding a `false` precondition to all the API functions that might create a weak reference (`Arc::downgrade`, `Arc::new_cyclic`).

Key Problem K4: How can the answers to K2 and K3 be extended to implicit capabilities that depend on runtime conditions?

```rust
fn arc_client(mut x: Arc<i32>, y: Arc<i32>) {
    if Arc::strong_count(&x) == 1 {
        assert!(Arc::strong_count(&x) == 1); // Succeeds: the counter is stable
        assert!(Arc::as_ptr(&x) != Arc::as_ptr(&y)); // Succeeds: non-aliasing
    } else {
        assert!(Arc::as_ptr(&x) != Arc::as_ptr(&y)); // Fails: might be aliasing
        assert!(Arc::strong_count(&x) != 1); // Fails: the counter is unstable
    }
}
```

Rust

Figure 4.4: Example of a client of the `Arc` library. The assertions in the first branch never fail at runtime, while those in the second branch might fail. Thus, a verifier should be able to verify the first branch, and should report verification errors for the two assertions in the second branch.

4.3 Approach

In this section, we present our verification technique based on the concept of capabilities, providing an annotation methodology to declare the implicit capabilities of library types. The verification technique of this work can be used to prove the functional correctness of safe Rust clients, building upon the memory-safety guarantees of safe Rust. In particular, we rely on the correctness of the Rust compiler in order to kick-start our verification technique, by assuming that the explicit capabilities of the Rust types hold whenever these are initialized and not mutably borrowed.

4.3.1 Core Capabilities

In this subsection, we present the *core* of our capability-based approach to verify clients of libraries with interior mutability. We start by showing at a high level the final results of our technique: the annotations and reasoning steps on the program of Fig. 4.5, which is a simplified version of Fig. 4.3 that avoids calling the possibly-panicking `x.borrow()` call. Then, we proceed by defining the core capabilities of our approach and their properties, showing how these finally justify these results.

4.3.1.1 Motivating Example

```rust
#[capable(&mut self => writeRef(self.as_ptr()))]
impl<T> RefCell<T> {}

#[capable(&self => readRef(self.refcell().as_ptr()))]
impl<'b, T> Ref<'b, T> {}

fn use_refcell(x: &RefCell<i32>) { /* ??? */ }

fn refcell_client(x: &RefCell<i32>, y: RefMut<i32>) {
    let Ok(a /*: Ref */) = x.try_borrow() else { return; };
    let before: i32 = *a;
    use_refcell(x);
    let Ok(b /*: Ref */) = x.try_borrow() else { return; };
    let after: i32 = *b;
    // Both following assertions succeed
    assert!(before == after);
    assert!(x.as_ptr() as *const _ != y.deref() as *const _);
}
```

Rust

Figure 4.5: Example of a client of the `RefCell` library, based on Fig. 4.3. The full annotations on the `RefCell` library can be seen in the test suite of the evaluation.

At the beginning of the code in Fig. 4.3, the `#[capable(..)]` annotations declare two implicit capabilities of the `RefCell` library. The first, on `RefCell`, states that any `&mut RefCell` instance is capable of obtaining a mutable reference to the content of `RefCell`. Or, in other words, mutable instances of `RefCell` hold the unique capability for mutating their content. This annotation uses the existing `RefCell::as_ptr` API method to identify the content of the `RefCell` by address. The second annotation, on `Ref`, states that any `&Ref` instance is capable of obtaining a shared reference to the content of the associated `RefCell`. In other words,

`Ref` instances hold the capability to read the content of the `RefCell` with the guarantee that it is immutable. To formalize this property, the annotation uses an auxiliary specification method `Ref::refcell` (not shown in the figure) that models the `RefCell` instance associated with a `Ref`. Using these two annotations, it is possible to prove panic-freedom of the programs in two key steps. First, the `a` instance of type `Ref` is alive across the `use_refcell(x)` call. Because of the implicit capability of `a`, the content of `x` is immutable across the call. This is the key property that, combined with standard functional specifications for the other methods in the program, makes it possible to verify the first assertion on the program. The second step is to notice that, before the last assertion, the instances `a` and `y` are both alive. The instance `a` still holds a capability that implies immutability of `x`'s content. At the same time, the instance `y` of type `RefMut` holds an implicit capability that implies exclusive mutable access to `y`'s content. Since these two capabilities are incompatible, it follows that the content of `x` *must* be at a different location than the content of `y`, which proves the second and last assertions in the program. Without the library capability annotations, neither of the two assertions could be verified.

In order to verify this example and other Rust programs, in the rest of this section we will define novel capabilities that can be annotated on library types. These capabilities make it possible to reason about the content of types with interior mutability, using implication, non-aliasing and immutability properties that are based on the properties of Rust reference types.

4.3.1.2 Kinds of Core Capabilities

To verify programs such as the one shown in Fig. 4.3, we define the following two kinds of capabilities. We call them *core* capabilities because they are fundamental capabilities inherent to the Rust language.

- `readRef` corresponds to the capability of Rust shared references. It provides *shared* read-only access, with the guarantee that the target memory location is not modified via aliases.
- `writeRef` corresponds to the capability of mutable references, and of fully initialized non-borrowed types[5]. It provides *exclusive* read and write access.

A type instance is said to *hold* one of these core capabilities *if and only if* it is *sound*[6] for the API of the type to return the shared or mutable reference corresponding to the capability, without causing side effects other than borrowing the receiver instance[7]. This definition makes it easy to determine some of the core capabilities of a library: it is enough to identify an existing safe method of the API that returns a reference while satisfying the side-effect-free requirements. For example, this is the case of `RefCell::get_mut`: a method that given `&mut RefCell` always returns a mutable reference to the content of the type, even though the type has interior mutability. Thus, an instance of `&mut RefCell` holds a `writeRef` capability for its content. In practice, we observed that implementations of the popular `Deref` and `DerefMut` traits provide similar side-effect-free methods for many real-world library types. Our tool does not make any assumption about the implementation of the

5: It would be possible to introduce a third core capability kind to distinguish between mutable references and non-borrowed types, but we did not find practical advantages in the use cases that we considered.

6: Soundness, here, refers to the *library soundness* principle of Rust, described in Sec. 2.1.5.

7: In Sec. 4.4 we define more precisely this special class of methods, calling them *conversion* methods.

`Deref` and `DerefMut` traits, but for the verification users it might be convenient to start from these traits when looking for conversion methods.

4.3.1.3 Library Annotations

The immutability implication of `Ref` instances and the exclusive mutability implication of `RefMut` are properties of the `RefCell` library, informally stated in the documentation. To make it possible for library designers to formally state them, we developed a new annotation to express the implicit capabilities of library types. Each annotation is associated with a library type and has two components: one describing the required borrowing state of the library type, because mutably and immutably borrowed library types might hold different capabilities, and one describing the capability held by the library type. The syntax of the annotation and its components are the following:

```
#[capable(⟨source_kind⟩ => ⟨target_kind⟩(⟨addr⟩))]
impl ⟨type⟩ {}
```
Rust

- ⟨type⟩ is the Rust type of the source instance.
- ⟨source_kind⟩ is the required borrowing state of the type. Possible values are `&self`, representing an immutably borrowed type, and `&mut self`, representing a mutably borrowed or non-borrowed type.
- ⟨target_kind⟩ is the capability kind of the target instance. Possible values are `readRef` and `writeRef`, which correspond to the capabilities of shared and mutable references, respectively.
- ⟨addr⟩ is a pure Rust expression of type raw pointer that identifies the target instance by address.

For example, consider the type `Ref<T>` used for the variable `a` in Fig. 4.5. The `Ref` type implements a method `Deref::deref` with signature `fn deref(&self) -> &T` that returns a shared reference pointing to the associated `RefCell`'s content. This testifies that any `&Ref` instance holds the capability corresponding to a shared reference for the content of the associated `RefCell` instance. Thus, `&Ref` can be annotated to hold a `readRef` capability, as expressed by the second library annotation in the example. In fact, we will discuss later in Sec. 4.4 that the core capabilities `readRef` and `writeRef` can be seen as representing the capability of obtaining a shared or a mutable reference through the library API, like `Ref`'s `deref` method does.

The annotation above declares capabilities that hold unconditionally, but there exist library types whose capabilities depend on runtime conditions. An example is the `Rc` type, a single-threaded reference-counting pointer that uses interior mutability to update its counter when cloning the type. A `&mut Rc` instance has a `writeRef` capability for its content, but only if (1) the counter of references is exactly 1, and (2) the counter of the weak references is zero. To express this conditional capability, we provide a second capability annotation with the following syntax. The new ⟨cond⟩ component is a pure Rust expression of boolean type, which expresses the condition that needs to be satisfied in order to for the source

instance to hold the target capability. Our technique does not require this condition to be stable, but only stable conditions are useful in practice. This is because target capabilities depending on unstable conditions are usually[8] lost when modeling interference from other threads.

8: In theory, libraries with mutually exclusive unstable conditions might still be combined to preserve a target capability.

```
#[capable(⟨source_kind⟩ if ⟨cond⟩ => ⟨target_kind⟩(⟨addr⟩))]
impl ⟨type⟩ {}
```
Rust

4.3.1.4 Implication Properties

The definitions of our capabilities are not independent; some can be obtained by *consuming* others. For example, given a mutable reference one can always obtain a shared reference, but without ever holding both the shared and mutable capabilities at the same time. Similarly, given a mutable reference to a structure in Rust, it is always possible to obtain a mutable reference to one of its visible fields, but both mutable references are never obtainable at the same time. For the moment, we model the property that a capability is sufficient to obtain another as an *implication* between capabilities. Later, in Sec. 4.3.1.5 and Sec. 4.3.1.6, we will introduce a mechanism to identify which capabilities can be obtained at the same time.

For a fixed memory location, the diagram in Fig. 4.6 represents with a solid directed edge the `writeRef` ⇒ `readRef` implication, motivated by the fact that in safe Rust clients, it is always possible to convert a mutable reference `x` into a shared one using the expression `&*x`.

The Rust type of a capability determines further implications. Capabilities of a tuple, structure or enumeration type imply capabilities for the visible fields, and capabilities of a reference imply capabilities for the reference's target. For example, the capability of obtaining a mutable reference to a local variable `x` of type mutable reference implies the capability of obtaining a mutable reference pointing to `x`'s target. The Rust expression to do so is `&mut **x`. Thus, we model this property with the implication `readRef(x)` ⇒ `readRef(*x)`, where the arguments `x` and `*x` represent the memory location at which `x` and `*x` are stored. An analogous rule holds for shared references and other types with fields, for which, given a reference to a type instance, it is always possible to obtain a reference to its visible fields using an expression like `&x.f` or `&mut x.f`. Overall, these rules are presented in the first two rows of Table 4.1. The only case where the kind of the capability is different than the initial one is when `x` is a shared reference. In this case, the capability of obtaining

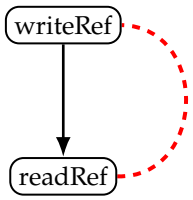


Figure 4.6: Diagram of the core capability implication (arrow edge) and incompatibility (dashed edge, explained later) for common types (non-zero-size types that do not implement `!Unpin`). The capabilities of shared and mutable references correspond to `readRef` and `writeRef`, respectively. For a fixed program point and memory address, implications are between capability facts with the same root, while incompatibilities are between capability facts with different roots.

Table 4.1: Table of the implications between the capability of a source instance `x` and the capability of its fields or dereferences. We will discuss the last line in Sec. 4.3.2.

Source capability	Target capability		
	if `x.f` is visible	if `x: &mut T`	if `x: &T`
`writeRef(x)`	`writeRef(x.f)`	`writeRef(*x)`	`readRef(*x)`
`readRef(x)`	`readRef(x.f)`	`readRef(*x)`	`readRef(*x)`
`localRef(x)`	`localRef(x.f)`	`localRef(*x)`	`readRef(*x)`

a mutable reference to `x` implies the capability of obtaining a *shared* reference to `*x`, mirroring the Rust rule by which shared references behind mutable references cannot be used for mutations. In the case of enumeration types, the discriminant is modeled as a field, and each other field implication only holds under the condition that the value of the discriminant corresponds to the variant in which the field is defined.

4.3.1.5 Syntactic Representation of Capabilities

To aid the presentation of the following properties, we need a syntax to represent capabilities. We do so by defining that the capabilities of a type T at a program point have in total two parameters and can be expressed as the boolean function application $\langle \text{kind} \rangle_T(r, a)$. This represents the capability of a certain kind associated with the instance of type T at address a[9]. The r parameter expresses the syntactic place from which the capability originates, among a set of places that we later define and call *roots* in Sec. 4.4. This parameter is used to model separation-logic-like *disjointness* properties of some memory locations associated with capabilities with different roots (cf. Sec. 4.3.1.6). For the moment, it is sufficient to know that the set of all initialized local variables is a definition of root places that is sound, although incomplete for some programs with moved-out fields. When convenient for the presentation, we omit T, r, or a if their value is made clear by the context. Moreover, when a is written as a Rust place, for example `x.f`, the value of a refers to its address, which in the example corresponds to the Rust expression `addr_of!(x.f)`.

9: The conditions under which a capability holds can additionally be expressed using implications, as in $\langle \text{cond} \rangle \Rightarrow \langle \text{kind} \rangle_T(r, a)$.

As an example, consider the function in Fig. 4.7. Both arguments `x` and `y` are valid roots in the only program point of the empty body. The first root, `x`, determines two capabilities `writeRef("x", x)` and `readRef("x", x)`, where the first parameter `"x"` identifies syntactically the root place `x`, while the second parameter `x` identifies the memory location pointed by the reference `x`. The second root, `y`, determines three more capabilities as shown in the figure. How exactly all these capabilities are generated will be discussed in detail later in Sec. 4.5. As anticipation, each root at each program point generates exactly one capability; `writeRef("x", x)` and `readRef("y", y)` in the example. The remaining capabilities are a consequence of an implication property between capabilities: `writeRef("x", x)` determines `readRef("x", x)`, while `readRef("y", y)` determines `readRef("y", y.0)` and `readRef("y", y.1)`. Note that the root parameter is preserved across these implications. As we are about to present, this enables defining non-aliasing properties between capabilities originating from different roots.

```rust
fn capabilities(x: &mut u32, y: &(u32, u32)) {
    // roots: [x, y]
    // capabilities: {
    //   readRef("x", x), writeRef("x", x),
    //   readRef("y", y), readRef("y", y.0), readRef("y", y.1)
    // }
}
```

Rust

Figure 4.7: Rust function annotated with the capabilities originating from the two arguments.

4.3.1.6 Non-Aliasing Properties

There are cases where the definitions of the capabilities are incompatible if they refer to the same type and memory location. For example, in Rust it is never be possible to have at the same time a shared and a mutable reference pointing to the same (non-zero-size) type instance. This condition can be exploited by a verification tool to infer non-aliasing properties, because if a library ensures `readRef` or `writeRef` for a memory location a_1 and, at the same program point, `writeRef` for a memory location a_2 then a verifier can deduce that $a_1 \neq a_2$. However, how the capability originates matters. For a fixed program point, incompatibilities only hold between capabilities that originate from *different* places:

$$\texttt{writeRef}(r_1, a_1) \land \texttt{readRef}(r_2, a_2) \land r_1 \neq r_2 \Rightarrow a_1 \neq a_2$$

This property of our capabilities is represented by the dashed edge in Fig. 4.6. We later use the syntax $\texttt{writeRef}(r_1) \nleftrightarrow \texttt{readRef}(r_2)$ to represent this formula. In contrast, the implications that we presented in Sec. 4.3.1.4 hold between capabilities that originate from the same place.

Why is it necessary to restrict this property to capabilities with different origins? This can be seen using the example in Fig. 4.7. In the program, there are two local variables that are alive: `x` and `y`. The former generates a `writeRef("x", x)` capability, while the latter generates a `readRef("y", y)` capability. Since `writeRef` and `readRef` are incompatible, it follows that `x` and `y` must point to different memory locations as expected. However, we previously stated that there is also an implication between `writeRef` and `readRef`. In the example, `writeRef("x", x)` implies `readRef("x", x)`. An unsound non-aliasing definition that ignores the root parameter would also apply between these two capabilities, because the former is a `writeRef` while the latter is a `readRef`. The consequence would be that a verifier would deduce that the address pointed by `y` is not the address pointed by `y`, which is a contradiction. The restriction regarding the root parameter is designed to avoid this case.

One limitation of this non-aliasing definition based on root arguments is that it does not model the non-aliasing property of different fields of the same structure, when both fields originate from the same root. This is an incompleteness. In such cases, a sufficient workaround is to briefly create two references pointing to those fields, expiring them immediately after. This forces for a moment the reasoning technique to model the two fields using different root arguments, from which the verifier can deduce their non-aliasing property, which can then be reused even after the expiration of the references.

4.3.1.7 Immutability Properties

With the rules presented so far, a verifier can prove only the non-aliasing properties resulting from the incompatibilities between capabilities. However, a verifier also needs to deduce which values do not change, i. e., are *framed*, across statements. To achieve this goal, we apply the capability reasoning *across* each statement. First, we define that a place is available *across* a statement if the place is available before the statement but is not used by the statement. This can be determined with a syntactic check. The capabilities held by the type instance of that place are also said to be held across the statement, including those obtained via implication properties. Then, as a framing rule, we define that a `readRef` capability on a primitive type held across a statement implies immutability of the associated memory location. The intuitive motivation is that `readRef` is the capability corresponding to a shared reference, and primitive types cannot have interior mutability. Thus, noticing that a shared reference is alive and unused across a statement is sufficient to deduce that the target memory location cannot be mutated, not even by concurrent threads. We discuss the motivation of this rule in more detail later in Sec. 4.4.

To see an example where this immutability rule is applied several times, consider again the `use_refcell(x)` statement of Fig. 4.5. The `a` root place is available across the statement, thus `writeRef(a)` also holds across it. By the implication rules of capabilities, `readRef(a)` holds as well. From here, there are many possible deductions that a verifier can make. First, since `readRef(a)` is available across a statement then the instance `a` is known to be immutable across `use_refcell(x)`. Second, because of the library annotations of `Ref`, `readRef(a)` implies `readRef(a.refcell().as_ptr())` across the statement. Third, the immutability rule can be applied again, but this time on `readRef(a.refcell().as_ptr())`. As a result, a verifier can deduce our goal: the memory location at `a.refcell().as_ptr()`, which is the content of the `RefCell`, is immutable across `use_refcell(x)`. While in general the reasoning needs to consider the case where the evaluation of `a.refcell().as_ptr()` might change across the statement, in this case the immutability of `a` and the purity annotations on the following method calls are sufficient to guarantee that the evaluation does not change. These details regarding the evaluation order are discussed later in Sec. 4.5.

4.3.2 Extended Capabilities

The core capabilities presented so far have immutability and non-aliasing properties that coincide with the properties of shared and mutable references. However, those capabilities are not expressive enough to annotate and verify usages of types of the standard library that, for soundness of their design, cannot have the same properties as Rust references.

4.3.2.1 Motivation

Consider for example the programs in Fig. 4.8. The code makes use of `Cell`, a library type with interior mutability that makes it possible to read and replace the content of the cell via shared references. The verification goal of these examples is to prove that the last assertion in the program on the left never fails at runtime, while the one on the right might. The key step to prove is that the `*x = 100` statement does not mutate the content of the cell pointed by `c`. Intuitively, this holds because (1) the `Cell` library never leaks to its clients a reference pointing to the content of the cell[10], and (2) `Cell` is thread-unsafe, so it cannot be modified by other threads while a function is holding `c`. The latter property is guaranteed by the compiler, under the assumption that all Rust libraries are sound. Knowing both properties, a verifier can deduce that (1) the `x` reference being modified in `*x = 100` does not point to the content of the cell `c`, and (2) `c` cannot be modified concurrently during the assignment. Thus, across the assignment to `x` the content of `c` does not change, from which a verifier can deduce that the assertion in the left program always succeeds. Regarding the program on the right, nothing prevents the `unknown()` call from obtaining a shared reference aliasing `c`, through which the call can modify the cell. However, how can we model the capabilities associated with the content of the cell? Declaring that each `&Cell` instance holds a `readRef` capability for its content would be unsound! In fact, such annotation would imply that across the `unknown()` call the cell's content is immutable, while it might actually change at runtime. The core capabilities are not expressive enough to annotate and verify this kind of example. What we need are more fine-grained capabilities than `readRef`, so that we can describe the cell's content using something that is similar to `readRef`, but weaker.

10: This is a property guaranteed by the API of `Cell` and necessary for the library to be sound (cf., Sec. 2.1.5). Violating this would be an important bug of the library because it would allow safe clients of `Cell` to have UB.

```rust
fn client_cell_1(c: &Cell<i32>, x: &mut i32) {
    c.set(42);
    *x = 100;
    assert!(c.get() == 42); // Succeeds
}
```

Rust

```rust
fn unknown() { /* ? */ }

fn client_cell_2(c: &Cell<i32>) {
    c.set(42);
    unknown();
    assert!(c.get() == 42); // Fails
}
```

Rust

Figure 4.8: Example of two clients of the `Cell` library. The assertion in the program on the left never fails at runtime because `Cell` is thread-unsafe and never returns a reference pointing to its content. In terms of extended capabilities, across `*x = 100` the caller holds a `local` and a `noWriteRef` capability for the content of `c`. Instead, the assertion in the program on the right can fail because the `unknown()` call might obtain a reference aliasing `c`.

4.3.2.2 Beyond Core Capabilities

To address the limitations of the core capabilities, we define new *extended* capabilities, which describe more fine-grained properties that cannot be expressed using Rust types. With them, we classify at a high level the capabilities in the following four groups. The new capabilities are presented individually shortly later.

1. **Core capabilities**, corresponding to the explicit capabilities of Rust references. These capabilities provide an abstraction that is

necessary to connect our capability reasoning to the actual Rust types that are used in function signatures.

2. **Fine-grained capabilities** to read, write, express immutability, unique access, or thread-local ownership (`read`, `write`, `immutable`, `unique`, `local`). These are fundamental properties used across many program reasoning techniques, such as the type system of Pony [75] or experimental type systems for C# [110]. Compared to the core capabilities, `read` and `write` are weaker definitions that lack the immutability and non-aliasing guarantees of shared and mutable references, respectively. Likewise, `unique` is weaker than `writeRef`, but for a subtle reason that we will discuss in Sec. 4.4: `writeRef` implies that it is possible to obtain a mutable reference through the library API, while `unique` does not.
3. **Deny capabilities** (`noReadRef` and `noWriteRef`), expressing that there *cannot* exist references to a certain memory location. This makes it possible to deduce (1) non-aliasing properties between existing references used in safe Rust and memory locations managed by libraries, and (2) immutability of memory locations across assignments to mutable references, as done to verify the example in Fig. 4.5.
4. **Combinations of other capabilities**. This is the case of `localRef`, defined as the conjunction of `local` and `readRef`, for convenience.

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

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

Our capabilities are a pragmatic choice, which gives us all the properties we need for our solution to work. Other choices are possible. For instance, we will define later that the capability to write to a memory location (`write`) implies the capability of reading from the same memory location (`read`). This special implication can be avoided by defining that the `write` capability is the conjunction of two capabilities: `read` and a new, more atomic, write capability that does *not* imply `read`. We do not do so for simplicity. From our experience, the capability framework that we propose to reason about Rust code is flexible and could also be instantiated with many variations of the capability definitions.

Using the new capabilities, in addition to `readRef` the content of a `&Cell` instance can be modeled as `local`, `noReadRef` and `noWriteRef`: the first expresses that the cell's content is thread-local, while the other two express that the library API never leaks shared or mutable references pointing to the cell's content. The capabilities of a `&mut Cell` instance are slightly stronger: the content of the cell can additionally be annotated with the `writeRef` capability to express that the `&mut Cell` instance can be used to obtain a mutable reference to the content of the cell. We will see how this implies more fine-grained capabilities such as `unique` and `local`. All these library annotations are shown in Fig. 4.9, which additionally includes annotations for a few methods of the `Cell`'s API. The `as_ptr` and `get` methods are marked with special purity annotations, which will be discussed later in Sec. 4.3.3, while `new` and `set` are annotated with postconditions expressing that after the call, the cell contains the value that was passed by argument. These library annotations on `Cell` are sufficient to verify the program on the left of Fig. 4.8, while still reporting the expected verification error for the program on the right.

The `writeRef` capability held by `&mut Cell` instances, in particu-

```rust
#[extern_spec]
#[capable(&self => local(self.as_ptr()))]
#[capable(&self => noReadRef(self.as_ptr()))]
#[capable(&self => noWriteRef(self.as_ptr()))]
#[capable(&mut self => writeRef(self.as_ptr()))]
impl<T> Cell<T> {
    #[pure_memory]
    pub fn as_ptr(&self) -> *mut T;

    #[ensures(deref(result.as_ptr()) ==== value)]
    pub fn new(value: T) -> Cell<T>;

    #[ensures(deref(self.as_ptr()) ==== value)]
    pub fn set(&self, value: T);
}

#[extern_spec]
impl<T: Copy> Cell<T> {
    #[pure_unstable]
    #[ensures(result ==== deref(self.as_ptr()))]
    pub fn get(&self) -> T;
}
```

Rust

Figure 4.9: Example of the capability and contract annotations on the `Cell` type of the standard library. `#[extern_spec]` states that we are attaching trusted contract annotations to an existing library API. The figure contains two `impl` blocks because in the second the `T` parameter has an additional `Copy` trait restriction; this is a design choice of the library API. All `#[capable(..)]` annotations are trusted by the verifier. `pure_memory` and `pure_unstable` are two purity annotations discussed later in Sec. 4.3.3. The method `as_ptr` acts as a model of the memory location of the cell's content, so that other contracts can refer to it. `====` is a special structural equality operator, discussed in Sec. 4.3.3, that includes the target memory address of reachable references in the comparison.

```rust
fn unknown() { /* ? */ }

fn client_cell_3(c: &mut Cell<i32>) {
    c.set(42);
    unknown();
    assert!(c.get() == 42); // Succeeds
}
```

Rust

Figure 4.10: Example of a clients of the `Cell` library. The assertion in the program never fails at runtime, because across `unknown()` the caller holds a `writeRef` capability for the content of `c`, which implies a `unique` capability.

lar, makes it possible to verify the example in Fig. 4.10. The function `client_cell_3` is similar to `client_cell_2` in Fig. 4.8, but one type argument is `&mut Cell` instead of `&Cell`. This small difference has a big semantic implication. Since `c` is not used in the `unknown()` call, and since `c` has *exclusive* access to the cell's content, it follows that the `unknown()` call cannot modify the cell's content. We will define later how the `writeRef` capability in the library annotation implies a `unique` capability, whose immutability rules make it possible to verify the assertion in the program.

4.3.2.3 Kinds of Extended Capabilities

All the new capabilities are defined as follows. The diagram in Fig. 4.11 represents the implication and incompatibility rules between capabilities, which will be discussed later.

- `unique` declares that the source instance has unique access to the target instance. This is weaker than `writeRef`, in that `unique` does not imply that a client can obtain a reference to the target instance, but it maintains other properties of `writeRef`, such as non-aliasing. For example, this is the capability that a `&mut Rc` type has for its (weak and strong) reference counters when the weak one is zero and the strong one is exactly one. The API never returns to the client a reference pointing to the value of one of

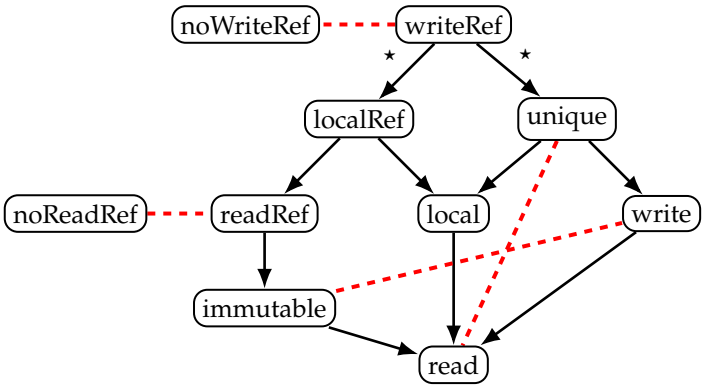


Figure 4.11: Diagram of the extended capability implications (arrow edges) and incompatibilities (dashed edges). All incompatibilities hold only for types that are not zero-size. The implications marked with ⋆ hold only for types that do not implement `!Unpin`. When they do, the edges with ⋆ end at `write` and `readRef` instead.

these counters, which means that the content cannot be modeled using a core capability (`readRef` or `writeRef`). Still, the content can be accessed and modified only via a `&mut Rc` instance, which implies useful non-aliasing and framing properties. The `unique` capability makes it possible to express that.

- `local` declares that the target instance is thread-local, i.e., it is not reachable at the same time from multiple threads. This capability is weaker than `unique`, in that multiple instances can hold a `local` capability for the same memory location, as long as all such instances can be used only by the same thread. The main advantage of `local` is that it guarantees absence of concurrent modifications. For example, this is the capability that a non-write borrowed `&RefCell` type has for its content because the `RefCell` type is not thread-safe (i.e., does not implement `Sync`) and the type checker guarantees that it is never used concurrently. Another example is the `Rc` type, which holds a `local` capability for the location of its reference counters because the library never modifies it concurrently.
- `write` declares that the target instance can be modified by using the source instance. For example, this is the capability that an `&AtomicI32` type implies for its content because it is always possible to modify the content of the type by calling the `store` method. This capability is weaker than `writeRef`, in that (1) `write` does not imply that a client can obtain a reference to the target instance, and (2) it does not imply that the source instance has unique access to the target instance.
- `immutable` declares that the target instance is immutable, as if it were referenced by a shared reference. This capability is weaker than `readRef`, in that `immutable` does not imply that a client can actually obtain a shared reference to the target instance. For example, this is the capability that a `&RefMut` instance holds for the memory location storing the counter of the active borrows (i.e., the number of `Ref` and `RefMut` instances referring to `RefCell`'s content). Such a counter is constant as long as the `RefMut` instance is alive, because (1) a `RefCell` instance can be referred to by at most one `RefMut` instance, and (2) the library ensures that one `RefMut` cannot coexist with other `Ref` instances.
- `read` declares that the target instance can be read by using the source instance. For example, atomic types such as `&AtomicI32`

hold a `read` capability for their content because it is always possible to read that by calling the `load` method. This capability is weaker than `readRef`, in that `read` (1) does not imply that a client can obtain a reference to the target instance, and (2) it does not imply that the target instance is immutable. Still, `read` provides non-aliasing guarantees with respect to memory locations modeled as `unique` because it cannot be that a memory location is uniquely reachable from a place x, but also readable without using x.

- `localRef` represents the capability of an instance that is reachable only from the shared references of one thread. This combination, which is equivalent to the conjunction of `readRef` and `local`, corresponds to the *local reference* type described in [111]. Whenever a local variable or mutable reference is blocked by shared references that have not been used in any function call, both the blocked place and the shared references satisfy the definition of local reference.
- `noReadRef` declares that there exist no shared references to the target instance. For example, this is the capability that a `&Cell` type implies for its content because the library guarantees to never leak a reference to the internal data (or the library would be unsound). This capability is useful to deduce non-aliasing properties between references and special memory locations managed by a library.
- `noWriteRef` declares that there exist no *aliasing* mutable references to the target instance or to other instances for which the target is a (sub)field[11]. Mutable references to the target that are obtainable from the source instance are still allowed. For example, this is the capability that a read-locked `&RwLock` type implies for its content: the API ensures that no mutable references to the content are given out while the lock is read-locked, and also ensures that `RwLock` remains borrowed as long as it is locked, so that there cannot be mutable references pointing to an instance that contains the `RwLock`. Like `noReadRef`, this capability is useful to deduce non-aliasing properties between references and special memory locations managed by a library.

[111]: Noble et al. (2022), *Rusty Links in Local Chains*

11: Instances behind a reference or raw pointer do not count as (sub)fields.

All these new capabilities can be used in the ⟨target_kind⟩ component of our capability annotations, but only `localRef` can be used in addition to `readRef` and `writeRef` in the ⟨source_kind⟩ component by using the special syntax `&loc self`. In the rest of this section, we present the properties and parameters of the new capabilities, following the same structure used in Sec. 4.3.1.

4.3.2.4 Implication Properties

As with core capabilities, there exist implications between the extended capabilities. These implications are represented in Fig. 4.11 and explained as follows:

- `writeRef` ⇒ `readRef`, which is the implication already presented in Sec. 4.3.1.4.
- `writeRef` ⇒ `write`, because any mutable reference can be used to perform modifications.
- `writeRef` ⇒ `unique`, because any mutable reference guarantees exclusive access to its target instance. For example, any local variable of type `&mut u32` guarantees that the referenced `u32` value cannot

be reached via other local variables. The Rust compiler implements one exception to this rule: if a type implements the special `!Unpin` trait, then its mutable references do not guarantee exclusive access and might even alias each other [112]. In our work, we implement the same exception, so that `writeRef` ⇒ `unique` holds only for the types that do not implement `!Unpin`. Reasoning about Rust programs rarely needs to use this exception, but we included it in our work for correctness.

[112]: Popov (2021), *Rust pull request #82834: Enable mutable noalias for LLVM >= 12*

- `unique` ⇒ `local`, because the target of a unique pointer is reachable only from the thread holding the pointer. As an example, any instance of `&mut u32` guarantees that the referenced `u32` value is thread-local, meaning that it cannot be modified by other threads.
- `localRef` ⇒ `readRef` ∧ `local`, because we defined `localRef` to be the conjunction of `readRef` and `local`.
- `writeRef` ⇒ `localRef`, because we defined that `writeRef` implies both `readRef` and `unique`, and `unique` in turns implies `local`. So, `writeRef` is stronger than `localRef`. Because of the `!Unpin` exception of `writeRef` ⇒ `unique` rule, also `writeRef` ⇒ `localRef` only holds for types that do not implement `!Unpin`.
- `readRef` ⇒ `immutable`, because all shared references guarantee immutability of their target.
- `unique` ⇒ `write`, because in our model, whenever a program holds a unique pointer, it is always possible (i.e., never UB) to modify the pointed-to instance. In other words, the model presented in this work does not have a capability for unique read-only pointers. As an example, consider the `&mut Cell` type. We previously described that its instances can be modeled as holding a `unique` capability for the content of the `Cell`. Any `&mut Cell` instance can also be used to modify the content of the `Cell`, via its `get_mut` method or by swapping the targets of `&mut Cell` instances. Since in practice we found that the `unique` capability is always accompanied by a `write` capability, we defined for convenience that the former capability implies the latter.
- `immutable` ⇒ `read`, `local` ⇒ `read`, and `write` ⇒ `read`, because in our model, whenever a program holds an immutable, local, or writable pointer, it is always possible (i.e., never UB) to read the pointed-to instance. In other words, the model presented in this work does not have a capability for unique, local, or writable pointers that cannot be dereferenced. These implications are defined only for convenience because we found that in practice when these capabilities are used the memory location can also be modeled with `read`.

There exist additional implications between the capability of an instance and the capability of its fields or dereference. Overall, all these implications are listed in Table 4.1. The additional cases compared to what we presented in Sec. 4.3.1.4 are:

- If `x` has a visible field `f`, `localRef(x)` ⇒ `localRef(x.f)` because `readRef(x)` ⇒ `readRef(x.f)` and because if an instance is local so are the fields that compose it.
- If `x: &mut T`, `localRef(x)` ⇒ `localRef(*x)` because

`readRef(x)` ⇒ `readRef(*x)` and because if a mutable reference is local, so is its target.
- If `x: &T`, `localRef(x)` ⇒ `readRef(*x)` because a shared reference type does not provide any locality guarantee.

4.3.2.5 Non-Aliasing Properties

With our extended capabilities, there are more combinations from which we can infer non-aliasing properties. Given two *different* roots r_1 and r_2, if the target type has non-zero-size we define the following incompatibilities between capabilities at the same program point. These are also represented by the red dashed edges in Fig. 4.11.

- `immutable(`r_1`)` ↮ `write(`r_2`)`, because the immutability property of `immutable(`r_1`)` would conflict with the modifications that `write(`r_2`)` enables through r_2. This rule generalizes the read-xor-write property of Rust, so that it holds not only between references, but also between implicit capabilities that might originate from library annotations.
- `unique(`r_1`)` ↮ `read(`r_2`)`, because `unique(`r_1`)` implies that there is no other root from which the target memory location is reachable, but `read(`r_2`)` contradicts that. This rule expresses the non-aliasing property of mutable references as an instance of the more general case of unique pointers. For example, given `x: &mut Cell<T>` and `y: &mut T` where `T` is a non-zero-size type then `y` cannot alias the content of the cell, because `x: &mut Cell<T>` implies a `unique` capability for its content while `y: &mut T` implies `read`.
- `noReadRef(`r_1`)` ↮ `readRef(`r_2`)`, because `readRef` implies that it is possible to obtain a shared reference to the target memory location, but `noReadRef` implies that there cannot be such references. For example, given `x: &Cell<i32>` and `y: &i32` then `y` cannot alias the content of the cell, because `x: &Cell<i32>` implies a `noReadRef` capability for its content while `y: &i32` implies `readRef`.
- `noWriteRef(`r_1`)` ↮ `writeRef(`r_2`)`, because `writeRef` implies that it is possible to obtain a mutable reference to the target memory location, but `noWriteRef` implies that there cannot be such references. For example, given `x: &Cell<i32>` and `y: &mut i32` then `y` cannot alias the content of the cell, because `x: &Cell<i32>` implies a `noWriteRef` capability for its content while `y: &mut i32` implies `writeRef`.

Note that contradictions can be derived for more pairs via implications, e.g., `readRef(`r_1`)` ↮ `writeRef(`r_2`)` because the first capability implies `immutable(`r_1`)` and the second implies `write(`r_2`)`, which are incompatible[12].

12: An alternative explanation is that the first capability implies `read(`r_1`)` and the second implies `unique(`r_2`)`, which are incompatible. Both explanations are valid; the non-aliasing between shared and mutable references has more than one reason to hold.

4.3.2.6 Immutability Properties

With the extended capabilities, there are two more rules, in addition to Sec. 4.3.1.4, to deduce immutability properties.

- Across any statement, an `immutable` capability ensures immutability of the target instance.

- Across any statement, a `unique` capability ensures immutability of the target instance. This is because any modifications done by the callee or by other threads would need to be done via an aliasing pointer that violates the uniqueness property of the capability.
- Across non-call statements S, the conjunction of a `local` and a `noWriteRef` capability for the same type instance x ensures immutability. This is because (1) `local` prevents any modification of x from other threads, and (2) `noWriteRef` ensures that the statement S does not modify x via a mutable reference. The latter holds because assignments to mutable references are the only way by which non-call statements can modify values in safe code.
- Across call statements S of functions that are marked by a purity annotation (presented in Sec. 4.3.3), the conjunction of a `local` and a `noWriteRef` capability for the same type instance x ensures immutability. This holds because pure calls are side-effect free, thus the only modifications can be (a) due to other threads, which is disallowed by the `local` capability, or (b) due to an assignment performed by S, which is disallowed by the `noWriteRef` capability.

With these rules, we can now show the step-by-step reasoning that makes it possible to verify the core of Fig. 4.8 and Fig. 4.10. In the program on the left of Fig. 4.8, it is necessary to deduce that the content of the cell cannot be modified during the assignment `*x = 100`. To do so, the first step is to notice `c` is alive (determined by a static analysis of the compiler) and unused (determined syntactically) across the assignment. This static information is encoded in the proof by assuming[13] a `writeRef(c)` capability. Even if the type of `c` is a shared reference, the capability it represents that `c` holds the capability to mutate the local variable, e.g., by replacing the shared reference with an assignment statement. Because of the implied capabilities of shared references, `writeRef` implies a `readRef` for the target of the reference: `readRef(*c)`. Due to the library annotations of `Cell`, `readRef(*c)` implies `local(c.as_ptr())` and `noWriteRef(c.as_ptr())`. By the immutability properties of `local` and `noWriteRef`, these capabilities are sufficient to deduce that the content of the cell does not change across the assignment. This was the core verification step, after which a verifier can deduce that the assertion in the program never fails.

13: In our proofs, we model the *trusted* type system information obtained from the compiler by assuming the capabilities associated with the root places.

In the program of Fig. 4.10, it is necessary to deduce that the content of the cell cannot be modified during the call to `unknown()`. Like in the previous example, the first step is to notice that `c` is alive and unused across the call, which is encoded in the proof by assuming a `writeRef(c)` capability[14]. Because of the implied capabilities of mutable references, `writeRef` propagates to the target of the reference as `writeRef(*c)`. Due to the library annotations of `Cell`, `writeRef(*c)` implies `unique(c.as_ptr())`, where `c.as_ptr()` describes the address of the content of the cell. Thus, by the immutability property of `unique`, the content of the cell does not change across the call. Using this, a verifier can prove that the assertion in the program never fails.

14: We will present in detail our first-order logic encoding and the formal proof roles that it enables in Sec. 4.5.

4.3.3 Purity Annotations

Many library APIs offer methods, which we call *pure*, that are deterministic, side-effect free, and always terminate. These properties make it

relatively easy to reason about their result: given two calls, it is sufficient to know that the arguments of the calls are equal to deduce that their results are equal as well. A standard example of a pure method is `Vec::len`, which returns the length of a vector by reading its value from a private field. Among libraries with interior mutability, `Cell::as_ptr` is a pure method that returns a raw pointer, whose target address is computed as a fixed offset from the address at which the `Cell` instance is stored. In the same API, the `Cell::get` method is pure as well, because it returns the value contained in the `Cell` instance by internally dereferencing the result of `Cell::as_ptr`. These three methods are all pure, but they rely on different kinds of input values. `Vec::len` depends on a field value, `Cell::as_ptr` depends on the target memory address of its reference-typed argument, and `Cell::get` depends on a value that is reached via a raw pointer. In program verification, the set of values on which a pure function can depend is called *footprint*. Defining the footprint of a pure function is a crucial task. The more precise the footprint is, the easier it is to reason about its result because fewer values might affect it. However, requiring a verification user to manually declare the precise footprint of each pure function might be detrimental to the usability of the verification technique. For example, declaring the precise footprint of `Vec::len` would force the user to expose implementation details of the library, breaking information hiding. In our verification approach, as a trade-off between precision and usability, we defined three classes of pure functions — `#[pure]`, `#[pure_memory]`, and `#[pure_unstable]` — corresponding to different footprint definitions, ordered from the more to the less restrictive.

The `#[pure]` annotation, also called *pure-value*, declares that a pure function depends only on the *values* reachable from its arguments. These values might be reached by following fields, dereferencing references, or calling other `#[pure]` functions. For example, this is the case of `Vec::len`, which depends only on the value of a private field. The *values* of fields of type raw pointer and type reference are defined differently: the value of a raw pointer is the address of its target, while the value of a reference is the value of its target. Accordingly, pure-value functions can depend on the address of the target of raw pointers but not on the value of its target, while pure-value functions can depend on the value of the target of references but not on the address of their target. Because of this restriction, `Cell::as_ptr` cannot be annotated as pure-value, because this function depends on the target *address* of a reference argument. As a special rule, pure-value functions cannot depend on the content of types with interior mutability (i.e., `UnsafeCell`). This follows the intuition that shared references guarantee immutability of their reachable target values. Because of this restriction, `Cell::get` cannot be annotated as pure-value: such function depends on a value that might be modified via usages of `&Cell`.

The `#[pure_memory]` annotation declares that a pure function depends only on the values reachable from its arguments, and their memory addresses. For example, this is the case of `Cell::as_ptr`, because such function depends on the target *address* of its only reference argument. The `Vec::len` method might also be annotated as pure-memory, but such annotation would be needlessly imprecise. As with pure-value functions, also pure-memory functions cannot depend on the content of

types with interior mutability (i.e., `UnsafeCell`), but they can depend on their memory address. Because of this restriction, `Cell::get` cannot be annotated as pure-memory.

The third purity annotation, `#[pure_unstable]`, is the most permissive. It declares that a pure function might depend on any memory value, including global variables or values at special memory addresses. The intuition is that these values are *unstable* and might change at any time, even though the execution of the pure function is deterministic. Because of this, when reasoning about a program it is typically difficult to ensure that the result of different pure-unstable calls is the same, so such functions should always be annotated with a postcondition that effectively restricts the footprint. One example of a pure-unstable function is `Cell::get`, as shown in Fig. 4.9. Its postcondition states that the result is equal to the dereference of `self.as_ptr()`, where the syntax `====` expresses a structural equality that considers both values and addresses (e.g., the target address of references), while `deref` is a built-in ghost pure-unstable function that models dereferences. The capabilities that the library declares for the memory location `self.as_ptr()` make it possible to reason about the results of multiple `Cell::get` calls. In contracts, which are considered to be evaluated atomically, the semantics of pure-unstable functions is stronger: within the same specification, calls of pure-unstable functions with the same arguments are guaranteed to evaluate to the same value. To make sure that the atomic semantics of contracts with *one* pure-unstable call is equivalent to the executable semantics of the same expression, we impose that the implementation of pure-unstable functions can call at most one pure-unstable function. However, in the presence of at least two pure-unstable function calls, the semantics of a specification differs from the executable semantics of the same expression: in a specification all these calls are evaluated in the same program state, while in executable code each of them is evaluated in a different program state. This design choice makes it possible to describe complex relations in memory, at the cost of losing the semantic equivalence.

Even though the evaluation of a specification uses atomic semantics, the evaluation of contracts in consecutive statements is *not* atomic, because we model the thread interference that might happen between them. For example, consider the `arc_client` function in Fig. 4.12. The postcondition of the first call `set_42` ensures that the `RefCell` contains an integer of value 42, while the precondition of the following function `require_100` requires such integer to be 100. Each contract is evaluated atomically, but in two different states, because in between the two calls other threads might interfere and modify all values that are not stable. In particular, the `i32` type contained in an instance of `&Arc<RefCell<i32>>` is not stable, because `Arc` instances have unique access to their content only as long as their reference counters are equal to 1. So, the expected verification result is to report an error stating that the precondition of `require_100` is not guaranteed to hold. Similarly, with the given signature and contracts it is not possible to implement `set_42` in a way that satisfies its postcondition, because immediately after returning from `set_42`, other threads might interfere and modify the content of the `RefCell`. To prevent this, the type argument should be `&mut Arc<..>` and the precondition should state at least that the

strong counter of the `Arc` is 1. This would model that the argument has unique access to the contained `RefCell<i32>` instance, which in turn would have unique access to the contained `i32` instance.

```rust
#[ensures(a.data() == 42)]
fn set_42(a: &Arc<RefCell<i32>>) { ... }

#[requires(a.data() == 100)]
fn require_100(a: &Arc<RefCell<i32>>) { ... }

fn arc_client(a: &Arc<RefCell<i32>>) {
    set_42(a);
    require_100(a);
}
```

Rust

Figure 4.12: Rust program with contracts referring to unstable values. The precondition of the `require_100(a)` call fails to verify.

Overall, these purity annotations can be ordered in a chain from the least restrictive (pure-unstable) to the most restrictive (pure-value), while at the same time weakening the guarantees about the result, because there are increasingly more values on which the result might depend on. By the definitions, each pure-value function is a valid pure-memory function, and each pure-memory function is a valid pure-unstable function.

The purity of the implementation of these functions is checked syntactically using the same rules mentioned in Sec. 3.6. Among them, the most important requirements are that pure functions must have copy-type parameters to prevent passing mutable references, cannot contain unsafe code to prevent implementing interior mutability, and can only call other pure functions to prevent, e.g., invoking libraries with interior mutability.

The footprint of the various purity kinds must be checked as well. For example, according to the footprint definitions, the result of a pure-memory function `f` with an argument `x: &i32` can depend on the target address of `x` but a pure-value function `g` with the same signature cannot. So, the former function can call the latter but the opposite direction should be disallowed, or `g` could violate its footprint by calling and returning `f(x)`. The general rule, checked statically, is that functions of a given purity kind can only call pure functions of the same kind. Since we defined all pure-value functions to be valid pure-memory functions, but not the other way around, only one of the two calls in our example is allowed. Similar footprint checks are performed on the statements of a pure function. So that, for example, the pure-memory function `f` in our example can be implemented to return the address of `x` by performing a cast (`x as *const i32`), while the same implementation is disallowed for pure-value functions.

Since we want our footprint checks to be performed statically and modularly, we introduced further restrictions to conservatively reject cases where the result of a pure function might be used as an argument in a pure function call with a larger footprint. Depending on the implementation of the functions, some of these cases might respect all footprint definitions, but detecting so would require an inter-procedural analysis that we want to avoid. Our restrictions, for example, prevent the result of pure-value functions from being used as an argument of pure-memory or pure-unstable functions. This rule can be seen as a second type-check pass in addition to Rust's: the arguments and result of pure-value functions

are what we call *value* snapshot types, while the arguments and result of pure-memory or pure-unstable functions are *memory* snapshot types. Intuitively, value snapshots represent the reachable values of an instance, while memory snapshots also represent the target memory addresses of references. Thus, a subtyping holds between the two: all memory snapshots are valid value snapshots, but the opposite direction does not hold for all types. Ideally, the user interface of a verification tool using this technique should represent in a different way expressions of type value or shared snapshot, to distinguish between them.

Many existing verification tools for Rust offer similar purity annotations: `#[pure]` in Prusti [29], `#[logic]` in Creusot [86], `specs` in Verus [62]. In all these cases, the semantics of these annotations correspond to the semantics of pure-value function annotations of our work. The definition of value snapshots is largely equivalent to the mathematical representation of type instances used in Creusot but, to the best of our knowledge, the definition of memory snapshots is novel and has not been used by any of the existing verifiers. Our approach provides the additional flexibility needed to reason about memory locations and interior mutability, as shown in the following examples.

[29]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*

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

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

4.3.4 Examples

By using the capability and purity annotations presented so far, it is now possible to explain the main proof steps needed to verify the examples in Fig. 4.2 (page 96), Fig. 4.3 (page 98), and Fig. 4.4 (page 99), which make use several types of the standard library. The full details of the encoding and memory model will be presented in Sec. 4.5, while the full annotation of the library types is available in the repository of our verifier [28].

[28]: (2024), *Repository of the Mendel verifier for safe Rust clients of interior mutability*. URL: https://github.com/viperproject/mendel-verifier

4.3.4.1 Usage of Option

The program on the left of Fig. 4.2 (page 96) requires the `Option::is_some` method to be marked as `#[pure]` to prove that its assertion always succeeds. Since `x` is a copy-type, across the call `use_option(x)` a capability `writeRef(x)` remains held by the client, implying `readRef(*x)` and then `immutable(*x)` (Sec. 4.3.2.4). This, in turn, implies that across the call the value of `*x` does not change (Sec. 4.3.2.6). Knowing that the arguments of the two calls of the pure function `is_some` are equal, we can then deduce that also the results are equal, which proves the assertion. This example does not contain any usage of libraries implemented with unsafe code, but it shows how the technique of this paper is generic enough to be used for reasoning about fully-safe Rust programs.

4.3.4.2 Usage of Cell

The program on the right of Fig. 4.2 (page 96) contains two assertions, only one of which should verify. Proving or disproving the assertions by using the library annotations provided in Fig. 4.9 requires first

reasoning about the value of `x.as_ptr()`, which is used in the contracts of the `Cell` library. Since the `Cell::as_ptr` method is marked as pure-memory, the call `x.as_ptr()` is known to depend only on the memory snapshot of the receiver `x`. This memory snapshot is constant throughout the execution of the `cell_client` function, because across each statement, a `writeRef(x)` capability remains held by the client, implying an `immutable(x)` capability that in turn guarantees immutability of `x`. Since across each statement `writeRef(x)` implies `readRef(*x)`, where `*x` is of type `Cell`, by the capability annotations of the library, we can then deduce `local(x.as_ptr())` and `noWriteRef(x.as_ptr())`. These two capabilities guarantee immutability of the target of `x.as_ptr()` across each statement of the function, except for `use_cell(x)` because it is a call of a non-pure function. This immutability property is sufficient to prove the last assertion in the program, as expected. Regarding the first assertion, checking `a == b`, our technique does not generate any deduction that proves it, so the assertion is reported as potentially failing. This is the correct outcome because to falsify `a == b` it would be enough to implement `use_cell` with the single statement `x.set(x.get() + 1)`.

4.3.4.3 Usage of RefCell

The program in Fig. 4.3 (page 98) contains two assertions, the first of which can be verified by deducing an immutability property for the content of the `RefCell` across the `use_refcell(x)` call, the second by proving a non-aliasing property between the data contained in `x` and `y`. To verify the absence of panics it would be necessary to also prove that the cell is not mutably borrowed when executing the `borrow` method, because in that case the library would panic. The `RefCell` type, in fact, contains two internal states mutable via shared references: one is used to store what is commonly called the content of the cell, while the second holds a reference counter, called borrow flag, that tracks how the content is borrowed. For simplicity, here we only present the main proof steps necessary to verify the two assertions, omitting the proof steps related to the borrow flag. A simplified version of the annotated library is in Fig. 4.13. Similarly to the annotations of `Cell`, the method `RefCell::as_ptr` is marked as pure-memory, thus the expression `x.as_ptr()` can be deduced to be constant throughout the execution of `refcell_client` because a `readRef(x)` capability is available across each statement, and `readRef` implies `immutable`. Initially, the content of the cell is copied into the variable `before`, because `*a` is desugared to `*a.deref()` and the contract of the `Ref::deref` function ensures that the result is a reference pointing to the memory location of `a.as_ptr()`, which in turn is known to be equal to `x.as_ref()` because of the postcondition of `try_borrow`. Across the initialization of `before` and all the following statements the place `a` remains *immutably* available (i. e., available, but immutably borrowed), implying for each statement a `readRef(a)` capability that, by the annotations of `Ref`, in turn implies `readRef(a.as_ptr())`. The immutability properties of the latter capability ensure that the content of the cell does not change across these statements, which in particular means that after `use_refcell(x)`, the value of `before` is still equal to the content of the cell. Next, the `*x.borrow()` expression returns a `Ref` instance whose `deref` method returns a reference to the content of the

```rust
#[capable(&self => readRef(self.as_ptr()))]
impl<'b, T> Ref<'b, T> {}

#[capable(&self => readRef(self.as_ptr()))]
#[capable(&mut self => writeRef(self.as_ptr()))]
impl<'b, T> RefMut<'b, T> {}

#[extern_spec]
impl<T> RefCell<T> {
    #[pure_memory]
    pub fn as_ptr(&self) -> *mut T;

    #[ensures(if let Ok(ref actual_ref) = result {
        actual_ref.as_ptr() == self.as_ptr() && ...
    } else { ... })]
    pub fn try_borrow<'b>(&'b self) -> Result<Ref<'b, T>, ...>;


    #[requires(...)]
    #[ensures(result.as_ptr() == self.as_ptr() && ...)]
    pub fn borrow<'b>(&'b self) -> Ref<'b, T>;
}

#[extern_spec]
impl<'b, T> Deref for Ref<'b, T> {
    #[pure_memory]
    #[ensures(result as *const _ == self.as_ptr())]
    fn deref<'a>(&'a self) -> &'a T;
}

#[extern_spec]
impl<'b, T> DerefMut for RefMut<'b, T> {
    #[ensures(result as *mut _ == self.data_ptr() && ...)]
    fn deref_mut<'a>(&'a mut self) -> &'a mut T;
}
```

Rust

Figure 4.13: A simplified portion of the library specification of the `RefCell` standard library module. Methods such as `borrow_flag_ptr` are not part of the API of `RefCell`, but clients of the library can model them by introducing new traits. The omitted code (...) corresponds to library properties that are not necessary to prove the examples in this thesis.

cell, which is then used to initialize the variable `after`. Since the value of `before` was already equal to the content of the cell, it thus follows that `before` = `after`, as required to verify the first assertion.

For the second assertion, it is enough to notice that `a` is available before the assertion, generating a `writeRef(a)` capability that implies `read(a.as_ref())` where `a.as_ref()` = `x.as_ref()`. Also `y` is available, generating a `writeRef(y)` that, by the annotations of `RefMut` and implication properties, leads to `unique(y.as_ref())`. Because of the non-aliasing property between the implied `unique` and `read` capabilities[15], it follows that `x.as_ref()` ≠ `y.as_ref()`, which, considering the postcondition of `RefMut::deref`, is then enough to prove that the second assertion always holds.

15: An alternative is to use the non-aliasing property between the implied `immutable` and `write` capabilities.

4.3.4.4 Usage of Arc

The program in Fig. 4.4 (page 99) requires reasoning about the values and properties of the reference counter of the `Arc` type, whose API is annotated in Fig. 4.14. Like with `RefCell`, the type definition of `Arc` actually contains (at least) two internal states mutable via shared references: one is used to store what is commonly called the content of the `Arc`,

while the second holds its reference counter, also called *strong* counter[16]. Since the API does not expose the memory location of this counter, the specification models it with a pure method `strong_count_ptr`, marked as ghost to make sure that it can be called only from specifications[17]. The key capability annotation of the library is the following, expressing that if the value of the reference counter is 1, then its value cannot be modified by other threads:

```
#[capable(&self if Arc::strong_count(self) == 1
    => local(Arc::strong_count_ptr(self))
)]
```
Rust

Thanks to this capability, in the first branch of `arc_client`, the reference counter is known to be local. This, combined with the `noWriteRef` capability of the `Arc` library, ensures that the counter remains exactly 1 for the whole first branch, making it possible to verify the first assertion. Knowing that the counter is 1, the second assertion can then be verified: before the assertion the root place `x` implies `writeRef(x.as_ptr())`, while the root place `y` implies `readRef(y.as_ptr())`. The non-aliasing property of the capabilities implied by `readRef` and `writeRef`, ensures that `x.as_ptr()` $\neq$ `y.as_ptr()`. The assertion in the `else` branch of the program checks a second time the exact same condition that was necessary to enter the branch. However, the evaluation of the condition of the `if` and the evaluation of the last assertion are done at different program points, and when the reference counter is not 1 nothing in the contracts of the library guarantees that its value remains the same. Indeed, other threads might drop all the existing clones of `x`, suddenly bringing the reference counter to 1 just before executing the last assertion. In the absence of any information about the counter, a verifier will then conservatively report a verification error, as expected.

16: For simplicity, in our work we assume that the *weak* counter is always zero, which can be verified by adding a `false` precondition to all the API functions that might create a weak reference (`Arc::downgrade`, `Arc::new_cyclic`). Reasoning about weak pointers in concurrent code is challenging because it is difficult to check atomically the state of *multiple* atomic counters, but we support that kind of reasoning in single-threaded libraries which are not affected by interference from other threads. For example, the `Rc` library that we use in our evaluation models both its weak and strong counters.

17: It does not need to be pure-memory, because moving an `Arc` instance does not move its content or reference counter, so it is sound to abstract from the concrete memory.

```
#[extern_spec]
#[capable(&self => readRef(Arc::as_ptr(self)))]
#[capable(&self => noReadRef(Arc::strong_count_ptr(self)))]
#[capable(&self => noWriteRef(Arc::strong_count_ptr(self)))]
#[capable(&self if Arc::strong_count(self) == 1 => local(Arc::strong_count_ptr(self)))]
#[capable(&mut self if Arc::strong_count(self) == 1 => writeRef(Arc::as_ptr(self)))]
#[capable(&mut self if Arc::strong_count(self) == 1 => unique(Arc::strong_count_ptr(self)))]
impl<T> Arc<T> {
    #[pure]
    fn as_ptr(this: &Self) -> *const T;

    #[pure_unstable]
    #[ensures((result == 1) == (deref(Arc::strong_count_ptr(this)) == 1))]
    fn strong_count(this: &Self) -> usize;

    #[pure] #[ghost_fn]
    fn strong_count_ptr(this: &Self) -> *mut usize;
}
```
Rust

Figure 4.14: A simplified portion of the specification of the `Arc` type of the standard library. The `strong_count_ptr` is not part of the API of `Arc`, but it can be introduced using new traits. Here we show it as part of `Arc` for simplicity.

Overall, in this section we presented how our capability annotations, in combination with several different kinds of purity annotations, make it possible to reason about clients of popular libraries with interior

mutability. In the next sections, we are going to motivate and justify the semantics of the core capabilities, then we later present how our verification technique can be encoded into first-order logic and checked by using an SMT solver.

4.4 Core Soundness

In this section, we present our novel proof technique, based on a semantics-preserving transformation (SPT) of the program, that makes it possible to derive the semantics of our *core* capabilities from simpler properties of Rust types.

4.4.1 Available Places and Root Places

Before presenting our proof technique, we first need to define our *available* and *root places* representation of the type system information, which are later used to define the semantics of our capabilities. In the context of this work, we define Rust *places* as expressions composed of a local variable followed by field accesses or downcasts of enumerations. Consider the example in Fig. 4.15, which we use to aid the presentation in this subsection. In the function `root_places_ex` below, `x` and `x.f` are places, but `&x.f` is not. The expression `x.g.0` would also be a valid place in the context of this function, even though the program does not make use of it.

```rust
struct T { f: i32, g: (i32, i32) }

fn root_places_ex(mut x: T, mut y: Mutex<i32>) {
    // ①
    let z = &x.f; // Ⓐ
    // ②
    drop(z); // Ⓑ
    // ③
}
```

Rust

Figure 4.15: Rust example used to present the *root* and the *available* places.

Given a compiling program written in safe Rust, for a fixed program point we call *available* the places that are usable by programmers. More precisely, these are the places that the compiler would allow to mutably or immutably borrow in a new statement injected at that particular location[18]. This determines two sets for each program point, which can be constructed using the compiler API: a set of places that are mutably available and a set of places that are immutably available, where the former is always a subset of the latter due to Rust language rules. In the example above, at ① the places `x`, `y`, `x.f`, `x.g`, `x.g.0`, and `x.g.1` are both mutably and immutably available. At ②, `x` and `x.f` are no longer mutably available because of the borrow `z`, but they are still immutably available. The mutably available places are then `y`, `x.g`, `x.g.0`, `x.g.1` and the new `z`, while the immutably available places are the same as at ① with the addition of `z`. At ③, after the expiration of the borrow `z`, the mutable and immutable places are the same as at ①.

18: We treat all local variables as being annotated with `mut`, which does not modify the semantics of Rust code because it is used only as a linter check in Rust.

The two-set representation of the available places is largely redundant. Knowing that `x` is mutably available already implies that the places `x.f`, `x.g`, `x.g.0` and `x.g.1` are mutably available too, because the fields `f` and `g`, as well as all the elements of the tuple at `g`, are visible in the program and can (one at a time) be mutably borrowed given a mutable borrow of `x`. Moreover, the two-set representation fails to capture which places can be simultaneously borrowed and which cannot. Just knowing that two elements a and b are in the set of mutably available places is not enough, because if $a =$ `x` and $b =$ `y` then it is permitted to mutably borrow both at the same time, but if $a =$ `x` and $b =$ `x.g` then after mutably borrowing the first place the second place would no longer be mutably available.

To solve these issues, we define a new single-set representation that we call the set of *root* places. Intuitively, root places are the available places that can be used independently from one another. Each root place, moreover, is mapped to a flag that represents whether the place is only immutably, or also mutably, available. The name root comes from the observation that the available places of a set can be seen as nodes in a forest of trees, where each edge represents a "is a subfield of" relation and where all the places in the same tree share the same prefix, which corresponds to the place represented by the root node of the tree. In the example above, at ① and ② the root places would then be just `x` and `y`, both mutably available. At ②, instead, the root places would be `x.f`, `x.g`, `y` and `z`, of which only `x.f` is marked as immutably available while all other places are marked as mutably available. Note that all subsets of the root places at the same program points can be borrowed at the same time, because no root place is a syntactic prefix of another root place.

Computing the set of root places is done in four steps:

1. **Remove redundancy in the mutably available places.** The set of mutably available places, obtained from the compiler API, is reduced by removing all places that are an extension of some other place in the set. This determines a set A.
2. **Remove overlap between immutably and mutably available places.** The set of immutably available places is reduced by removing all places that are in A, or a prefix of one of the places in A. This determines a set B.
3. **Remove redundancy in the remaining immutably available places.** The set B is further reduced by removing all its places that are an extension of some other place in the set. This determines a set C.
4. **Construction of the root places.** The set of root places is computed by taking the union of A and C, which are guaranteed by construction to have no elements in common. The places from C are the roots that are only immutably available, while the places from A are the roots that are also mutably available.

Applying these instructions step-by-step on the places available at ② in the example, we obtain:

1. Initially, the mutably available places are {`x.g`, `x.g.0`, `x.g.1`, `y`, `z`}. Removing `x.g.0` and `x.g.1`, which are extensions of `x.g`, we obtain A = {`x.g`, `y`, `z`}.

2. Initially, the immutably available places are {`x`, `x.f`, `x.g`, `x.g.0`, `x.g.1`, `y`, `z`}. Removing the elements of A and their extensions, we obtain B = {`x.f`}.
3. Since B only contains one place, no element is removed from it when computing C, which is then defined as {`x.f`}.
4. Overall, the set of root places is {`x.f`, `x.g`, `y`, `z`}, where `x.f` is immutably available and all the others are mutably available.

The main benefit of the set of root places representation is that a set can be seen as a separating conjunction of predicates in separation logic, where each mutably available place is interpreted as a predicate that guarantees unique access to the reachable memory locations, while each immutably available root is interpreted as a predicate that guarantees immutability of the reachable memory locations[19]. This representation has many similarities with the capability state defined in Sec. 3.7, although the construction technique is completely different.

19: The precise definition of the uniqueness or immutability properties is presented later and has some exceptions, due to special Rust language rules related to visibility, zero-size types and implementations of the `!Unpin` trait.

When a place is mutably available at a program point but is not used by the following statement, we say that the place is mutably available *across* the statement. For example, the places `x` and `x.f` are mutably available at ① but are not mutably available across Ⓐ, while `x.g`, `x.g.0`, `x.g.1` and `y` are. In this simple program, the places mutably available across Ⓑ happen to be equal to the places mutably available across Ⓐ.

Similarly, when a place is immutably available at a program point but is not used *for modifications* by the following statement, we say that the place is immutably available across the statement. For example, all places that are immutably available at ① are also immutably available across Ⓐ, because the only place used for modifications at Ⓐ is `z`. As before, in this simple program the places immutably available across Ⓑ happen to be equal to the places immutably available across Ⓐ, because `z` is consumed by the call.

The computation of root places available across a statement works exactly as described above, only starting from the sets of places available across a statement rather than available at a program point. Thus, across both Ⓐ and Ⓑ the mutably available roots are `x.g` and `y`, while the only immutably available root is `x.f`.

4.4.2 Proof by Semantics-Preserving Transformation

When reasoning about safe clients of libraries that are implemented with unsafe code, a recurring need is to prove what the unsafe implementation of the libraries cannot mutate. For example, consider the code on the left of Fig. 4.16, which uses a type with interior mutability `RefCell` and its associate types `Ref` and `RefMut`, which act as immutable and mutable pointers to a `RefCell`, respectively. The `try_borrow` call performs a runtime aliasing check, and then returns an instance `a` of type `Ref` only if there exist no aliasing `RefMut` instances. Inside the branch where `a` is alive, `x` is still usable and is passed to a function `unknown`, of which we ignore the implementation. The informal documentation of `RefCell` describes that a `Ref` instance guarantees immutability, which in our technique we model with a `readRef` capability. However, `unknown` might be implemented with unsafe code, and it is not clear which properties of the `RefMut` library still hold in that context. Our key

question to motivate the soundness of the semantics of `readRef` is the following. Assuming that the program executing `client` does not have UB and that all unsafe code is wrapped in a safe library, can we deduce that `unknown` does not mutate the content of the `RefCell`? Our answer is yes, and we deduce so with a novel proof technique that relies only on well-established properties of safe Rust – immutability of shared references to primitive types – and a basic property of library APIs that we later call *conversion methods*.

```rust
// Before:
fn client(x: &RefCell<i32>) {
    if let Ok(a) = x.try_borrow() {
        // a is available from here...
        unknown(x);
        // ...to here
    }
}
```

Rust

```rust
// After:
fn client(x: &RefCell<i32>) {
    if let Ok(a) = x.try_borrow() {
        let tmp: &i32 = a.deref();
        unknown(x);
        drop(tmp); // restores a
    }
}
```

Rust

Figure 4.16: Example of a refactoring that introduces a conversion method call, `a.deref()`, using a place that was available across the `unknown(x)` statement.

The main idea of the proof is that by leveraging the notion of available places, it is possible to rewrite the program into a semantically equivalent one, in which there are new local variables whose explicit capabilities reveal semantic properties that were implicit in the original program. We call this semantics-preserving transformation *SPT*. The code on the right of Fig. 4.16 shows the result of an SPT, which introduces a local variable `tmp` pointing to the content of `x`, and dropping `tmp` after `unknown(x)` so that it remains alive during the call. This transformation does not introduce compilation errors because (1) `a` is available before the `unknown` call, and (2) `a` is not being used in the `unknown` call. Moreover, it also does not modify the observable semantics of the program because (1) `deref` is what we later define to be a *conversion* method, roughly meaning that it has no side effects, and (2) the transformation does not introduce any UB because we assumed that all used Rust libraries are sound. Thus, the transformed program cannot have UB, because we assumed that also the original one has no UB. At this point, the important step is to notice that in safe Rust, it is UB to have a shared reference whose target value is modified while the reference is alive. Considering the shared reference `tmp`, and knowing that the transformed program has no UB, we deduce that the `unknown(x)` call cannot modify the content of `x`. This result holds in the transformed program, but since the `a.deref()` call that we introduced is side-effect free, the execution of `unknown` cannot detect whether the client called `a.deref()` or not[20] and the set of possible executions is the same. Thus, we can deduce that also in the original program the `unknown(x)` call cannot modify the content of `x`. Overall, we showed with an SPT that in both the original and transformed program the memory described by the `readRef` capability cannot be mutated by the call. This result holds based on the soundness of all used libraries, and on the knowledge that certain API methods such as the one used in `a.deref()` are side-effect free.

20: Some details of the runtime execution would differ, such as the number of elements in the call stack, or DWARF debugging information, but the soundness of Rust libraries should not depend on them.

In this example, the SPT that we used seems to have been constructed ad-hoc to make the example work. However, the reasoning can be generalized in a *proof by SPT* that can be applied to any program that

has a core capability available across a statement, determining in each of such cases that the associated memory location cannot be modified by the statement, even when the statement calls a library implemented with unsafe code. The key step of the generalization is to notice that a core capability available across a statement is sufficient to ensure that there exists an SPT that creates a local variable of type shared reference pointing to the memory location associated with the capability. Moreover, concretely performing the SPT in a tool is not necessary in order to derive the immutability property. Instead, it is sufficient to identify that the SPT is *possible*, as we will do later to motivate the semantics of the core capabilities[21].

21: The other capabilities do not coincide with the *explicit* capabilities of a Rust type, so their semantics cannot be derived using this proof technique.

One assumption of our proof technique is to know what are the *conversion methods* of a library API, which we define as the public safe methods whose only purpose is to convert between two types in a way that has no side effects. In particular, we consider consuming a non-copy type parameter to be a side effect, as well as non-terminating, triggering a panic, and using synchronization primitives. Intuitively, this means that usages of conversion methods cannot be detected at runtime by the library by means such as incrementing a counter each time the method is executed. Compared to the class of pure functions, these conversion methods can also accept non-copy type arguments such as mutable references. Examples of conversion methods in the standard library are the methods `get_mut` (implemented for `Mutex`, `RwLock`, `Cell`, `RefCell`, `OnceCell`, `UnsafeCell`, `SyncUnsafeCell`), `Deref::deref` (implemented for `MutexGuard`, `RwLockReadGuard`, `RwLockWriteGuard`, `Ref`, `RefMut`, `Box`), and `DerefMut::deref_mut` (implemented for `MutexGuard`, `RwLockWriteGuard`, `RefMut`, `Box`) [113]. Identifying the conversion methods in a library might be achieved with an interprocedural static analysis that inspects the implementation of such methods, but in this work we do not do so. Instead, we ask our users that for each core library capability annotation that they write, there should exist a corresponding conversion method that motivates the capability. Identifying conversion methods in an API can be done much more easily than proving the immutability of a memory location handled via raw pointers in unsafe code, because conversion methods are usually implemented with a few lines of code. For example, the `deref` method of `Ref` is implemented with `unsafe { self.value.as_ref() }`, where `as_ref` is a conversion method of the `NotNull` type that essentially performs a cast from a raw pointer to a shared reference.

[113]: Rust (2023), *The Rust Standard Library, version 1.69.0 (84c898d65 2023-04-16)*

One might wonder if the properties provable with our SPT technique are provable using a UB model such as Stacked Borrows or Tree Borrows [42, 43]. The answer is no, because such models define whether one program execution is UB or not, while in our technique we reason about UB of a *different* program, in which there are new local variables and method calls. Instead, we believe that our proof technique may be formalizable in RustBelt [101]. The difference is that we designed the assumptions of our proof to be easy enough to be checked manually (safe clients, conversion methods) or to coincide with Rust's library soundness assumptions. RustBelt's approach makes it possible to prove type soundness statement that we expect to be more expressive than our capabilities, but at the cost of requiring expertise and extensive manual effort regarding interactive theorem proving using the Iris framework [58]. Moreover, RustBelt's

[42]: Jung et al. (2020), *Stacked Borrows: An aliasing model for Rust*
[43]: Villani (2024), *Tree Borrows: A new aliasing model for Rust*

[101]: Jung et al. (2018), *RustBelt: Securing the foundations of the Rust programming language*

[58]: Jung et al. (2018), *Iris from the ground up: A modular foundation for higher-order concurrent separation logic*

formalization uses a *possible* definition of UB that does not capture the aliasing model of Stacked Borrows or Tree Borrows, while our proof by SPT is very conservative and (at the cost of incompleteness) makes only one assumption about UB: modifications to the primitive-typed target of active shared references are UB. As a consequence, our technique conservatively assumes that the private fields of a library type might be used to implement interior mutability even if the library type does not contain raw pointers or `UnsafeCell` types. By making as few assumptions about the definition of UB, our verification technique is sound for almost any formalization of UB that will be officially chosen by the Rust language team.

4.4.3 Semantics of Core Capabilities

Using the SPT technique, we can now give a more precise definition of the core capabilities, and prove the correctness of their properties. Other capabilities do not correspond to the *explicit* capabilities of a Rust type, so their semantics depend entirely on our definitions.

4.4.3.1 Definition

A `writeRef(x)` corresponds to the capability of obtaining a *mutable* reference with target address `x`, via an SPT (e.g., by introducing conversion method calls and borrows).

A `readRef(x)` corresponds to the capability of obtaining a *shared* reference with target address `x`, via an SPT.

Proving that a Rust library with an unsafe implementation provides a certain capability is out of scope for our work, because it requires verifying the unsafe-code implementation. Instead, in this section, we start from the two definitions above, corresponding to trusted library annotations, and we derive all properties presented in Sec. 4.3.1.

4.4.3.2 Implication Properties

To prove that the `writeRef` ⇒ `readRef` holds we need to prove that it is possible to obtain a shared reference (right-hand side), under the assumption that it is possible to obtain a mutable reference (left-hand side). This holds because in Rust any mutable reference `r` can be temporarily converted to a shared one using the expression `&*r`.

To prove the implications expressed by the first two rows of Table 4.1, we need to prove in a similar way that the reference represented by the target capability can be obtained from the reference corresponding to the source capability:

- `writeRef(x)` ⇒ `writeRef(x.f)`, where `x.f` is a visible field. In this case, we start by assuming that we can obtain a mutable reference `a` pointing to the memory address of `x` (left-hand side of the implication). Obtaining a mutable reference pointing to the address of `x.f` can always be done with `let b = &mut a.f`,

which proves the `writeRef(x.f)` in the right-hand side of the implication.

- `writeRef(x)` ⇒ `writeRef(*x)`, where `x: &mut T` for some type `T`. In this case, we start by assuming that we can obtain a mutable reference `a: &mut &mut T` pointing to the memory address of `x` (left-hand side of the implication). Obtaining a mutable reference pointing to the address of `*x` can always be done with `let b = &mut **a`, which proves the implication.
- `writeRef(x)` ⇒ `readRef(*x)`, where `x: &T`. This case is analogous to the second one, but using `let b = &**a`.
- `writeRef(x)` ⇒ `writeRef(x.f)`, where `x.f` is a visible field. This case is analogous to the first one, using `let b = &a.f`.
- `readRef(x)` ⇒ `readRef(*x)`, where `x: &mut T`. This case is analogous to the second one, but using `let b = &**a`.
- `readRef(x)` ⇒ `readRef(*x)`, where `x: &T`. This case is analogous to the second one, but using `let b = &**a`.

4.4.3.3 Non-Aliasing Properties

The only non-aliasing property that we defined for core capabilities is the following, in which all capabilities refer to the same program point.

$$\texttt{writeRef}(r_1, a_1) \land \texttt{readRef}(r_2, a_2) \land r_1 \neq r_2 \Rightarrow a_1 \neq a_2$$

To prove this implication, we start by assuming the left-hand side and we derive the right-hand one by using a proof-by contradiction that internally uses our SPT technique. From the capabilities on the left-hand side, we have that `x: &mut T` and `y: &T` can be obtained, and $r_1 \neq r_2$ guarantees that both `x` and `y` are obtainable in the same SPT and usable at the same time. We then prove the right-hand side $a_1 \neq a_2$ by contradiction, showing that $a_1 = a_2$ leads to an impossibility. This equality means that both `x` and `y` point to the same memory location. This is a seemingly impossible case in Rust, because subsequent code might perform modifications via `x` that would end up modifying the target value of the shared reference `y`. For example, in the case where `T` is the primitive type `u32`, it is possible to introduce an SPT like the following.

```rust
let tmp = *x;
*x = 0;
let y_val_0 = *y;
*x = 1;
let y_val_1 = *y;
*x = tmp;
```

Rust

This snippet guarantees that `y` will be observed when `x` holds different values. This conflicts with the immutability guarantee of `y`, by which `y_val_0` and `y_val_1` should be equal. Because of this contradiction, we conclude that the two references `x` and `y` must point to different memory locations, which coincides with the right-hand side of the implication that we wanted to prove.

However, there are certain exceptional cases where mutable references such as `x` and `y` might actually point to the same memory address in Rust. The official formal definition of UB is still under discussion, but to the best of our knowledge, the conjunction of the following conditions is sufficient to rule out all such exceptional cases:

(i) `T` is a type with a non-zero size. References of zero-size types can indeed point to the same memory address in Rust.
(ii) `T` does implement the `Unpin` trait (i.e., it does not implement `!Unpin`). Mutable references to `!Unpin` implementations do not imply unique access [112].

[112]: Popov (2021), *Rust pull request #82834: Enable mutable noalias for LLVM >= 12*

In our approach, we model these cases by weakening the implication or non-aliasing properties of our capabilities when one of the conditions above holds. For example, by defining that the incompatibility edges of Fig. 4.11 (page 110) only hold between non-zero-size types, or that the implication edges starting from `writeRef` skip `localRef` and `unique` for types that do not implement `!Unpin`.

4.4.3.4 Immutability Properties

The only immutability property that we defined for core capabilities is that a `readRef(x)` capability available across a statement ⟨S⟩ guarantees that `x`'s memory location, of type `T`, remains unchanged. Holding the `readRef` capability means that before the statement it is possible to obtain, using an SPT, a shared reference that remains alive at least until after the statement. For example, the reference `shared` in the following snippet where `T` is `u32`.

```
let shared: &u32 = ... // Shared reference pointing to x
⟨S⟩ // The original statement to reason about
drop(shared);
```
Rust

Since the Rust community agreement is that modifications to the target of an active shared reference should always be undefined behavior, being able to obtain the variable `shared` as shown by the snippet is sufficient to guarantee that the value of `x` cannot change across ⟨S⟩. In particular, if one considers the special case where ⟨S⟩ is an empty statement (e.g., a `{}` block in Rust), then the immutability result can actually be used to prove which values cannot be modified by concurrent threads. This motivates the soundness of our technique in the presence of multithreading: across any statement and between any two consecutive statements, our technique loses all information regarding memory values except for those for which it is possible to obtain a shared reference.

4.5 Encoding

In this section, we present the encoding of Rust programs and of the implicit capabilities to a first-order-logic subset of the Viper language [23]. This approach makes it possible for us to define our Rust verification technique independently from the lower-level techniques used for verification, such as symbolic execution or verification condition generation. At

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

a high level, our encoding uses a versioning technique to model the state of the memory at different program points in a control-flow graph. Initially, each version of the program memory is completely unconstrained. Then, while encoding each statement in the control-flow graph, the encoding progressively introduces constraints between memory versions. This expresses framing properties that hold across the execution of the statement, analogously to what in separation logic is done by the framing rule. Possible thread interferences are encoded as no-op Rust statements, across which only the values that cannot be modified by other threads are framed. At each program point, the typing information additionally generates non-aliasing constraints between instances stored with the same memory version, similar to what is expressed by separating conjunctions between type instances in separation logic.

4.5.1 Type Instances

We represent the memory (both heap and stack) of a Rust program with a family of *memory* total functions

$$\mathscr{M}_T : \mathscr{A}_T \times \mathscr{V} \to \mathscr{S}_T$$

each of which, indexed by a type T, maps an address $\mathscr{A}_T$ and a memory version $\mathscr{V}$ to an instance $\mathscr{S}_T$ modeled with a mathematical data type (defined later) that we call a *memory snapshot*. In the Viper language, these functions are defined using Viper *domains*. Since $\mathscr{M}_T$ is total, the instance of invalid addresses is defined, but their value is unconstrained and we rely on the Rust compiler to ensure that such invalid addresses are never used in the executable code of safe Rust programs. The choice of using a version parameter to model the values of a memory location at different points during the execution of a program was inspired by VCC [19]. Unlike VCC, which uses a version for each object instance, we use a global version to identify the state of the whole memory at a fixed point in time. Since a function execution and its specifications use only a subset of the memory, all the unused memory locations are left unconstrained without affecting the verification. Modeling the whole memory, moreover, might also make it easier to model in the specifications of a library special memory addresses corresponding to global variables or hardware access.

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

The memory snapshot of a Rust type instance is a mathematical representation of its values and memory locations that are reachable without following raw pointers or entering `UnsafeCell` types. In our encoding, we model the memory snapshot of a variable `x` with a memory address (representing the location of `x`) and an algebraic data type (representing its reachable values and memory locations) with the following recursive definition:

- The memory snapshot of a primitive type is its mathematical value.
- The memory snapshot of a raw pointer is the address of the target, represented as an integer.
- The memory snapshot of a reference is composed of both the address and the memory snapshot of its target.

- The memory snapshot of an `UnsafeCell` does *not* contain any representation of its content. It is as if the `UnsafeCell` were defined as an empty tuple.
- The memory snapshot of tuples is composed of the memory snapshot of its elements.
- The memory snapshot of structures is composed of the memory snapshot of its fields[22].
- The memory snapshot of an enumeration is composed of the discriminant represented as an integer and then, depending on the value of the discriminant, the memory snapshot of the fields of the corresponding variant.

22: In this model, the memory snapshot of a structure is isomorphic to the memory snapshot of a tuple whose elements have the same type of the fields of the structure.

Memory snapshots are convenient for modeling pure-memory Rust functions, which are encoded as a mathematical function whose arguments and return value are the memory snapshots of the corresponding Rust instances. To model pure-value functions, however, we rely on a weaker representation of Rust instances that abstracts over memory addresses. We do so by declaring a weaker kind of *value snapshots*, or $\hat{\mathcal{S}}_T$. Since memory snapshots are always at least as descriptive as value snapshots, it is always possible to convert instances of the former to the latter. The conversion between them is modeled by the following function:

$$\text{toValue}_T : \mathcal{S}_T \to \hat{\mathcal{S}}_T$$

This function is a bijection for types T that either do not contain references, or are defined such that all references are behind raw pointers or inside unsafe cells. For such cases the conversion is lossless, while for all other cases the conversion loses the address of reference targets.

Given a memory snapshot of a structure, tuple, or enumeration, the snapshot of a field is identified by the following function, which can be seen as a selector of the algebraic data type of memory snapshots. In the case of enumerations, this total function is defined only for the snapshots where the discriminant represents a variant that contains f, leaving other cases underspecified.

$$\text{fieldSnap}_f : \mathcal{S}_T \to \mathcal{S}_{T_f}$$

For each type T, we model the constant offset between the base address of an instance of T and a field f of type T_f with the following bijective function.

$$\text{fieldAddr}_f : \mathcal{A}_T \to \mathcal{A}_{T_f}$$

Given the address of an instance, there are now two ways to obtain the snapshot of one of its fields: by using fieldSnap (i.e., $\text{fieldSnap}_f \circ \mathcal{M}_T$) or by using fieldAddr (i.e., $\mathcal{M}_{T_f} \circ \text{fieldAddr}_f$). The consistency between the two is modeled with axioms, generated for each Rust type. For example, given an instance of `(i32, i32)` at address a, calling f an element of the tuple, such axiom states that the following consistency property holds for any memory version v:

$$\text{fieldSnap}_f(\mathcal{M}_{\texttt{(i32, i32)}}(a, v)) = \mathcal{M}_{\texttt{i32}}(\text{fieldAddr}_f(a), v)$$

For reference and raw pointer types T, the function to obtain the memory address of its dereference ($*T$) is targetAddr_T, while, only for reference

types, the function to obtain the memory snapshot of the dereference is targetSnap_T:

$$\text{targetAddr}_T : \mathcal{S}_T \to \mathcal{A}_{*T}$$
$$\text{targetSnap}_T : \mathcal{S}_T \to \mathcal{S}_{*T}$$

Note that the domain of targetAddr are memory snapshots, because the memory address of the target is defined as part of the memory snapshot, and because the memory location of the target of a reference is not a constant offset from the memory address at which the reference is stored. Like with fieldSnap_f and fieldAddr_f, the consistency between targetAddr_T and targetSnap_T is modeled with axioms. An example usage of snapshots is presented later when explaining the encoding of the Rust program in Fig. 4.17 to the Viper program in Fig. 4.18.

Instead of defining two snapshot kinds, an alternative approach would be to just use memory snapshots everywhere in the encoding. On the one hand, this would have two disadvantages. First, for all snapshots passed to pure-value functions, this approach would require constraining the target address of references to a fixed dummy value, by defining a function that lowers a snapshot into a snapshot with dummy addresses. Second, using the same snapshot kind everywhere would not benefit from Viper's type-checks, which help uncover encoding bugs when, e.g., the encoder incorrectly passes the result of a pure-value function to a pure-memory function. On the other hand, this approach might have the potential advantage of generating a shorter and more efficient encoding. Overall, for this project we preferred the safer route of defining two snapshot kinds, leaving the evaluation of the alternative single-snapshot approach to future work.

4.5.2 Capabilities

For each type, the approach described in Sec. 4.3 defines that capabilities at a given program point have two parameters: a root identifier and a memory address. Since all a verifier needs to know about a capability is whether it holds or not in a particular program point, we model them as uninterpreted boolean functions. Given a type T, we define the capability functions $\langle \text{kind} \rangle_T(r, a, w)$, where $\langle \text{kind} \rangle$ is one of the extended capability kinds (i.e., `readRef`, `writeRef`, etc.), $w \in \mathcal{V} \cup (\mathcal{V} \times \mathcal{V})$ is either a memory version that models the memory at a program point or a pair of memory versions that models a transition between two program points, $a \in \mathcal{A}_T$ is the memory address of the capability, and $r \in \mathbb{N}$ is the identifier of the root place at program point w from which the capability originates[23].

23: The domain of r is $\mathbb{N}$ because, at any program point, the root places are a finite set that can be ordered and enumerated, associating each root place with a numerical index. The numerical index of syntactically-equal root places can be different between different program points.

4.5.2.1 Implication Properties

For each Rust type T used in a program, the implication properties of its capabilities are modeled with axioms.

The implications represented by the arrow edges in Fig. 4.11 are modeled by axioms of the following shape, where $\langle \text{kind}_{\text{LHS}} \rangle$ and $\langle \text{kind}_{\text{RHS}} \rangle$ are

the source and target capability kinds of the edge:

$$\forall r \in \mathbb{N}, a \in \mathcal{A}_T, w \in \mathcal{V} \cup (\mathcal{V} \times \mathcal{V}) : \\ \langle \text{kind}_{\text{LHS}} \rangle_T(r, a, w) \Rightarrow \langle \text{kind}_{\text{RHS}} \rangle_T(r, a, w)$$

As an example, one of the implications for the `i32` type is the following, expressing that for all roots, memory addresses, and versions, a `writeRef` capability always implies `unique`:

$$\forall r, a, w : \texttt{writeRef}_{\texttt{i32}}(r, a, w) \Rightarrow \texttt{unique}_{\texttt{i32}}(r, a, w)$$

The implications represented by the cells of Table 4.1 are encoded by a slightly more generic axiom of the following shape, where $\langle \text{a}_{\text{RHS}} \rangle$ is the expression representing the address of a field instance or of the target of a reference:

$$\forall r, a, w : \langle \text{kind}_{\text{LHS}} \rangle_T(r, a, w) \Rightarrow \langle \text{kind}_{\text{RHS}} \rangle_T(r, \langle \text{a}_{\text{RHS}} \rangle, w)$$

As an example, one of the implications for the `&i32` type is the following, expressing that a `writeRef` capability on a shared reference implies a `readRef` capability for its target:

$$\forall r, a, w : \texttt{writeRef}_{\texttt{\&i32}}(r, a, w) \Rightarrow \texttt{readRef}_{\texttt{i32}}(r, \text{targetAddr}_{\texttt{\&i32}}(a), w)$$

4.5.2.2 Non-Aliasing Properties

Similarly to the implication properties, the non-aliasing represented by the red dashed edges in Fig. 4.11 are encoded as axioms of the following shape:

$$\forall r_1, r_2, a, w : \langle \text{kind}_{\text{LHS}} \rangle_T(r_1, a, v) \wedge \langle \text{kind}_{\text{RHS}} \rangle_T(r_2, a, v) \Rightarrow r_1 \neq r_2$$

For example, one of the non-aliasing properties of the `i32` type is the following, expressing that a `unique` capability cannot alias a `read` capability originating from a different root place:

$$\forall r_1, r_2, a, w : \texttt{unique}_{\texttt{i32}}(r_1, a, v) \wedge \texttt{read}_{\texttt{i32}}(r_2, a, v) \Rightarrow r_1 \neq r_2$$

4.5.2.3 Capability Annotations

The encoding of any capability annotation, conditional or not, is done by generating an axiom of the following shape, slightly more generic than those used to model the implication properties. In the axiom template, $\langle \text{cond} \rangle$ represents the runtime condition that guards the capability annotation ("true" if there is none), and $\langle \text{a}_{\text{RHS}} \rangle$ is the encoding of the Rust expression identifying the target memory location of the capability.

$$\forall r, a, w : \langle \text{kind}_{\text{LHS}} \rangle_T(r, a, w) \wedge \langle \text{cond} \rangle \Rightarrow \langle \text{kind}_{\text{RHS}} \rangle_T(r, \langle \text{a}_{\text{RHS}} \rangle, w)$$

To encode $\langle \text{cond} \rangle$ and $\langle \text{a}_{\text{RHS}} \rangle$ it is sometimes necessary to apply $\mathcal{M}_T$, for example when dereferencing raw pointers. However, the parameter v of $\mathcal{M}_T$ has type $\mathcal{V}$ while w has type $\mathcal{V} \cup (\mathcal{V} \times \mathcal{V})$. To express that

the evaluation should be done by using the memory version before the transition it is enough to define the following function

$$\begin{aligned} \text{start} \quad &: \quad \mathcal{V} \cup (\mathcal{V} \times \mathcal{V}) \to \mathcal{V} \\ \text{start}(v) \quad &= \quad \begin{cases} v & \text{if } v \in \mathcal{V} \\ v_1 & \text{if } v \in \mathcal{V} \times \mathcal{V}, \quad \text{where } v = (v_1, v_2) \end{cases} \end{aligned}$$

As an example, for the type `Cell<i32>` the first capability annotation of Fig. 4.9 generates the following axiom:

$$\forall r, a, w : \texttt{readRef}_{\texttt{Cell<i32>}}(r, a, w) \wedge \text{true} \Rightarrow \texttt{local}_{\texttt{i32}}(r, \langle \text{a}_{\text{RHS}} \rangle, w)$$

where in this case $\langle \text{a}_{\text{RHS}} \rangle$ is

$$\text{targetAddr}_{\texttt{*const i32}}(\texttt{Cell<i32>::as\_ptr}(\mathcal{M}_{\texttt{Cell<i32>}}(a, \text{start}(w))))$$

4.5.2.4 Pure Functions

Pure-memory functions such as `Cell<i32>::as_ptr` are encoded as uninterpreted functions that take memory snapshots as arguments, and return a memory snapshot. Pure-unstable functions take as argument both the memory snapshots of the Rust parameters and the memory version in which the function is evaluated, returning a memory snapshot. Regular pure functions, instead, take as arguments and return value snapshots. In Viper, the generated uninterpreted functions can be constrained by axioms, which we generate by modeling the body of the pure function as a pure Viper expressions, using the same functional encoding technique presented in Sec. 3.6.

4.5.3 Modeling of Non-Call Statements

When encoding a Rust function, the address of each local variable and argument is modeled as an unconstrained memory address. The body of the function is encoded starting from the middle intermediate representation (MIR) used by the Rust compiler: each statement is encoded independently, as well as each program point (before and after each statement). To model that any memory value might change when transitioning from a program point to the next, e. g., because of unsafe code, synchronized data races, or interior mutability, the encoding generates a fresh memory version to represent the state of the memory at each program point. The semantics is that the memory version represents any of the possible states of the memory at the modeled program point: each memory location is by default unconstrained, and is progressively constrained by encoding statements and capabilities.

Each statement is encoded by assuming a relation between applications of $\mathcal{M}$ that use the memory version before the statement and applications that use the memory version after the statement. For example, consider the program in Fig. 4.17, where the locations of the two local variables are modeled as `loc_x:` $\mathcal{A}_{\texttt{i32}}$, `loc_y:` $\mathcal{A}_{\texttt{\&i32}}$, the location of the parameter is modeled as `loc_unused:` $\mathcal{A}_{\texttt{i32}}$, and the three program points marked by comments are modeled with the memory versions `v0, v1, v2:` $\mathcal{V}$.

`prusti_assert` is a ghost assertion, which does not generate executable code but is verified to hold. The intermediate encoding of the body of this program is provided in Fig. 4.18, where `new_snap`$_{\texttt{i32}}$`(123)` is the uninterpreted function that yields the memory snapshot containing the value 123.

```rust
fn example(unused: i32) {
    /* v_0 */ let x: i32  = 123;
    /* v_1 */ let y: &i32 = &x;
    /* v_2 */ prusti_assert!(x == 123 && addr_of!(unused) != addr_of!(x));
}
```

Rust

Figure 4.17: Rust function used to demonstrate the encoding of non-call statements in Sec. 4.5.3.

```
// Encoding of 'let x: i32 = 123'
assume 𝓜_i32(loc_x, v_1) == new_snap_i32(123);

// Encoding of 'let y: &i32 = &x'
assume targetAddr_&i32(𝓜_&i32(loc_y, v_2)) == loc_x;
assume targetSnap_&i32(𝓜_&i32(loc_y, v_2)) == 𝓜_i32(loc_x, v_1);

// Encoding of 'prusti_assert!(x == 123 && addr_of!(unused) != addr_of!(x))'
assert 𝓜_i32(loc_x, v_2) == new_snap_i32(123) && loc_unused != loc_x;
```

Viper

Figure 4.18: Viper encoding, excluding capabilities, of the body of the function of Fig. 4.17. All variables modeling memory versions `v_`i, and address of local variables `loc_`j, are initially unconstrained.

This encoding is correct in the presence of concurrency, interior mutability, or, in general, libraries implemented with unsafe code, because in safe Rust code, non-call statements in different threads are guaranteed by the compiler to *commute*. That is, any of their thread interleavings will result in the same execution of the program. This is ensured by Rust because observable data races in safe Rust programs can be implemented only by using safe function calls that internally use unsafe code, otherwise they are undefined behavior. The model described above already takes care of removing any information regarding unstable locations, for which no relation is assumed across statements.

As next encoding step, it is necessary to model the capabilities generated by root places by assuming them at the program point where they hold. More precisely, for the program in Fig. 4.17 the generated capabilities should express that the place `unused` is available at every program point and across every statement, `x` is available at the program points `v_0` and `v_2` but only immutably available between `v_1` and `v_2`, while `y` is never available because its lifetime ends immediately after its initialization. These information are computed from the compiler API, using the algorithm presented in Sec. 4.4. The resulting encoding is in Fig. 4.19, where `root_0` and `root_1` are syntax sugar for 0 and 1, respectively.

The encoding presented so far is now sufficient to deduce immutability and non-aliasing properties. For example, at `v_1` the two `writeRef` capabilities imply that `loc_unused` $\neq$ `loc_x`, as a direct application of the axiom expressing the non-aliasing properties of capabilities. Across the assignment `let y: &i32 = &x`, instead, the value of both `x` and `unused` are known to not mutate, because of the `immutable` capability

```
// Encoding of program point v_0
assume writeRef_i32(root_0, loc_unused, v_0); // ‘unused‘ is available

// Encoding of ‘let x: i32 = 123‘
assume M_i32(loc_x, v_1) == new_snap_i32(123);
assume writeRef_i32(root_0, loc_unused, (v_0, v_1)); // ‘unused‘ is available

// Encoding of program point v_1
assume writeRef_i32(root_0, loc_unused, v_1); // ‘unused‘ is available
assume writeRef_i32(root_1, loc_x, v_1);      // ‘x‘ is available

// Encoding of ‘let y: &i32 = &x‘
assume targetAddr_&i32(M_&i32(loc_y, v_2)) == loc_x;
assume targetSnap_&i32(M_&i32(loc_y, v_2)) == M_i32(loc_x, v_1);
assume writeRef_i32(root_0, loc_unused, (v_1, v_2)); // ‘unused‘ is available
assume readRef_i32(root_1, loc_x, (v_1, v_2));       // ‘x‘ is immutably available

// Encoding of program point v_2
assume writeRef_i32(root_0, loc_unused, v_2); // ‘unused‘ is available
assume writeRef_i32(root_1, loc_x, v_2);      // ‘x‘ is available

// Encoding of ‘prusti_assert!(...)‘
assert M_i32(loc_x, v_2) == new_snap_i32(123) && loc_unused != loc_x;
```

Viper

Figure 4.19: Viper encoding, including capabilities, of the body of the function of Fig. 4.17. The encoding of capabilities introduces non-aliasing and immutability properties that are necessary to prove the last assertion.

that can be deduced for both local variables. Overall, these properties make it possible to verify the two `assert` statements in the generated code. Note that the encoding contains some redundancy, e.g., because the non-aliasing between `x` and `unused` can be deduced by both `v_1` and `v_2`, but no issue arises from that.

4.5.4 Calls and Semantics of Contracts

For modularity, each function call is modeled using only the contract of the callee, that is, preconditions and postconditions. The expressions written in a contract are evaluated as the body of pure-unstable functions, and are subject to the same restrictions. All expressions within the same precondition or postcondition are evaluated atomically, using the same memory version. For example, this means that multiple dereferences of the same raw pointer in the same expression will evaluate to the same value.

Our encoding models the interference of other threads by making sure that across any statement, any knowledge about memory locations that might be modified by other threads is lost. This way, their value is left unconstrained and the encoding soundly models any possible value that they might have. When encoding function calls and the postcondition check at the end of the function, the effect of other threads is modeled by introducing a *thread-interference* statement[24] with havocking semantics, which prevents the evaluation of contracts from incorrectly stating facts about unstable values. Crucially, the available capabilities at the program point of the thread-interference must also be encoded as being available *across* the statement that models it, or all knowledge about the stable memory values would be lost as well. This thread-interference step

24: For the capability reasoning (e. g., immutability rules), this statement can be considered equivalent to a no-op non-call Rust statement, such as `let _ = ()`.

is encoded just before the corresponding check: on the caller side for preconditions, and on the callee side for postconditions.

Overall, the encoding of a Rust call statement between two program points modeled by the memory versions `v_pre` and `v_post` is composed of the following steps, which internally make use of a fresh memory version `v_havocked` to encode the thread-interference before the precondition check:

1. Model the environment interference, by introducing a new memory version `v_havocked` and propagating to it only the memory values of `v_pre` that are known to be stable. As a result, all knowledge about unstable memory locations is havocked in the memory model corresponding to the new version.
2. Check of the precondition, evaluated using `v_havocked`.
3. Assumption of the postcondition, evaluated using `v_post`. Since `old(..)` expressions in a postcondition refer to the program state in which the precondition holds, such expressions are evaluated using `v_havocked`.

As part of this encoding, all root places that are available at `v_pre` will also be encoded as being available across the transition from `v_pre` to `v_havocked`, while the root places that are available across the call will be encoded as being available across the transition from `v_havocked` to `v_post`.

An example of the Viper encoding of the function `wrapper` of Fig. 4.20 is reported in Fig. 4.21. This function is annotated with a contract and its body only contains the call of a non-pure function, which is annotated with a contract as well. Note how all the memory versions in the encoding are initially unconstrained, and are gradually constrained and checked by assuming or asserting two-state expressions.

```rust
#[requires(deref(x.as_ptr()) >= 0)]
#[ensures(deref(x.as_ptr()) > old(deref(x.as_ptr())))]
fn increment_positive(x: &Cell<i32>) { /* ... */ }

#[requires(deref(x.as_ptr()) >= 123)]
#[ensures(deref(x.as_ptr()) > 123)]
fn wrapper(x: &Cell<i32>) {
    // v_0
    increment_positive(x);
    // v_1
}
```

Rust

Figure 4.20: A Rust program used to demonstrate the encoding of calls.

4.5.5 Branches and Loops

At a high level, branching statements in Rust are encoded as branching statements in Viper. The only relevant detail is the encoding of memory versions, because when entering one branch, the memory version should correspond to the version after the side effects of the evaluation of the branch condition. Similarly, inside each branch, the memory version at its end determines the memory version to be used after the join. An example of the Viper encoding of an `if` statement is provided in Fig. 4.22, where

```
method wrapper(loc_x: 𝒜_&Cell<i32>) {
    var v_0, v_0_havocked, v_1, v_1_havocked: 𝒱;

    // Assumption of the precondition ‘deref(x.as_ptr()) >= 123‘ at v_0
    assume let as_ptr == (Cell<i32>::as_ptr(ℳ_&Cell<i32>(loc_x, v_0))) in
        getValue_i32(ℳ_i32(as_ptr, v_0)) >= 123

    // Encoding of program point v_0
    assume writeRef_&Cell<i32>(root_0, loc_x, v_0); // ‘x‘ available

    // Encoding of call ‘increment_positive(x)‘ between v_0 and v_1
    {
        // Apply thread-interference to v_0
        assume writeRef_&Cell<i32>(root_0, loc_x, (v_0, v_0_havocked)); // ‘x‘ available

        // Encoding of program point v_0_havocked
        assume writeRef_&Cell<i32>(root_0, loc_x, v_0_havocked); // ‘x‘ available

        // Check of the precondition ‘deref(x.as_ptr()) >= 0‘ at v_0_havocked
        assume let as_ptr == (Cell<i32>::as_ptr(ℳ_&Cell<i32>(loc_x, v_0_havocked))) in
            getValue_i32(ℳ_i32(as_ptr, v_0_havocked)) >= 0

        // Assumption of the postcondition at v_1
        // ‘deref(x.as_ptr()) > old(deref(x.as_ptr()))‘
        assume let as_ptr == (Cell<i32>::as_ptr(ℳ_&Cell<i32>(loc_x, v_1))) in
            let old_as_ptr == (Cell<i32>::as_ptr(ℳ_&Cell<i32>(loc_x, v_0_havocked))) in
            let lhs == (getValue_i32(ℳ_i32(as_ptr, v_1))) in
            let rhs == (getValue_i32(ℳ_i32(old_as_ptr, v_0_havocked))) in
            lhs > rhs
        assume writeRef_&Cell<i32>(root_0, loc_x, (v_0_havocked, v_1)); // ‘x‘ available
    }

    // Encoding of program point v_1
    assume writeRef_i32(root_0, loc_x, v_2); // ‘x‘ available

    // Apply thread-interference to v_1
    assume writeRef_&Cell<i32>(root_0, loc_x, (v_1, v_1_havocked)); // ‘x‘ available

    // Encoding of program point v_1_havocked
    assume writeRef_&Cell<i32>(root_0, loc_x, v_1); // ‘x‘ available

    // Check of the postcondition ‘deref(result.as_ptr()) > 123‘ as v_1_havocked
    assert let x_as_ptr == (Cell<i32>::as_ptr(ℳ_&Cell<i32>(loc_x, v_1_havocked))) in
        getValue_i32(ℳ_i32(x_as_ptr, v_1_havocked)) > 123
}
```

Viper

Figure 4.21: The Viper encoding of the program in Fig. 4.20.

the memory version after the join `v_4` is initialized using the memory versions at the end of each branch, `v_2` and `v_3`.

In our technique, the encoding of loops is defined in two steps. First, a Rust loop annotated with an invariant is encoded as a non-deterministic `if` statement; a standard encoding technique designed such that the only branch of the `if` models and verifies an arbitrary loop iteration:

- The invariant is (1) checked before the `if` using the memory version of the program state that first enters the loop, (2) assumed inside a branch of the `if` using a fresh memory version that represents the program state at the beginning of any loop body execution, (3) checked at the end of the same branch using a memory version that represents the program state at the end of a loop iteration. Before any check of the invariant, all memory values are stabilized using the no-op statement encoding described before.
- The loop guard is modeled by assuming that it evaluates to true at the beginning of the branch of the `if`, while after the `if` it is modeled by assuming that it evaluates to false.
- An `assume false` statement terminates the branch of the `if`, in order to model that the verifier should not consider program traces that, from the branch of the `if`, join the encoding after the `if`.

```
fn branch() {
    ...
    // v_0
    if x {
        // v_1
        ⟨A⟩
        // v_2
    } else {
        // v_1
        ⟨B⟩
        // v_3
    }
    // v_4
    ...
}
```
Rust

```
method branch() {
    var v_0, ..., v_4: 𝒱;
    ...
    // Encoding of 'if x ...' starting from v_0
    var tmp: Bool := getValue_bool(ℳ_bool(loc_x, v_0));
    if (tmp) {
        ... // encoding of ⟨A⟩ starting from v_1
        v_4 := v_2;
    } else {
        ... // encoding of ⟨B⟩ starting from v_1
        v_4 := v_3;
    }
    ...
}
```
Viper

Figure 4.22: Example of the Viper encoding (right) of a branching statements in Rust (left).

The second step takes care of modeling all the information that should be propagated across the loop. This is done by first statically computing the root places that are statically known to be available across all statements of the loop[25]. Then, the capabilities generated by such root places are assumed to hold between the program state immediately before the loop and (i) the program state after the loop, as well as (ii) the program state at the beginning of an arbitrary loop iteration. This way, an arbitrary number of loop iterations is abstracted and modeled by capabilities that are known to be available across the memory versions representing the beginning and the end of the loop iterations. An example of the Viper encoding of a simple Rust loop is provided in Fig. 4.23. The most important thing to notice is that inside the `if`, the capabilities are assumed to hold between the memory versions `v_0_havocked` and `v_1`, modeling all past iterations before the one that is represented by the branch of the `if`, while after the `if` the capabilities are assumed using

25: Resolving trait calls is not necessary to compute available places. All that is needed are function signatures, and the compiler already uses them during type- and borrow-checking.

the memory versions `v_0_havocked` and `v_3`, modeling all iterations of a loop that terminated.

```rust
fn a_loop(z: bool) {
    let mut y: bool;
    let mut z: bool;
    ...
    // v_0
    #[invariant(y)]
    while x {
        // v_1
        x, y = foo();
        // v_2
    }
    // v_3
    ...
}
```

Rust

```
method a_loop(loc_z: 𝒜bool) {
    ...
    // Apply thread-interference to v_0, before the check
    ...
    // Check of 'y' at v_0_havocked
    assert getValuebool(ℳbool(loc_y, v_0_havocked));
    // Encoding of the loop body
    if * {
        // Assumption of the guard at v_1
        assume getValuebool(ℳbool(loc_x, v_1));
        // Capabilities available across the loop
        assume writeRefbool(0, loc_z, (v_0_havocked, v_1));
        // Assumption of 'y' at v_1
        assume getValuebool(ℳbool(loc_y, v_1));
        // Encoding of 'x, y = foo()'
        ...
        // Apply thread-interference to v_2, before the check
        ...
        // Check of 'y' at v_2_havocked
        assert getValuebool(ℳbool(loc_y, v_2_havocked));
        // Kill trace
        assume false;
    }
    // Assumption of the negated guard at v_3
    assume !getValuebool(ℳbool(loc_x, v_3));
    // Capabilities available across the loop
    assume writeRefbool(0, loc_z, (v_0_havocked, v_3));
    // Assumption of 'y' at v_3
    assume getValuebool(ℳbool(loc_y, v_3));
    ...
}
```

Viper

Figure 4.23: Example of the Viper encoding (right) of a while loop in Rust (left). `if *` is a non-deterministic choice. In the Rust program, the place `z` is the only one that is available across all statements of the loop.

4.6 Implementation and Evaluation

We implemented our verification technique in *Mendel*[26]: our new capability-based verification tool for Rust [28]. As an engineering choice, we developed Mendel by reusing various components from the codebase of Prusti: parsing and type-checking of specifications, retrieval of borrow-checker information, various intraprocedural dataflow analyses (e.g., definite initialization), bindings with the Viper verification framework, back-translation of verification errors to user-readable diagnostics on source code, caching, IDE integration and so on. We then used Mendel to evaluate our verification technique.

To the best of our knowledge, the technique that we presented handles any safe Rust code, except for the two-phase borrows feature of Rust, by which the properties of mutable references might hold starting from an *activation* program point that does not coincide with the *creation* of the mutable reference. To support this Rust feature, it might be enough to update the capability analysis so that the capabilities of mutable references are

26: Gregor Mendel, the father of modern genetics, opened the way to the study of *capabilities* and *mutations* of DNA.

[28]: (2024), *Repository of the Mendel verifier for safe Rust clients of interior mutability*. URL: https://github.com/viperproject/mendel-verifier

Table 4.2: Description of the modeled unstable memory locations and assumptions for each library type.

Library	Modeled unstable locations	Assumptions
`UnsafeCell`	Content	-
`Rc`	Content, strong and weak reference counters	
`Arc`	Content, strong reference counter	No weak references
`Cell`	Content	-
`RefCell`	Content, borrow flag	-
`AtomicI32`	Content	-
`Mutex`	Content, poison flag, lock flag	-
`RwLock`	Content, poison flag	-
`Box`	Target	-
`Mutex` with invariant	-	No poisoning
Verus-style cell	-	-
Verus-style pointer	Target	Always verified

generated starting from the activation rather than the creation of the reference. However, we have not explored that so far. Although our *technique* is general, due to time constraints, our *implementation* has some notable limitations: it does not support loops, quantifications, array and slice types, and integer type bounds. These features were not needed to test the examples in our evaluation, which use fairly simple features of Rust and whose verification challenges are largely orthogonal to the language limitations of the tool. More precisely, the complexity of verifying the programs in our evaluation lies in the interactions between types with interior mutability: aliasing, mutations via shared references and interference from other threads.

Our evaluation aims to show that our capability annotations are *useful* and, on the client side, *lightweight*. To do so, we first annotated the APIs of popular types with and without interior mutability of the standard library using capability annotations, contracts and helper ghost methods. As a second source of libraries, we ported to our specification language the core of three libraries related to interior mutability taken from the test suites of Creusot and Verus: a `Mutex` with a monitor invariant from the former; a cell-like and a pointer-like type from the latter. Overall, the annotated libraries are described in Table 4.2 and Table 4.3. Then, we built several safe clients for each of the annotated libraries, while including a large number of assertions to check that, in addition to panic freedom, the tool can verify the functional behavior of the API interactions. These clients are rather simple compared to real-world code, but the goal is to test short sequences of API calls. Finally, we ran our verification tool on the clients, measuring the verification time for each of them. The client programs and their verification times are described in Table 4.4. All the measurements were averaged over 10 runs (after a warm-up of the JVM) and were done on a laptop with a `i7-7700HQ` processor, 16 GB of RAM, and operating system Ubuntu 22.04.

In Table 4.2, we listed for each library the unstable memory locations that we modeled using our capability annotations. The simplest library types, such as `UnsafeCell`, or `Cell`, only contain one such memory location, while more complex types can have more of them. For example, we modeled three unstable memory locations in the `Rc` and `Mutex` libraries. When annotating the `Arc` library, we made one simplifying assumption.

Table 4.3: Description of the library annotations. "LOC" reports the number of lines of code needed to annotate the library (excluding empty lines, comments, imports and module declarations). "Functions" reports how many existing functions (including methods) were annotated as pure or not, and how many ghost functions were necessary to annotate the library. "Specifications" reports the number of library capability annotations, and the number of lines of code occupied by the contracts (i.e., pre- and postconditions, and purity attributes).

Type	LOC	Functions			Specifications	
		Impure	Pure	Ghost	Capabilities	Contracts
`UnsafeCell`	16	3	1	0	1	6
`Rc`	89	4	4	2	10	33
`Arc`	51	2	3	1	6	18
`Cell`	28	3	2	0	4	8
`RefCell`	127	6	1	10	8	52
`Ref`	34	1	1	2	3	9
`RefMut`	45	2	1	2	5	11
`AtomicI32`	23	2	0	1	3	5
`Mutex`	103	4	1	8	9	40
`MutexGuard`	48	3	0	2	6	12
`RwLock`	76	5	1	5	4	35
`RwLockReadGuard`	23	1	0	2	1	4
`RwLockWriteGuard`	37	2	0	2	4	7
`Box`	40	3	1	1	3	10
`Option`	33	3	2	0	0	22
`Result`	37	4	2	0	0	25
`ControlFlow`	9	0	2	0	0	4
`Mutex` with invariant	51	6	2	0	0	10
Verus-style cell	82	5	0	3	0	22
Verus-style pointer	74	4	1	3	2	16

The `Arc` type is implemented using two reference counters, one counting the `strong` references and the other counting the `weak` references. However, the library does not provide a way to check atomically the value of *both* the strong and weak counter, which makes it difficult to write a library capability annotation with an atomic condition that checks the value of both of them at the same time[27]. To work around this issue of the API, in our annotations we ignored the weak counter, assuming that it is always zero[28]. This challenge does not apply to single-threaded libraries such as `Rc`, for which we are able to annotate both its strong and weak reference counter. Regarding the types taken from the test suite of Creusot and Verus, we kept the existing assumptions: the type of a `Mutex` with invariant assumes that no mutex is ever poisoned – something that happens when a thread panics while holding a lock. This is not the case for our other annotations of the `Mutex` and `RwLock` types, for which we also model the panic flag. Regarding the Verus-style pointer type, its API is sound only under the assumption that its clients always respect the declared preconditions. This is a choice of Verus that brings greater flexibility in the design of libraries. We retained this choice to evaluate our tool on these advanced cases of specification.

In Table 4.3, we reported some statistics regarding our annotations of the libraries. For many library types, we were able to reuse some existing methods of the API, marking the methods as pure. For example, in the case of the `Rc` library we marked the `Rc::as_ptr` method as pure-value, `Deref::deref` pure-memory, `Rc::strong_count` and

27: This is also a challenge for Rust developers. In fact, the documentation of the `Arc::weak_count` method that returns the value of the weak counter states that "This method by itself is safe, but using it correctly requires extra care. Another thread can change the weak count at any time, including potentially between calling this method and acting on the result." [114]. The documentation does not suggest what the correct way to use this method is, and does not even state the memory ordering model that is used internally to perform the atomic updates on the counter. The API provides two more useful methods, `Arc::get_mut` and `Arc::make_mut`, which pose no particular issues for our verification tool.

28: This can be verified by adding a `false` precondition to all the methods of the API that might create a weak reference.

Table 4.4: Description of the verified clients, each of which tests properties of the library type reported in the "Used library type" column. "Lines of code" reports the total number of lines of code of the program (excluding empty lines, comments, empty `main` functions, imports and module declarations) and, among them, the lines of code used for contracts. "Assertions" reports the number of assertions to be verified in the program, classified by meaning: "Expected" are assertions that verify or that report a verification error as expected; "Incompl." are assertions for which the verifier reports an error due to an incompleteness. Each assertion occupies 1 line of code, counted as part of the "Total" column. "Time" reports the average verification time with standard deviations, out of 10 runs, using the Viper backend based on verification condition generation. The three groups of clients, from top to bottom, are hand-written clients making use of libraries with interior mutability, hand-written clients making use of libraries implemented using raw pointers and unsafe code, and clients adapted from the test suite of Creusot [86] or Verus [62].

Client	Used library type	Lines of code		Assertions		Time (s)
		Total	Contracts	Expected	Incompl.	
`arc.rs`	`Arc<i32>`	66	0	27	0	10.0 ± 0.7
`arc_rwlock.rs`	`Arc<RwLock<Vec<i32>>>`	97	6	29	2	34.6 ± 0.8
`atomic.rs`	`AtomicI32`	35	0	9	2	5.6 ± 0.1
`cell.rs`	`Cell<i32>`	102	5	30	0	8.8 ± 0.2
`mutex.rs`	`Mutex<i32>`	47	0	18	0	12.7 ± 0.4
`rc.rs`	`Rc<i32>`	102	0	53	0	15.8 ± 0.9
`refcell.rs`	`RefCell<i32>`	71	6	25	0	13.5 ± 0.8
`unsafecell.rs`	`UnsafeCell<i32>`	35	7	7	0	4.8 ± 0.2
`box.rs`	`Box<i32>`	10	0	4	0	5.5 ± 0.1
`mutex_inv.rs`	`Mutex<i32>` with invariant	11	0	1	0	4.9 ± 0.3
`verus_cell.rs`	Verus-style cell	9	0	4	0	6.2 ± 0.4
`verus_ptr.rs`	Verus-style pointer	34	7	10	0	10.3 ± 0.5

`Rc::weak_count` as pure-unstable. However, the existing methods were not always sufficient to specify the library. So, we introduced new ghost methods (declared in a new trait, and implemented for the library types) to model some type properties. For example, in the case of the `Mutex` library we added a `data_ptr` method that returns the address at which the content of the mutex is stored. We marked as ghost all methods that we added, even though in most cases it would be possible for the library to provide an implementation. Overall, most ghost methods were necessary to model some aspects of the interiorly mutable values of libraries. For example, by exposing their address, or by making it possible to refer to their value from specifications. Among the types with the highest number of ghost methods, `RefCell` uses them to expose the address and value of the contained data, and of the borrowing flag tracking the aliasing status of the type (non-borrowed, read-borrowed, or write-borrowed). In `Mutex`, we used the ghost methods to model the address and value of the protected data, and of other internal flags (locking and poisoning). In `RwLock` we did the same as in `Mutex`, but without modeling the locking flag. In the Verus-style libraries, some of the existing methods were already marked as ghost.

In Table 4.4, we reported some statistics regarding the clients in our evaluation, among which the used library type and the average time took by Mendel to verify them. Each function in the client programs makes a sequence of API calls, checking with a large number of assertions, or with a postcondition, that the verifier is able to prove the expected functional behavior of the library. These properties that are checked are mostly comparisons between primitive values, or between the target address of raw pointers. In a few clients, we also used preconditions to check only a particular scenario. For example, in the `refcell.rs` client, we required some functions to be such that their `RefCell` argument is not read-

nor write-borrowed. In two cases, we also hit an incompleteness of our technique, described at the end of this section. The verification time that we measured for these programs goes from 4.8 seconds for the simplest libraries, to 34.6 seconds for the more complex ones. In fact, the longest-taking client, `arc_rwlock.rs`, requires the verifier to reason about nested library types: a `Arc` containing a `RwLock`, containing a `Vec`. The second longest-taking client, `refcell.rs`, uses the `RefCell` library, which is one of those with the highest number of modeled unstable locations. These measurements were all made while using the Viper backend that internally translates Viper programs to Boogie. By profiling the verifier on a few cases, our preliminary results seem to indicate that most of the verification time is actually spent in the generation and parsing of Boogie code. This might be a result of our encoding technique that generates one axiom to model each capability property. Future work might confirm these profiling results and explore more efficient engineering solutions.

From the results of the evaluation, we can conclude that our specification technique effectively works on real-world libraries with interior mutability. It makes it possible to describe the properties decided by library developers and to verify usages of these libraries using an automated verifier. In particular:

1. The specification language is expressive, in that it made it possible to explicitly declare properties of all the types with interior mutability of the standard library that we considered.
2. The technique works well in the presence of nested types such as `Arc<RwLock<Vec<i32>>>`, for which the capabilities automatically propagate properties across the type boundaries thanks to the capability annotations and the implication properties that we defined.
3. The technique allows developers to reuse existing library methods in the specifications, for example by modeling with the `as_ptr` method the address of the content of the `Arc` library (Fig. 4.14). In fact, the annotations of libraries such as `UnsafeCell` and `Cell` required no new methods at all.
4. The technique makes it possible to verify clients of libraries with interior mutability, requiring in many cases little to no proof annotations on the client side.

Nevertheless, our verification technique has some limitations. Our technique relies on capabilities to deduce all framing and non-aliasing properties of a program. In some situations, the capabilities that we presented are not expressive enough to describe the content of some type with sufficient strength. In such cases, our technique has to conservatively consider the case that the content might be mutated by any function call, or aliased by any other type instance. As a result, our technique has incompleteness when reasoning about some framing or non-aliasing properties. This happened in the evaluation when using the `Arc` and `AtomicI32` types, but it might be possible to construct examples using other libraries. For the `Arc` type, we do not have a capability that precisely describes its content when (a) the strong reference counter is not 1, or (b) the strong reference counter is 1, the weak reference counter is 0, and the `Arc` is immutably shared. For the `AtomicI32` type, we do not have a capability that describes its content when the `AtomicI32` instance is borrowed by *local references*[29]. Future work might overcome this by introducing new

29: That is, a reference that has not been passed to any function call.

capabilities, or by combining our capability-based technique with other verification techniques for concurrent code.

4.7 Related Work

4.7.1 Rust Verifiers

In Ch. 3 we presented Prusti, a deductive verifier for Rust that leverages Rust's type properties to automatically build a memory-safety proof based on separation logic and — given user-written contracts — verify the functional correctness of the program. Prusti does not support reasoning about interior mutability because (like many other automated deductive verifiers) it lacks the necessary expressivity and completeness, even though its verification technique is sound in the presence of libraries implemented with unsafe code. In contrast, Mendel's technique supports reasoning about libraries with interior mutability, proving functional correctness of their clients. While Mendel's technique is primarily meant for reasoning about clients of interior mutability, it can also be used to reason about fully-safe Rust code, with some cases of incompleteness described at the end of our evaluation. Since both Mendel and Prusti are sound on their own, but incomplete in different cases, combining both approaches would make it possible to reduce the incompleteness to only the cases where both techniques are at the same time incomplete.

RustBelt [101] is a Coq formalization of Rust in which it is possible to model and verify the soundness of libraries. This work was later extended by RustHornBelt [102], adding support for verifying functional correctness. While both works are based on the Iris [58] framework and require manual proofs, our verification technique is automated and can be used by developers who do not have advanced knowledge of Coq or separation logic. The language of RustBelt and RustHornBelt is more expressive than the capability specifications of our work, but also more verbose. We believe that our capabilities are a useful user-readable abstraction that might be used to automatically generate parts of the Rust type definitions encoded in Coq. Proving the properties of our capabilities in Iris is an interesting research question for future work.

[101]: Jung et al. (2018), *RustBelt: Securing the foundations of the Rust programming language*

[102]: Matsushita et al. (2022), *RustHornBelt: A semantic foundation for functional verification of Rust programs with unsafe code*

[58]: Jung et al. (2018), *Iris from the ground up: A modular foundation for higher-order concurrent separation logic*

Creusot [86] is a deductive verifier for Rust that leverages Rust's type properties to verify functional properties. Creusot uses a technique based on *prophecies* to encode Rust programs into first-order logic, using the Why3 language [17]. The technique that it uses is not based on notions of capabilities and only supports reasoning about interior mutability by wrapping the types behind an API with a monitor invariant. Our technique makes it possible to reason more precisely about mutations to the content of types with interior mutability, without needing to change the signature of existing methods nor to wrap the types behind a new API. Moreover, Creusot's technique does not support reasoning about memory addresses, while our work has first-class support for them.

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

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

Verus [62] is another deductive verifier for Rust that, like Creusot, is based on a first-order logic encoding of Rust programs. One novelty of Verus is that it uses the linearity and borrow checks of Rust to let the user manage separation-logic permissions by using regular (ghost)

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

Rust variables. In our view, the capability system of Mendel can be seen as a generalization of the permissions tracked by Verus. For example, Verus' `PermData` type might be modeled in our technique as a structure providing a `unique` capability for the target of the associated `PPtr` type. The approach of Verus requires defining custom libraries in which the types representing permissions show up explicitly as arguments or return types, while our technique makes it possible to annotate existing libraries without modifying their method signatures; only adding new methods. As a result, our technique requires fewer annotations on the client side and can be applied to existing Rust code. Like Creusot, Verus has no support for explicitly reasoning about memory addresses, while our technique has first-class support for them.

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

Aeneas [103] translates a subset of Rust into a pure lambda calculus, which can then be verified using F*. Their technique explicitly does not support interior mutability nor unsafe code, but we believe that library annotations like ours might be used to detect usages of interior mutability that could in principle be translated into a pure functional language.

[115]: Gäher et al. (2024), *RefinedRust: A Type System for High-Assurance Verification of Rust Programs*

RefinedRust [115] is a Coq-based refinement types system for Rust that enables foundational semi-automated deductive verification of both safe and unsafe Rust code. RefinedRust has not yet been applied to verification of usages of interior mutability, but the language of its type invariants is based on RustBelt and is thus quite expressive. As such, it would be interesting to try to formalize the capabilities of our work in RefinedRust. We expect that this should be easier to do for our core capabilities because their semantics coincide with that of Rust references, while other capabilities such as `local` might require more work to formalize their concurrency-related aspects.

4.7.2 Verification of Other Languages

[116]: Sammler et al. (2021), *RefinedC: Automating the foundational verification of C code with refined ownership types*

RefinedC [116] verifies functional correctness of C code by using a type system with ownership and refinement types, carefully designed so that the Coq proof of memory safety and functional correctness is automated and syntax-directed. Compared to our specification language, their type system is more complex and does not have a notion of immutability.

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

[117]: Cohen et al. (2010), *Local Verification of Global Invariants in Concurrent Programs*

VCC [19] verifies low-level concurrent C code annotated with global invariants [117]. Their invariants typically require each shared object to keep track of its referencing objects using a set of back-pointers. This technique could be ported to Rust by modeling a ghost set of back-pointers for each object, but cyclic data structures are unidiomatic in Rust and manually updating the set of back-pointers is verbose. Our technique requires neither of the two. Still, our first-order logic encoding is inspired by their encoding to Boogie [16].

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

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

Pony [75] is a programming language that ensures data-race freedom of concurrent actor-based code by using a strong type system with capabilities, among which deny and unique properties. Part of our work on the capabilities of Rust libraries was inspired by the rich expressivity of Pony types.

4.8 Conclusions

We have presented a new technique to specify the capabilities of Rust libraries implemented with unsafe code, and to verify the functional correctness of some of their safe clients. To ground the semantics of the capabilities associated with Rust types, we have presented a proof technique, based on a semantics-preserving transformation, that enables reasoning about safe clients without requiring knowledge of the semantics of unsafe code. Our approach enables developers to explicitly declare the intended aliasing and mutability properties of *existing* libraries, making it easier for both automated verifiers and other human developers to reason about their usages. An open-source implementation of our verification technique and our evaluation data is available on GitHub [28].

[28]: (2024), *Repository of the Mendel verifier for safe Rust clients of interior mutability*. URL: https://github.com/viperproject/mendel-verifier

5 Tool Implementation

We implemented both of our verification tools, Prusti (cf. Ch. 3) and Mendel (cf. Ch. 4), following the Rust verification principles outlined in Sec. 2.3. These tools largely share the same codebase. In this chapter, which is partially based on our NFM 2022 paper 'The Prusti Project: Formal Verification for Rust' [31], we describe the common architecture of our tools and highlight some usability solutions that we adopted to ensure a good user experience.

[31]: Astrauskas et al. (2022), *The Prusti Project: Formal Verification for Rust*

Collaborations Overall, the development of the first version of Prusti was done in collaboration with fellow doctoral student Vytautas Astrauskas. While Vytautas and I contributed equally to most aspects of the project, I led the implementation of the `cargo` integration (Sec. 5.1.1), verification server (Sec. 5.1.1), testing infrastructure, source-level error reporting (Sec. 5.2.1), error handling (Sec. 5.2.2) and IDE integration (Sec. 5.2.3). Later versions of Prusti were developed with additional collaboration from fellow doctoral students Aurel Bílý and Jonáš Fiala. Among the notable improvements, Aurel and Jonáš reimplemented the specification embedding (Sec. 5.1.2) using procedural macros, and implemented the retrieving of specification from external crates (Sec. 5.1.3.2).

5.1 Architecture

Our verification tools target real-world code written in Rust, which is itself a mature and complex language. Accordingly, our tools are designed to reuse existing functionality from the Rust compiler whenever possible, in order to reduce the implementation burden and faithfully maintain compatibility with the constantly-evolving Rust ecosystem.

5.1.1 Design Overview

Our tools are implemented as *compiler drivers* [118], using the library interface of the standard `rustc` compiler extensively; the overall workflow is presented in Fig. 5.1. When verifying a crate, our verifiers start by launching a `cargo` instance ①, which in turn launches in the correct order several instances of `rustc` to verify the dependencies ② and the local crate to be verified ③. Our verifiers then interact with each `rustc` process, using its program representations, analysis results, and error reporting utilities. To have verification-specific program annotations type-checked analogously to regular Rust expressions (including error-reporting), our tools perform a *specification embedding* pre-processing, converting annotations into Rust code so that any type error in the annotations is caught when type-checking the generated code ④ (cf. Sec. 5.1.2). As a result of this step, `rustc` holds a mid-level representation (MIR) of both the program and its (embedded) specifications. Working on the MIR, our tools can then perform their own analyses (in ⑤), and generate a Viper program that is sent to a verification server, which wraps an

[118]: (2024), *Rust Compiler Development Guide: rustc_driver and rustc_interface*

instance of the Viper verifier. If verification fails, our tools translate the Viper errors back to user-readable Rust errors reported via the compiler API (cf. Sec. 5.2.1). If, instead, verification succeeds, our tools persist the public specifications on disk, so that they can be retrieved when verifying dependent crates.

The compiler driver architecture is used by popular tools such as Clippy [67] and Miri [44]; it has two main advantages. First, it raises confidence that the semantics used by our tools is faithful. In fact, our verifiers directly obtain a control-flow graph (CFG) representation of any parsed Rust function from the compiler, instead of inventing a new representation, which could lead to errors or semantic differences over time. The CFG-based representation used in our work, called *unoptimized MIR*, has a simple order-of-execution semantics and a limited number of statements; at this stage, many of the more-subtle aspects of Rust's evaluation semantics have been *already handled* by the compiler. For example, our tools do not need to be aware that Rust uses short-circuiting semantics for Boolean operators, because Boolean expressions are already transformed by the compiler into multiple statements evaluating individual operators. Unoptimized MIR maintains all type-checker information, along with back-links that allow the compiler (and thus also our tools) to translate error messages back to the source code.

[67]: (2024), *Clippy: A collection of lints to catch common mistakes and improve your Rust code*
[44]: (2024), *Miri: An interpreter for Rust's mid-level intermediate representation*

Second, the above architecture enables our verifiers to reuse compiler components. Besides building upon unoptimized MIR, our tools reuse the compiler's type and borrow checker to ensure that user-written annotations follow typing rules analogous to regular Rust expressions, as explained in Sec. 5.1.2. Similarly, our tools reuse the Rust compiler's error reporting component to display verification errors. This way, the default syntax of the reports is familiar to Rust programmers and the compiler can be configured to report machine-readable errors. The latter simplifies integrating our verifiers with other tools. For example, IDE extensions like our Prusti Assistant extension for Visual Studio Code [119], but even verification-unaware tools such as Rust-analyzer [120], can be configured to report verification errors generated by running, e.g., `cargo prusti` instead of `cargo check`.

[119]: (2024), *Repository of Prusti Assistant, the IDE extension of Prusti.* URL: `https://github.com/viperproject/prusti-assistant`
[120]: (2024), *rust-analyzer: A Rust compiler front-end for IDEs*

5.1.2 Specification Embedding

Verification-specific program annotations (e.g., method contracts) are implemented with *procedural macros* [121]. These macros are defined to generate nothing when compiled using the regular Rust compiler. However, when compiled with one of our verifiers, a *specification embedding* is performed: to make the compiler both type-check and translate (to MIR) these specifications, corresponding methods are added to the program. For some specification constructs, such as logical implications and quantifiers, the specification embedding is more involved, replacing them with usages of Rust features that have the right type-checking requirements. For example, logical quantifiers are embedded as *closures*.

[121]: (2024), *Procedural Macros*

Our verifiers use a Pratt parser [122] to perform the embedding of specification constructs, before invoking the `syn` [123] Rust parser on the result, yielding an AST representation. The resulting specification expressions are embedded into the bodies of new methods with unique names. Then,

[122]: Pratt (1973), *Top Down Operator Precedence*
[123]: (2024), *The syn crate*

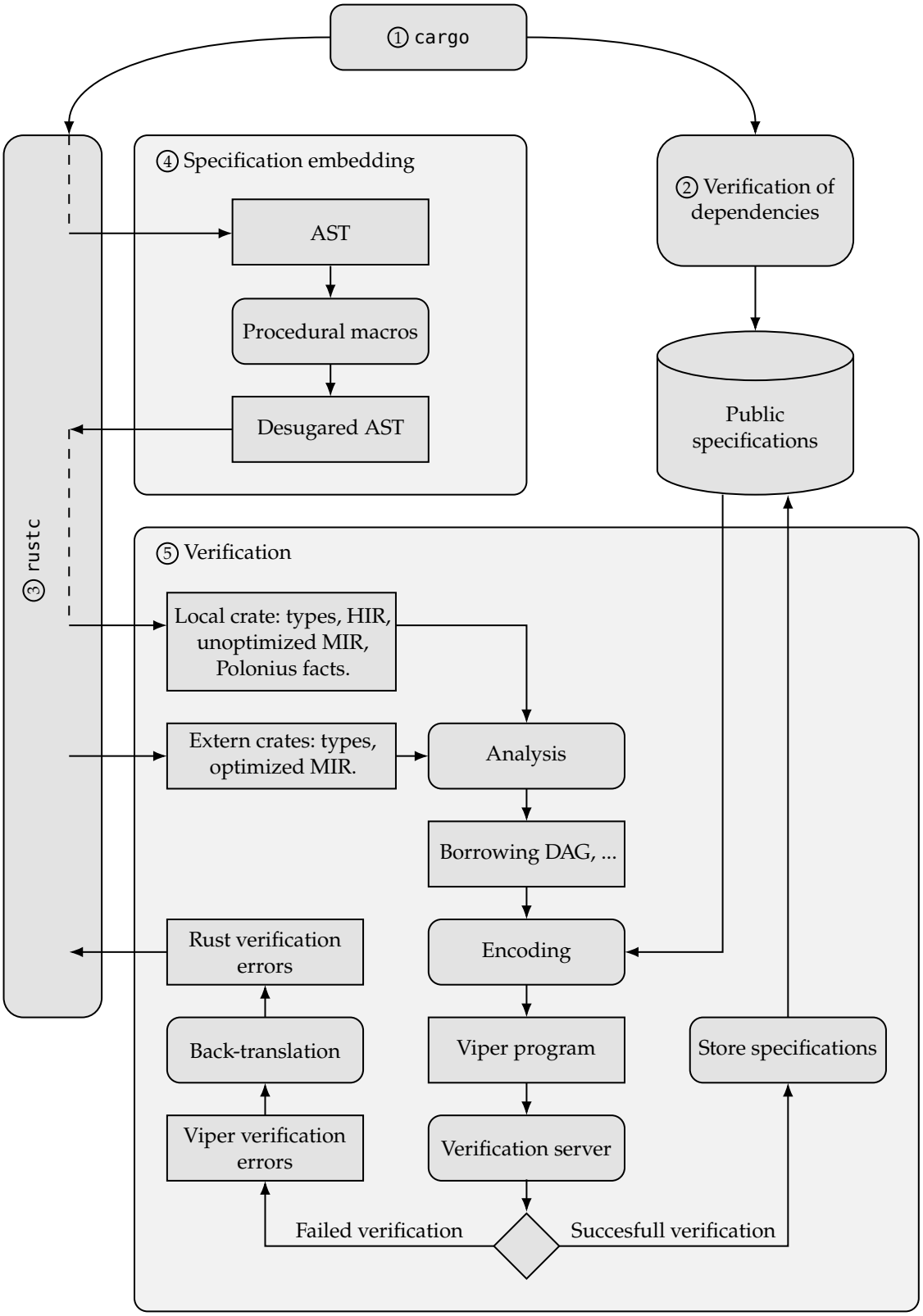


Figure 5.1: Overview of the encoding process of our verification tools. Rounded rectangles represent actions, while non-rounded rectangles represent data structures. The verification of dependencies, internally, applies the same steps ③, ④, and ⑤ on each crate. The public specifications of each crate are persisted on disk and are retrieved when verifying the dependent crates.

our tools construct a mapping between these generated methods (called *specification items*) and the relevant construct in the original source code (e.g., for a precondition, the method it is a precondition of). By feeding the program augmented with specification items through the compiler, we both check that the specifications type-check and can obtain corresponding MIR representations of the specifications. The type-checking and evaluation semantics reflected by this translation to MIR are those of standard `rustc`; this approach reuses the standard semantics of the Rust language for specification checking and compilation.

5.1.3 Compiler Interface

Out tools obtain various information from `rustc`'s data structures, as illustrated in the edges from ③ to ④ in Fig. 5.1. Given how Rust compilation works, different information is available (and used by our tools) for the *local* crate (i.e., the crate being compiled/verified) and *external* crates (the dependencies of the local crate).

5.1.3.1 Local Crate

For the local crate, our tools obtain a high-level AST representation (HIR), the type definitions, the unoptimized CFGs of the functions (MIR), and borrow-checker information (Polonius facts), defining the compiler-determined lifetimes of references. In particular, our tools use HIR, in which function names have already been associated with their definition, to retrieve specifications embedded in specification items, as described in Sec. 5.1.2. Type definitions are used to generate Viper predicate or domain definitions that model those types in the program proof, while unoptimized MIR is used to generate the corresponding Viper code itself.

The compiler offers various versions of MIR at different stages during the compilation process. Our tools use the *unoptimized* version because it is the only one on which the borrow checker runs. This also has a semantic advantage, since we do not need to worry whether compiler optimizations preserve the strong type properties that Prusti exploits[1]. Our tools use the results from the Polonius borrow-checker, also called *facts*, in the static analysis of capabilities that automates the generation of annotations such as folding and unfolding of Viper predicates (cf. Sec. 3.7), or in the static analysis that computes which Rust places are fully initialized and non-borrowed (cf. Sec. 4.4.1).

1: See for example [124] for an optimization that used to copy non-duplicable mutable references.

Previously, the compiler API did not expose Polonius facts, but the compiler developers were very supportive in accepting our proposed additions to the API [125]. Our changes have since been used by at least one other static analysis tool, Flowistry [89], to access precise aliasing information.

[125]: Astrauskas (2021), *Rust pull request #86977: Enable compiler consumers to obtain mir::Body with Polonius facts*

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

5.1.3.2 External Crates

For external crates, the compiler offers strictly less information than for the local one, primarily for performance reasons. Type definitions and

optimized MIR are available (our tools use the former to encode calls), but the HIR, the unoptimized MIR, and the Polonius facts are not present. Since the overall methodology of our verifiers is modular, the only real limitation this imposes is that any verification specifications written *in* an external crate cannot be seen via the compiler interface. To work around this, following the example of the MIRAI static analyzer [126], our verifiers store the public specifications of successfully verified crates on disk, so that when verifying dependent crates these specifications can be retrieved automatically. Additionally, in case the source code of the external crates cannot easily be modified by the user (e. g., Rust's standard library), our tools support declaring trusted external specifications in the local crate.

[126]: Facebook (2024), *MIRAI: An abstract interpreter for the Rust compiler's mid-level intermediate representation*

5.2 Usability Solutions

In the design of our verification tool, we strived to make the user experience as smooth as possible, so that users can focus on the important correctness properties of their program instead of, e. g., jumping between several different program representations or different tools. In this section, we present some of the usability solutions that we implemented.

5.2.1 Source-Level Error Reporting

A core usability feature of our verification tools is the error reporting on the Rust source code, which is enabled by a *back-translation* of verification errors from Viper into Rust. At a high level, the result of this translation has to be informative enough for the user to fix a verification error, but without mentioning any implementation detail of the verifier. This conversion is handled by an error manager that, in its simplest form, generates a source-code span and an error message. The overall back-translation technique is similar to what already used in other verifiers based on Viper, such as Nagini [25].

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

To generate the source-code span, the error manager registers during the encoding process which source-code span was responsible for the generation of which Viper statement and expression. More precisely, the encoding process associates a unique *position ID* with each AST node of the generated Viper program, and the error manager keeps a map from position IDs to source-code spans, where the source-code span is determined by querying the Rust compiler for the span associated with the MIR statement that the verifier was encoding to Viper.

To generate a user-readable message, the error manager additionally registers during the encoding contextual error information for each item of the Viper program that might fail with a verification error. For example, Viper's `assert` statements. Since the encoding is such that each Viper AST node is associated with a unique position ID, the error manager only needs to map each position ID of a possibly-failing node to its contextual information. This information consists of a verification-error kind and, in some cases where the kind alone is insufficient, a string with some additional details. The error kind serves to distinguish, for example, a failing Viper `assert` statement that encodes a Rust `assert!` statement

from an identical Viper statement that, instead, encodes the check of the precondition of a Rust function call.

Having registered all this information, to perform the back-translation, the error manager only needs to read from a Viper verification error the position ID of the AST node responsible for the error, using that ID to retrieve a source-code span and an error context. The error context is then converted to a Rust-level verification error message that is reported on the associated source-code span. The diagram in Fig. 5.2 represents the encoding of a Rust program into Viper (going right) and the back-translation of a Viper error into a source-code span. If the Viper verification error reports, in addition to the failing statement, the precise subexpression that caused the failure (e. g., an unsatisfiable conjunct in a `assert` statement), then its position is back-translated using the same technique to the source-code level and is reported to the user.

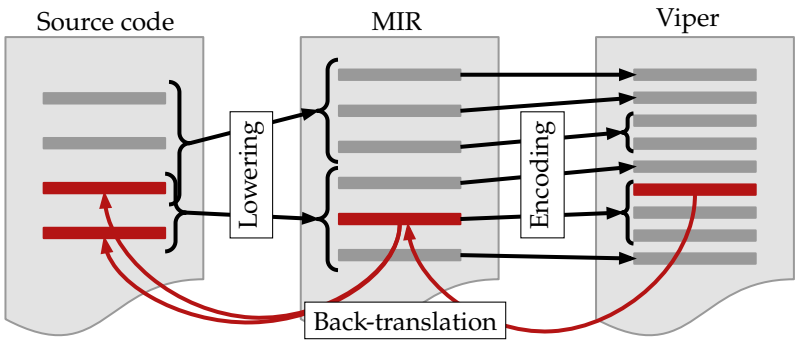


Figure 5.2: Overview of the encoding (going right) from Rust, through MIR, into Viper. The red lines and blocks (going left) show the back-translation of a Viper verification error into the source-code span that generated the failing Viper statement. Every horizontal rectangle represents the position of a statement on the Rust, MIR, or Viper level.

5.2.2 Error Handling

A verification tool for Rust has to handle several different kinds of errors, and reporting them in a clear way enables a better user experience. For example, the verification might fail because the implementation contains some bugs or disagrees with the specification. Otherwise, when writing program annotations the users might make mistakes, such as calling a non-pure function from a contract, that have to be reported in a special way to avoid confusing them with, e. g., functional-correctness verification errors. Moreover, the tool might also encounter internal errors, such as broken invariants, that have to be reported to the user. Terminating the tool is one option, but not the best one.

In our tools, we classified these errors into three kinds, depending on *what* is to blame for the error.

- **Verification errors** report that the tool reached the point of checking the correctness of the program, but that step failed. This signals that there is a bug in the implementation, that the specification expresses the wrong property, or that the property to be checked is too complex for the underlying SMT solver. In this case, the solution is to change the implementation or specifications[2].
- **Specification errors** report that a user-written specification is invalid. For example, a contract is trying to call a non-pure function. In this case, the blame is on the verification user who wrote it and the specification has to be fixed.

2: There are several techniques to mitigate incompleteness errors, one of which is to split the verification of difficult properties into smaller parts, each of which is verified on its own in a *lemma* function [127]. By explicitly calling these lemmas were needed, the user guides the SMT solver toward a solution.

- **Internal errors** report that an internal check of the tool failed. In this case, the blame is on the tool and the error always signals a bug in the implementation. Depending on the bug, there might be known workarounds in the issue tracker of the tool. To help users and tool developers identify the cause of the failure, these errors are reported on the verification unit (e.g., Rust function or method) or the precise Rust statement that was being processed by the tool when the failure happened.

In the implementation of our tools, these error kinds are described by an enumeration type that optionally contains the span in the source code that corresponds to the error. Every function in our codebase that might fail with one of these errors returns a *result* type, following the standard error handling practices of Rust.

The reason for having explicit handling of internal errors, instead of just terminating with a panic, is to ensure a good user experience even in those cases where the tool implementation is incomplete. Rust is a real-world programming language based on a complex type system. As such, developing a static tool for it requires handling many edge cases and unexpected usages. For example, and for reference for future developers that might aim to build a Rust tool from scratch, we report some of the special cases that we encountered:

- The CFG representation of some functions might not have a path between the entry point and the return point. This can happen when the function contains an intentionally-infinite loop such as `loop {}`.
- The CFG of a function might have edges that go from one branch of the CFG to another, e.g., Fig. 5.3. This can happen when the function contains `match` statements with `if` conditions on its arms. Such CFGs are *reducible*, but naive static analysis implementations might incorrectly expect that loop-free code cannot produce such shapes[3].
- The condition of a loop is an expression, which (1) can contain a loop itself, and (2) can contain `continue` statements that jump to the next loop iteration, *before* finishing the evaluation the loop condition. These cases, shown in Fig. 5.4, are usually not possible in other programming languages.
- Some definitions, such as `enum` types and `match` statements, might contain an unexpectedly high number (i.e., thousands) of variants and cases, respectively. This happens, for example, in automatically-generated code handling Unicode characters. A tool with a naive quadratic (or worse) memory complexity might run out of memory and terminate when handling these cases.
- Certain mutable borrows have an implicitly *two-phase* semantics, meaning that their properties do not hold immediately after their creation, but are postponed until the borrow is *activated* at its first usage. This rule requires special handling in verification techniques.
- Types that are zero-size have properties that might be surprising at first glance. For example, in safe Rust, it is possible to have several mutable references all active at the same time and pointing to the same memory location, provided that their target type is zero-sized. This case is permitted by Rust because, having size zero, there are no overlapping memory regions between the targets of the mutable references.

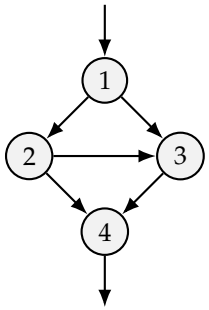


Figure 5.3: CFG with an edge between two branches.

```rust
while {
  // loop condition block
  if a { continue; }
  while b { /* ... */ }
  c
} {
  // loop body block
}
```

Figure 5.4: Rust loop with a `continue` and a nested loop in the code block that computes the loop condition.

3: The reader might be interested to know that the compiler uses some MIR transformations that can make the CFG of a Rust function irreducible [128].

- Types that do *not* implement the `Unpin` trait have special aliasing properties, by which it might be possible to have several mutable references to the same non-zero-size type all active at the same time and pointing to the same memory location.

5.2.3 IDE Integration

In addition to reporting errors on the command line, our verification tools can also be used from the Visual Studio Code IDE and its freely-licensed VSCodium distribution [54, 55]. There are several ways to integrate the error reporting of a verification tool in an IDE, with various degrees of engineering effort and customizability. Some of these options are made possible by our design choice of using the Rust compiler's error reporting infrastructure, which can report machine-readable error messages in a JSON format that is well-understood by existing tools in the Rust ecosystem [129].

[54]: (2024), *Visual Studio Code*
[55]: (2024), *VSCodium*
[129]: (2024), *The rustc book: JSON output*

5.2.3.1 Using Rust Analyzer

The simplest of all options does not involve writing new code at all, and only requires a special configuration of Rust Analyzer [120], the official Visual Studio Code extension for Rust. The extension has been built to report in the IDE the error messages that a command reports using the compiler's JSON format. By default, this command is a standard `cargo check`, which only checks that a Rust crate compiles. However, the extension can also be customized to use one of our verifiers. For example, the Rust Analyzer configuration reported in Fig. 5.5 shows how to make the extension verify crates using Prusti instead of just reporting Rust compilation errors. Despite being very simple, this approach has some nice advantages. For example, it automatically benefits from Rust Analyzer's caching to avoid re-verifying crates that do not contain new modifications, and the extension behaves reasonably well even in the cases where the tool unexpectedly crashes. However, there are also several disadvantages. Since the extension has been designed for a different use case, the user interface does not clearly distinguish a crate that has not been verified from one where the verification succeeded. It is not possible to make the extension automatically start the verification server component of our tools' architecture, so the overall performance of the tool is worse. Switching on and off the verification is tedious because it requires changing several settings of Rust Analyzer each time. Moreover, it is still up to the user to download and update the binaries of the verifier. All these issues might not be a problem for quick demos, but on real projects, they degrade the user experience.

[120]: (2024), *rust-analyzer: A Rust compiler front-end for IDEs*

```json
{
    "rust-analyzer.check.overrideCommand": [
        "/path/to/cargo-prusti",
        "--quiet",
        "--workspace",
        "--message-format=json",
    ],
}
```

JSON

Figure 5.5: An example configuration to use Prusti from Rust Analyzer (v0.3.1868). `cargo-prusti` is the Prusti binary to verify a crate.

5.2.3.2 Custom Extension

A second option, slightly more involved but much more customizable, consists of developing a new IDE extension built around a parser of the JSON output of our verification tools. This approach brings many advantages, because such an extension can automatically take care of downloading the binaries of the verifier, or checking for updates, so that the user does not have to follow complex instructions to set up their IDE. Moreover, it makes it possible to add verification-specific buttons and messages to the user interface, so that the user always knows what is the outcome of the verification. The disadvantage of this approach is that it requires developing and maintaining a project for the extension written in JavaScript or TypeScript, using a completely different set of tools compared to the development of Rust projects. Nevertheless, this is the approach that we chose when building our Prusti Assistant extension [119]. Our experience was quite pleasant, and the required maintenance effort was minimal compared to the maintenance of Prusti.

[119]: (2024), *Repository of Prusti Assistant, the IDE extension of Prusti.* URL: `https://github.com/viperproject/prusti-assistant`

Workflow At its start, Prusti Assistant automatically downloads the verification binaries if they are missing, and runs a set of pre-requisite checks to make sure that, e.g., Rust has been installed on the operating system. Then, the extension launches our Prusti verification server to cache and handle efficiently the Viper verification requests generated by our tools. When the user asks to verify a Rust program, for example by clicking on a verification button set up in the IDE, the extension launches in a child process one of our Prusti binaries: `prusti-rustc` when verifying a standalone Rust file with no dependencies (convenient for quick demos), or `cargo-prusti` when verifying a Rust crate. These processes verify the Rust program as presented in Sec. 5.1, contacting the verification server managed by the extension and writing in their output a list of verification errors in the JSON format. These errors are parsed by the IDE extension and then converted to IDE diagnostics that are displayed on the source code. If the verification takes too long, the user can decide at any moment to interrupt the verification, implemented with a command that kills the verification processes.

In addition to starting or stopping the verification of a program, the user interface provides ways to restart the verification server and clear its in-memory cache, show the exact Prusti version used by the tool, check for updates, clear all Prusti-related diagnostics, and also offers a few links of convenience to reach the user guide or the chat of our verification project. The interface of these actions is shown in Fig. 5.6. The core of this architecture — parsing the JSON output of a child process — comes from Rust Assist [130]: a simple and neat extension (now deprecated in favor of Rust Analyzer) to report Rust compilation errors in Visual Studio Code. This extension was of great help to us in kick-starting the development of our Prusti Assistant.

[130]: (2024), *The Rust compiler API: Function rustc_driver::install_ice_hook*

5.2.3.3 Language Server

A third option is to develop a custom extension that, instead of launching verification processes and parsing their JSON output, delegates all this logic to a *language server*, which communicates with the IDE through

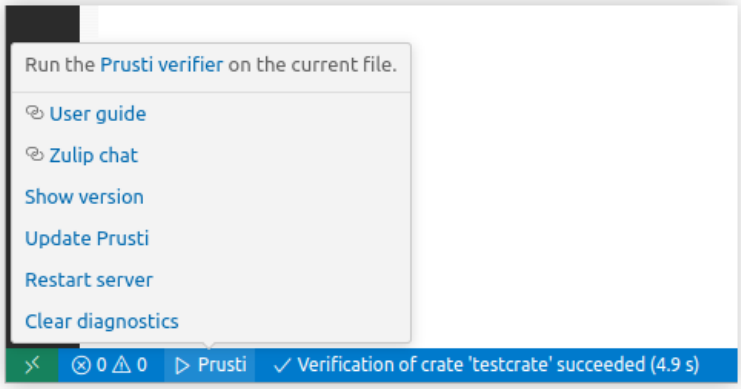


Figure 5.6: Screenshot of a menu of Prusti Assistant.

[131]: (2024), *Official page for Language Server Protocol*

the *language server protocol* [131]. The intention of this protocol, which can be used to implement functionalities such as error reporting or code navigation, is to decouple the implementation of the extension from the API of the IDE, making it possible to use the same language server in different IDEs. One advantage of this architecture, used by popular extensions such as the Rust Analyzer, is that it makes it possible to move most of the logic of the extension into a server that can be written in languages other than JavaScript or TypeScript. By choosing to implement a language server in Rust, for example, it would be easier to reuse components implemented in the Rust Analyzer. This approach, however, does not solve any of the technical challenges of the approach in Sec. 5.2.3.2. The language server, even when implemented in Rust, would still have to launch a verifier in a child process, because the compiler driver architecture of our tools does not support executing multiple instances of the Rust compiler in the same process. Moreover, the language server would still need to obtain from the verifier a list of errors to report, and the JSON output might still be the best way to do so. In our case, the language server protocol was not stable and complete enough when we started implementing Prusti Assistant, and since then we did not find a killer feature useful enough to motivate rewriting our extension to an architecture based on a language server. However, this approach might make sense for new tool developers, who might need to develop a new IDE integration from scratch.

6 Conclusions and Future Work

"Sokath. His eyes uncovered!"
— Dathon (2368)

6.1 Summary

In this thesis, we have proposed several novel techniques to verify Rust programs in an automated deductive verification tool. These techniques are *lightweight* because they have been designed to require few user annotations, while still being *rich* enough to verify expressive user-chosen properties. Chapter 2 presented why Rust is a good fit for software verification, showing the language properties, verification opportunities and design principles of our approach to Rust verification. Chapter 3 presented our technique to reason about programs and libraries entirely written in safe Rust code, which verified automatically memory safety and user-provided partial correctness properties. However, such a technique is incomplete when reasoning about some usages of libraries implemented with unsafe code, e.g., libraries exposing interior mutability. To handle those cases, Chapter 4 presented a technique to declare and reason about library-defined type capabilities on trusted libraries implemented with unsafe code, making it possible to verify, e.g., usages of popular types with interior mutability of the standard library. All our techniques have been implemented in open-source verification tools that are available online: Prusti [27] for Chapter 3 and Mendel [28] for Chapter 4. Chapter 5 described some notable implementation decisions behind these tools, which were designed to offer a particularly curated user experience, intuitively reporting verification error messages on the Rust code both when running on the command-line and in an IDE. Next, we summarize our key ideas and solutions to the challenges of Sec. 1.2.

[27]: (2024), *Repository of the Prusti verifier for safe Rust*. URL: https://github.com/viperproject/prusti-dev

[28]: (2024), *Repository of the Mendel verifier for safe Rust clients of interior mutability*. URL: https://github.com/viperproject/mendel-verifier

Modeling Type Properties A common goal of our verification techniques is to model relevant properties of Rust's type system, using — and optionally checking — them during the program verification. In Chapter 3, this is done by modeling Rust types in the implicit-dynamic-frames logic, by (1) modeling the read and write capabilities of memory locations using Viper's fractional permissions, (2) modeling the composition of types using nested Viper predicates, and (3) modeling reborrowing cases using Viper's magic wands. In Chapter 4, the type properties are modeled in first-order logic by defining a fine-grained system of capabilities, a subset of which correspond to the explicit capabilities of shared and mutable references of safe Rust code. This enables reasoning about the memory locations that are managed by Rust libraries implemented with unsafe code, and that cannot be described using the explicit capabilities of regular Rust types.

Annotations Our techniques provide program annotations that enable specifying functional correctness properties and type properties. These

annotations are useful in two ways. First, they enable the realization of modular deductive verification techniques, as we do. Second, they make it possible to state explicitly several program properties on which software developers implicitly rely on. For example, in addition to functional properties and loop invariant properties, the purity annotations introduced by Chapter 3 make it possible to declare that the implementation of certain functions is deterministic, does not have side effects, and depends only on the values of its arguments. In Chapter 4, this idea is brought forward by introducing new annotations for pure functions whose result can additionally depend on the memory addresses reachable from the function arguments. Moreover, the implicit capability annotations defined in the same chapter make it possible for Rust developers to clearly state several properties of the memory locations that it manages via unsafe code: immutability, uniqueness, non-aliasing, thread-local usage, and so on.

Automation Our techniques have been designed for automation, which is achieved in two steps. First, we developed our techniques to be modular, in that they verify each Rust function independently from the implementation of other functions. This is made possible by the program annotations, which provide an abstraction of the function behavior that can be used in place of the implementation. Second, we constructed our techniques to work on top of an SMT solver. The choice of using the Viper verification language greatly helped in the realization of both these two steps.

Usability A distinguishing feature of our techniques is their high usability, which is achieved in several ways. First, we designed the language of specifications to be as familiar as possible to Rust developers. For example, declaring boolean conditions in contract annotations can be done using the exact same syntax and semantics as Rust. Second, we designed our tools to be user-friendly in the way they report verification errors and other possible failures. Notably, our tools report verification errors on the source-code level — the same level on which developers reason and write program annotations. This minimizes fatiguing context-switches for the users. To ensure a good user experience, we implemented our tools to be easily usable from both the command line and from an IDE: on the command line, our error messages are reported using the familiar syntax of Rust's compiler errors; in the IDE, our errors are reported as native IDE diagnostics.

Overall, we believe our work had a great impact on the verification and programming languages community, showing that it is possible to leverage the Rust types to simplify the task of verification, encouraging this way further lines of research. For example, following our publication of Prusti [29] in 2019, RustHorn [85] and Creusot [86] showed how borrowing relations can be encoded into first-order logic using prophecy variables, enabling more efficient verification without constructing a memory-safety proof. Verus [62] showed how Rust's type checks can be used to build new Rust-like languages that are particularly well-suited for verification. Flowistry [89] showed how Rust's type system can additionally help and bring precision to static analysis techniques. Flux [81] showed how Rust types enable efficient liquid-type-based

[29]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*

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

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

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

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

[81]: Lehmann et al. (2023), *Flux: Liquid Types for Rust*

verification of some functional correctness properties.

6.2 Future Work

There are many directions along which our verification techniques for verification of safe code (Ch. 3) or safe clients (Ch. 4) can evolve. Most of them are specific to one of these two techniques, but a natural extension of our work is to unify the two techniques, encoding the capabilities of Ch. 4 in separation logic as described at the end of this section.

6.2.1 Verification of Safe Code

Formalization Our encoding of Rust programs into Viper is based on a soundness principle by which if the verification of the Viper program succeeds, then the execution of the Rust program is guaranteed to never fail. We have justified the correctness of our approach on paper [29] and we continuously validate in our test suite that the verifier behaves as expected. However, both approaches have their limitations. To fill this gap, it would be interesting to formalize our verification technique in a foundational way. One promising approach could be using Iris [58] and building upon RustBelt [101] or its various evolutions [102, 115].

[29]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*

[58]: Jung et al. (2018), *Iris from the ground up: A modular foundation for higher-order concurrent separation logic*

[101]: Jung et al. (2018), *RustBelt: Securing the foundations of the Rust programming language*

[102]: Matsushita et al. (2022), *RustHornBelt: A semantic foundation for functional verification of Rust programs with unsafe code*

[115]: Gäher et al. (2024), *RefinedRust: A Type System for High-Assurance Verification of Rust Programs*

Specification Language Prusti currently does not support reasoning about memory addresses or object identifiers. Designing new specifications to reason about these program properties would enable new specification approaches, similar to those introduced by Verus. As discussed in Ch. 3, it might also be possible to re-design the pledge specifications to offer a more generic syntax inspired by Creusot's prophecies. This is not trivial because it might require generating new kinds of magic wands in the Viper encoding, as well as proving a formal connection between them and the prophecies formulation of specifications.

Rust Language Features By constructing a core memory-safety proof, our technique is well-suited to be extended to the verification of unsafe code, making it a natural extension of our work. For example, research in this direction might design new annotations to associate special resources with the memory locations used in unsafe code. Within safe code, our technique can still be extended to model more language features, such as complex cases of borrows, traits and lifetime annotation. All these require solving challenges in the generation and automation of the program proof.

Verification-Enabled Optimizations The program reasoning that is possible in a verifier is usually more advanced than the compiler reasoning that is used to enable optimizations. One interesting research direction to explore consists of feeding back useful verified properties to the compiler, in order to enable new optimizations. For example, this approach might elide the redundant runtime checks that we identified in the second part of our evaluation of Ch. 3. A research challenge in this

direction is to prove that none of the new optimizations will introduce undefined behavior.

Shared Tool Framework As visible in the discussion of the related work of Ch. 3, there are some recurring concepts that have been implemented with some variation in multiple works. For example, the construction of a graph representation of the borrowing relations, or a language for contract annotations used by deductive (and other) verifiers. This duplication leads to duplicate engineering effort in the implementation, and additional work for the user when switching from one verification tool to the other. This can be mitigated by designing a shared technique-agnostic verification framework between tools, and by publishing tool components such as the analysis that computes the borrowing graph or the capability analysis as libraries.

6.2.2 Verification of Safe Clients of Interior Mutability

Formalization Similar to our approach in Ch. 3, we have justified on paper the soundness of our encoding of Rust programs into Viper (cf., Sec. 4.4, Sec. 4.5) and we have developed a test suite to validate the verifier's behavior on numerous programs. Moving forward, it would be interesting to formalize our verification technique in a foundational way. For example, using Iris [58] and building upon RustBelt [101] or its many evolutions [102, 115]. In particular, we expect that the correctness of each of our library capability annotations might be proven as a lemma stating that the capability (expressed as a semantic invariant) associated with the content of a library type with interior mutability can be obtained from the capability of the library type that contains it.

[58]: Jung et al. (2018), *Iris from the ground up: A modular foundation for higher-order concurrent separation logic*

[101]: Jung et al. (2018), *RustBelt: Securing the foundations of the Rust programming language*

[102]: Matsushita et al. (2022), *RustHornBelt: A semantic foundation for functional verification of Rust programs with unsafe code*

[115]: Gäher et al. (2024), *RefinedRust: A Type System for High-Assurance Verification of Rust Programs*

Automated Core Capability Discovery In our work, we require the user to write trusted capability annotations on the libraries implemented with unsafe code. However, some of the core capabilities of a library are already implied by the implementation of the library. For example, any library implementing the `Deref` trait provides a `deref` method that returns a shared reference to a memory location that, in some cases, is computed by the library by casting a raw pointer. Because of this, it should be possible to develop a whole-program static analysis technique that analyzes the source code of a library to identify a subset of its implicit core capabilities. These discovered capabilities could be then used to suggest capability annotations to the user, or could be directly used in our verification technique.

Soundness-Based Optimizations The semantics-preserving transformation (SPT) technique that we presented can be applied to other contexts. One example is to motivate a new kind of optimization, which can only be applied to safe Rust code that is known to never be called from unsafe code. Our insight is that there seem to be more optimizations possible in this safe subset of Rust than in the general Rust language. Optimizations of this kind would have the nice side effect of encouraging developers to avoid unsafe code in order to benefit from the potential additional optimizations.

Fuzz Testing Soundness A second possible usage of our SPT technique is the realization of automated techniques for testing the soundness of Rust libraries. The idea is to use our SPTs as random mutations in a fuzzing tool whose aim is to detect sequences of safe API calls that demonstrate that a Rust library is unsound. Concretely, the fuzzer could start from a set of seed programs that call small sequences of API calls. These seed programs could be, for example, the programs in the test suite of the library to be fuzzed. Then, the fuzzer could generate random mutations using our SPTs, checking the result of each mutation with Miri to identify which of them have any UB. Since our transformations only preserve the behavior of a program under the hypothesis that all used libraries are sound, finding UB in any of the modified programs is only possible if one of the libraries (or their annotations) is unsound.

Separation-Logic Encoding In our work, we defined a first-order-logic model of the capabilities of Rust types. The choice of using first-order logic derives from flexibility and automation needs, that we felt were more challenging to satisfy using a separation logic. However, we think that separation-logic model of our capabilities is possible. The core idea is that each of the implications between capabilities in our technique could be expressed using nested logical predicate definitions in separation logic, while non-aliasing properties could be expressed using a resource algebra similar to what done in concurrent abstract predicates [132]. A separation-logic encoding of this kind would enable unifying our verification technique of Ch. 4 with Ch. 3.

[132]: Dinsdale-Young et al. (2010), *Concurrent Abstract Predicates*

Language Support Even though the Mendel verifier that we implemented in Ch. 4 uses the safe verification framework of Prusti, and thus benefits from the various same usability benefits among which error reporting and IDE integration, the Rust language support of Mendel is still quite limited. A possible direction is to continue the engineering effort and implement support for quantifications, loops, arrays, slices, and two-phase borrows. This would greatly increase the set of verifiable clients, to the point where Mendel could be tested using almost the entire test suite of Prusti. Continuing in this direction, one of the main challenges could be to add support for checking pledge annotations in Mendel. This should be possible by taking inspiration from the internal checks that Viper does to check packaging of magic wands.

Bibliography

[1] Stack Overflow. *Stack Overflow developer survey 2023*. URL: https://survey.stackoverflow.co/2023 (cited on page 1).

[2] 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 1, 88, 93).

[3] 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 1, 11, 12, 36).

[4] 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 page 1).

[5] Leslie Lamport. 'Proving the correctness of multiprocess programs'. In: *IEEE trans. software eng.* 3.2 (1977), pp. 125–143. DOI: 10.1109/TSE.1977.229904 (cited on page 2).

[6] Douglas D. Dunlop and Victor R. Basili. 'A comparative analysis of functional correctness'. In: *ACM comput. surv.* 14.2 (1982), pp. 229–244. DOI: 10.1145/356876.356881 (cited on page 2).

[7] Amir Pnueli. 'The temporal logic of programs'. In: *18th annual symposium on foundations of computer science, providence, rhode island, usa, 31 october - 1 november 1977*. IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32 (cited on page 2).

[8] Edmund M Clarke and E Allen Emerson. 'Design and synthesis of synchronization skeletons using branching time temporal logic'. In: *Workshop on logic of programs*. Springer. 1981, pp. 52–71 (cited on page 2).

[9] 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 2).

[10] Patrick Cousot and Radhia Cousot. 'Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints'. In: *Conference record of the fourth ACM symposium on principles of programming languages, los angeles, california, usa, january 1977*. Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973 (cited on page 2).

[11] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. *Model checking, 2nd edition*. MIT Press, 2018 (cited on page 2).

[12] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development - Coq'Art: The calculus of inductive constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004 (cited on pages 3, 36).

[13] 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 3).

[14] 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 page 3).

[15] 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).

[16] 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 3, 36, 72, 89, 146).

[17] 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 3, 36, 72, 90, 145).

[18] 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 3).

[19] 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 pages 3, 72, 130, 146).

[20] 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 3, 36, 72).

[21] Bernard Carré and Johnathan Randall Garnsworthy. 'SPARK - an annotated Ada subset for safety-critical programming'. In: *Proceedings of the conference on TRI-ADA 1990, tri-ada 1990, baltimore, maryland, usa, december 3-6, 1990*. Ed. by Charles B. Engle Jr. ACM, 1990, pp. 392–402. DOI: 10.1145/255471.255563 (cited on page 3).

[22] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 'Frama-C - A software analysis perspective'. In: *Software engineering and formal methods - 10th international conference, SEFM 2012, thessaloniki, greece, october 1-5, 2012. proceedings*. Ed. by George Eleftherakis, Mike Hinchey, and Mike Holcombe. Vol. 7504. Lecture Notes in Computer Science. Springer, 2012, pp. 233–247. DOI: 10.1007/978-3-642-33826-7_16 (cited on page 3).

[23] 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 3, 12, 72, 129).

[24] 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 4, 15, 36).

[25] 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 pages 4, 72, 153).

[26] 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 pages 4, 72).

[27] *Repository of the Prusti verifier for safe Rust*. 2024. URL: https://github.com/viperproject/prusti-dev (cited on pages 6, 36, 43, 83, 159).

[28] *Repository of the Mendel verifier for safe Rust clients of interior mutability*. 2024. DOI: 10.5281/zenodo.12751898. URL: https://github.com/viperproject/mendel-verifier (cited on pages 6, 118, 140, 147, 159).

[29] 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 7, 33, 43, 82, 86, 93, 97, 118, 160, 161).

[30] 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 page 7).

[31] Vytautas Astrauskas, Aurel Bílý, Jonás Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 'The Prusti project: Formal verification for Rust'. In: *NASA formal methods - 14th international symposium, NFM 2022, pasadena, ca, usa, may 24-27, 2022, proceedings*. Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez. Vol. 13260. Lecture Notes in Computer Science. Springer, 2022, pp. 88–108. DOI: 10.1007/978-3-031-06773-0_5 (cited on pages 7, 43, 83, 149).

[32] Shuofei Zhu, Ziyi Zhang, Boqin Qin, Aiping Xiong, and Linhai Song. 'Learning and programming challenges of Rust: A mixed-methods study'. In: *44th IEEE/ACM 44th international conference on software engineering, ICSE 2022, pittsburgh, pa, usa, may 25-27, 2022*. ACM, 2022, pp. 1269–1281. DOI: 10.1145/3510003.3510164 (cited on pages 9, 11).

[33] Anna Zeng and Will Crichton. 'Identifying barriers to adoption for Rust through online discourse'. In: *9th workshop on evaluation and usability of programming languages and tools, plateau@splash 2018, november 5, 2018, boston, massachusetts, USA*. Ed. by Titus Barik, Joshua Sunshine, and Sarah E. Chasins. Vol. 67. OASIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 5:1–5:6. DOI: 10.4230/OASIcs.PLATEAU.2018.5 (cited on pages 9, 11).

[34] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 'Undefined behavior: What happened to my code?' In: *Asia-pacific workshop on systems, apsys '12, seoul, republic of korea, july 23-24, 2012*. ACM, 2012, p. 9. DOI: 10.1145/2349896.2349905 (cited on page 9).

[35] *Glossary definition of soundness*. 2019. URL: https://github.com/rust-lang/unsafe-code-guidelines/blob/5325cdedeca312dd242227fc8899c150356ec965/reference/src/glossary.md#soundness-of-code--of-a-library (cited on page 10).

[36] *The Rust Reference: Interior mutability*. 2024. URL: https://doc.rust-lang.org/reference/interior-mutability.html (cited on page 12).

[37] *Unsafe Code Guidelines Reference: Interior mutability*. 2024. URL: https://rust-lang.github.io/unsafe-code-guidelines/glossary.html#interior-mutability (cited on page 12).

[38] Werner Dietl and Peter Müller. 'Object ownership in program verification'. In: *Aliasing in object-oriented programming. types, analysis and verification*. Springer, 2013, pp. 289–318 (cited on page 12).

[39] Jean-Yves Girard. 'Linear logic'. In: *Theor. comput. sci.* 50 (1987), pp. 1–102. DOI: 10.1016/0304-3975(87)90045-4 (cited on page 13).

[40] *The Rust Reference*. 2024. URL: https://doc.rust-lang.org/reference (cited on page 16).

[41] *The Rust Reference: Behavior considered undefined*. 2023. URL: https://doc.rust-lang.org/reference/behavior-considered-undefined.html (cited on page 16).

[42] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 'Stacked Borrows: An aliasing model for Rust'. In: *Proceedings of the ACM on programming languages* 4.POPL (2020), 41:1–41:32. DOI: 10.1145/3371109 (cited on pages 16, 17, 92, 126).

[43] Neven Villani. *Tree Borrows: A new aliasing model for Rust*. 2024. URL: https://perso.crans.org/vanille/treebor/ (cited on pages 17, 92, 126).

[44] *Miri: An interpreter for Rust's mid-level intermediate representation*. 2024. URL: https://github.com/rust-lang/miri (cited on pages 17, 92, 150).

[45] David Tolnay. *Soundness bugs*. 2019. URL: https://docs.rs/dtolnay/latest/dtolnay/macro._03_soundness_bugs.html (cited on page 18).

[46] *RustSec advisory database: Advisories with keyword 'unsound'*. 2024. URL: https://rustsec.org/keywords/unsound.html (cited on page 18).

[47] *Constructor of core::iter::StepBy*. 2023. URL: https://github.com/rust-lang/rust/blob/cd71a37f320c379df47ff64abd934f3a2da94c26/library/core/src/iter/adapters/step_by.rs#L30-L37 (cited on page 20).

[48] *The rust-osdev/volatile crate*. 2024. URL: https://github.com/rust-osdev/volatile (cited on page 21).

[49] *Issue #26 in the rust-osdev/volatile crate*. 2022. URL: https://github.com/rust-osdev/volatile/issues/26 (cited on page 22).

[50] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 'Transparent runtime randomization for security'. In: *22nd symposium on reliable distributed systems (SRDS 2003), 6-8 october 2003, florence, italy*. IEEE Computer Society, 2003, p. 260. DOI: 10.1109/RELDIS.2003.1238076 (cited on page 22).

[51] *Square root implementation in the ontology-wasm-cdt-rust crate*. 2020. URL: https://github.com/ontio/ontology-wasm-cdt-rust/blob/b0c6c49ce0756dbfdddc032deda29356522a3b0a/ontio-std/src/utils.rs#L7-L29 (cited on page 23).

[52] *Issue #29723 in rustc: Variables moved from in match guards are still accessible in other match arms*. 2015. URL: https://github.com/rust-lang/rust/issues/29723 (cited on pages 23, 24).

[53] *Fuzz-gen tool implementation in the WaVe project*. 2022. URL: https://github.com/PLSysSec/wave/tree/b63b2b734b211b1cc729e8efbc00d5cb19b73415/tools/fuzz-gen (cited on page 29).

[54] *Visual Studio Code*. 2024. URL: https://code.visualstudio.com/ (cited on pages 30, 31, 156).

[55] *VSCodium*. 2024. URL: https://vscodium.com/ (cited on pages 30, 31, 156).

[56] 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 36).

[57] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. *Software artifact for the OOPSLA'19 paper titled "Leveraging Rust types for modular specification and verification"*. 2019. DOI: 10.5281/zenodo.3363914 (cited on pages 36, 83).

[58] 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: *Journal of functional programming* 28 (2018), e20. DOI: 10.1017/S0956796818000151 (cited on pages 36, 126, 145, 161, 162).

[59] 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 37).

[60] Litao Zhou, Jianxing Qin, Qinshi Wang, Andrew W. Appel, and Qinxiang Cao. 'VST-A: A foundationally sound annotation verifier'. In: *Proc. ACM program. lang.* 8.POPL (2024), pp. 2069–2098. DOI: 10.1145/3632911 (cited on page 50).

[61] *Rust Compiler Development Guide: Glossary*. 2024. URL: https://rustc-dev-guide.rust-lang.org/appendix/glossary.html (cited on page 54).

[62] 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*. 2023. DOI: 10.1145/3586037. URL: https://doi.org/10.1145/3586037 (cited on pages 58, 90, 93, 118, 143, 145, 160).

[63] Nils Becker, Peter Müller, and Alexander J. Summers. 'The axiom profiler: Understanding and debugging SMT quantifier instantiations'. In: *Tools and algorithms for the construction and analysis of systems - 25th international conference, TACAS 2019, held as part of the european joint conferences on theory and practice of software, ETAPS 2019, prague, czech republic, april 6-11, 2019, proceedings, part I*. Ed. by Tomás Vojnar and Lijun Zhang. Vol. 11427. Lecture Notes in Computer Science. Springer, 2019, pp. 99–116. DOI: 10.1007/978-3-030-17462-0_6 (cited on page 72).

[64] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. *Verifast: A powerful, sound, predictable, fast verifier for C and Java*. Tech. rep. 2011, pp. 41–55. DOI: 10.1007/978-3-642-20398-5_4 (cited on page 72).

[65] Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers. 'Verification condition generation for permission logics with abstract predicates and abstraction functions'. 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. 451–476. DOI: 10.1007/978-3-642-39038-8_19 (cited on page 72).

[66] Vytautas Astrauskas. 'Leveraging uniqueness for modular verification of heap-manipulating programs'. PhD thesis. ETH Zurich, 2024 (cited on page 81).

[67] *Clippy: A collection of lints to catch common mistakes and improve your Rust code*. 2024. URL: https://github.com/rust-lang/rust-clippy (cited on pages 83, 150).

[68] *The Rust community's crate registry*. 2024. URL: https://crates.io (cited on page 84).

[69] *Rosetta Code, Category Rust*. 2024. URL: https://rosettacode.org/wiki/Category:Rust (cited on pages 85, 86).

[70] *Learn Rust by writing entirely too many linked lists, first final code*. 2024. URL: https://rust-unofficial.github.io/too-many-lists/first-final.html (cited on pages 85, 86).

[71] Nicholas D. Matsakis. *MIR-based borrowck is almost here*. 2018. URL: http://smallcultfollowing.com/babysteps/blog/2018/10/31/mir-based-borrowck-is-almost-here/ (cited on pages 85–87).

[72] Nicholas D. Matsakis. *MIR-based borrow check (NLL) status update*. 2018. URL: http://smallcultfollowing.com/babysteps/blog/2018/06/15/mir-based-borrow-check-nll-status-update (cited on pages 85–87).

[73] 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 page 88).

[74] 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 page 88).

[75] 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 88, 108, 146).

[76] 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 trans. program. lang. syst.* 36.1 (2014), 2:1–2:42. DOI: 10.1145/2543920 (cited on page 88).

[77] Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 'Reference capabilities for flexible memory management: Extended version'. In: *Corr* abs/2309.02983 (2023). DOI: 10.48550/ARXIV.2309.02983 (cited on page 88).

[78] *The Rust programming language*. 2024. URL: https://www.rust-lang.org/ (cited on page 88).

[79] 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 88).

[80] 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 88).

[81] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 'Flux: Liquid types for Rust'. In: *Proc. ACM program. lang.* 7.PLDI (2023), pp. 1533–1557. DOI: 10.1145/3591283 (cited on pages 88, 160).

[82] 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 page 88).

[83] 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 88).

[84] Claire Dross. *Pointer based data-structures in SPARK*. 2019. URL: https://blog.adacore.com/pointer-based-data-structures-in-spark (cited on page 88).

[85] 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 89, 90, 160).

[86] 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 89, 90, 93, 118, 143, 145, 160).

[87] 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: *Corr* abs/2110.08362 (2021) (cited on page 89).

[88] Zachary Amsden, Ramnik Arora, Shehar Bano, Mathieu Baudet, Sam Blackshear, Abhay Bothra, George Cabrera, Christian Catalini, Konstantinos Chalkias, Evan Cheng, Avery Ching, Andrey Chursin, George Danezis, Gerardo Di Giacomo, David L. Dill, Hui Ding, Nick Doudchenko, Victor Gao, Zhenhuan Gao, François Garillot, Michael Gorven, Philip Hayes, J. Mark Hou, Yuxuan Hu, Kevin Hurley, Kevin Lewi, Chunqi Li, Zekun Li, Dahlia Malkhi, Sonia Margulis, Ben Maurer, Payman Mohassel, Ladi de Naurois, Valeria Nikolaenko, Todd Nowacki, Oleksandr Orlov, Dmitri Perelman, Alistair Pott, Brett Proctor, Shaz Qadeer, Rain, Dario Russi, Bryan Schwab, Stephane Sezer, Alberto Sonnino, Herman Venter, Lei Wei, Nils Wernerfelt, Brandon Williams, Qinfan Wu, Xifan Yan, Tim Zakian, and Runtian Zhou. *The Libra blockchain*. 2020. URL: https://developers.libra.org/docs/the-libra-blockchain-paper (cited on page 89).

[89] 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 pages 89, 152, 160).

[90] 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 page 89).

[91] 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 page 89).

[92] 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 page 89).

[93] Kani. *Kani Rust verifier*. 2024. URL: https://github.com/model-checking/kani (cited on page 89).

[94] Daniel Kroening, Peter Schrammel, and Michael Tautschnig. 'CBMC: The C bounded model checker'. In: *Corr* abs/2302.02384 (2023). DOI: 10.48550/ARXIV.2302.02384 (cited on page 89).

[95] *Kani RFC book: Function contracts*. 2023. URL: https://model-checking.github.io/kani/rfc/rfcs/0009-function-contracts.html (cited on page 89).

[96] 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 90).

[97] Sebastian Ullrich. 'Simple verification of Rust programs via functional purification'. Master's Thesis. Karlsruhe Institute of Technology, Dec. 2016 (cited on page 90).

[98] 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 90).

[99] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 'Combinators for bi-directional tree transformations: Linguistic approach to the view update problem'. In: *ACM SIGPLAN notices* 50.8s (2015), pp. 49–62. DOI: 10.1145/2854695.2854700 (cited on page 90).

[100] 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 90).

[101] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 'RustBelt: Securing the foundations of the Rust programming language'. In: *Proceedings of the ACM on programming languages* 2.POPL (2018), 66:1–66:34. DOI: 10.1145/3158154 (cited on pages 91, 126, 145, 161, 162).

[102] 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, 145, 161, 162).

[103] 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 pages 91, 93, 146).

[104] Eric Reed. 'Patina: A formalization of the Rust programming language'. In: *University of washington, department of computer science and engineering* (2015). Technical Report UW-CSE-15-03-02, p. 264 (cited on page 92).

[105] 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 92).

[106] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. 'KRust: A formal executable semantics of Rust'. In: (2018). Ed. by Jun Pang, Chenyi Zhang, Jifeng He, and Jian Weng, pp. 44–51. DOI: 10.1109/TASE.2018.00014 (cited on page 92).

[107] Aaron Weiss, Daniel Patterson, and Amal Ahmed. 'Rust distilled: An expressive tower of languages'. In: *Corr* abs/1806.02693 (2018) (cited on page 92).

[108] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 'Oxide: The essence of Rust'. In: *Corr* abs/1903.00982 (2019) (cited on page 92).

[109] Nicholas D. Matsakis. *An alias-based formulation of the borrow checker*. 2018. URL: https://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/ (cited on page 92).

[110] 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 page 108).

[111] James Noble, Julian Mackay, and Tobias Wrigstad. *Rusty links in local chains*. 2022. DOI: 10.48550/ARXIV.2205.00795. URL: https://arxiv.org/abs/2205.00795 (cited on page 111).

[112] Nikita Popov. *Rust pull request #82834: Enable mutable noalias for LLVM >= 12*. 2021. URL: https://github.com/rust-lang/rust/pull/82834 (cited on pages 112, 129).

[113] Rust. *The Rust standard library, version 1.69.0 (84c898d65 2023-04-16)*. 2023. URL: https://doc.rust-lang.org/std/ (cited on page 126).

[114] *The Rust standard library, Struct std::sync::Arc, Function weak_count*. 2024. URL: https://doc.rust-lang.org/std/sync/struct.Arc.html#method.weak_count (cited on page 142).

[115] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 'RefinedRust: A type system for high-assurance verification of Rust programs'. In: *PLDI '24: 42nd ACM SIGPLAN international conference on programming language design and implementation*. To appear. 2024 (cited on pages 146, 161, 162).

[116] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 'RefinedC: Automating the foundational verification of C code with refined ownership types'. In: *PLDI '21: 42nd ACM SIGPLAN international conference on programming language design and implementation, virtual event, canada, june 20-25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 158–174. DOI: 10.1145/3453483.3454036 (cited on page 146).

[117] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. 'Local verification of global invariants in concurrent programs'. In: *Computer aided verification, 22nd international conference, CAV 2010, edinburgh, uk, july 15-19, 2010. proceedings*. Ed. by Tayssir Touili, Byron Cook, and Paul B. Jackson. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 480–494. DOI: 10.1007/978-3-642-14295-6_42 (cited on page 146).

[118] *Rust compiler development guide: rustc_driver and rustc_interface*. 2024. URL: https://rustc-dev-guide.rust-lang.org/rustc-driver.html (cited on page 149).

[119] *Repository of Prusti Assistant, the ide extension of Prusti*. 2024. URL: https://github.com/viperproject/prusti-assistant (cited on pages 150, 157).

[120] *Rust-analyzer: A Rust compiler front-end for IDEs*. 2024. URL: https://github.com/rust-lang/rust-analyzer (cited on pages 150, 156).

[121] *Procedural macros*. 2024. URL: https://doc.rust-lang.org/reference/procedural-macros.html (cited on page 150).

[122] Vaughan R. Pratt. 'Top down operator precedence'. In: *Conference record of the ACM symposium on principles of programming languages, boston, massachusetts, usa, october 1973*. Ed. by Patrick C. Fischer and Jeffrey D. Ullman. ACM Press, 1973, pp. 41–51. DOI: 10.1145/512927.512931 (cited on page 150).

[123] *The syn crate*. 2024. URL: https://github.com/dtolnay/syn (cited on page 150).

[124] *Issue #46420 in rustc: MIR InstCombine introduces copies of mutable borrows*. 2017. URL: https://github.com/rust-lang/rust/issues/46420 (cited on page 152).

[125] Vytautas Astrauskas. *Rust pull request #86977: Enable compiler consumers to obtain mir::Body with Polonius facts*. 2021. URL: https://github.com/rust-lang/rust/pull/86977 (cited on page 152).

[126] Facebook. *MIRAI: An abstract interpreter for the Rust compiler's mid-level intermediate representation*. 2024. URL: https://github.com/facebookexperimental/MIRAI (cited on page 153).

[127] *Dafny documentation: Lemmas and induction*. 2024. URL: https://dafny.org/dafny/OnlineTutorial/Lemmas (cited on page 154).

[128] *Issue #114047 in rustc: Can MIR have irreducible control flow?* 2023. URL: https://github.com/rust-lang/rust/issues/114047 (cited on page 155).

[129] *The rustc book: JSON output*. 2024. URL: https://doc.rust-lang.org/rustc/json.html (cited on page 156).

[130] *The Rust compiler API: Function rustc_driver::install_ice_hook*. 2024. URL: https://github.com/mooman219/rust-assist (cited on page 157).

[131] *Official page for language server protocol*. 2024. URL: https://microsoft.github.io/language-server-protocol/ (cited on page 158).

[132] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 'Concurrent abstract predicates'. 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. 504–528. DOI: 10.1007/978-3-642-14107-2_24 (cited on page 163).