Vérification déductive de programmes Rust

Xavier Denis

université
PARIS-SACLAY

Deductive Verification of Rust Programs

Vérification déductive de programmes Rust

Thèse de doctorat de l'Université Paris-Saclay

École doctorale n° 580, sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : **Informatique**
Graduate School : Informatique et Sciences du Numérique
Référent : Faculté des sciences d'Orsay

Thèse préparée dans l'unité de recherche LMF (Université Paris-Saclay, CNRS, ENS Paris-Saclay),
sous la direction de **Claude MARCHÉ**, directeur de recherche,
et le co-encadrement de **Jacques-Henri JOURDAN**, chargé de recherche

Thèse soutenue à Paris-Saclay, le 18 Décembre 2023, par

Xavier DENIS

Composition du jury

Membres du jury avec voix délibérative

François POTTIER Directeur de Recherche, INRIA	President
Sylvain BOULMÉ Maître de Conférences (HDR), Université Grenoble Alpes	Rapporteur & Examinateur
Peter MÜLLER Professor, ETH Zürich	Rapporteur & Examinateur
Claire DROSS Docteure, AdaCore	Examinatrice
Ralf JUNG Assistant Professor, ETH Zürich	Examinateur
Mihaela SIGHIREANU Professor, Université Paris-Saclay	Examinatrice

THESE DE DOCTORAT
NNT : 2023UPASG101

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Vérification déductive de programmes Rust
Mots clés: Rust, appartenance, vérification, itérateurs, clôtures

Résumé: Rust est un langage de programmation introduit en 2015, qui apporte au programmeur des éléments de sûreté concernant l'utilisation de la mémoire. Le but de cette thèse est le développement d'un outil de vérification déductive pour le langage Rust, en exploitant les spécificités de son système de types afin notamment de simplifier la gestion de l'aliasing mémoire. Une telle approche de vérification permet de s'assurer de l'absence d'erreurs à l'exécution des programmes considérés, ainsi que leur conformité vis-a-vis d'une spécification formelle du comportement fonctionnel attendu. Le fondement théorique de l'approche proposé dans cette thèse est d'utiliser une notion de prophétie qui permet d'interpréter les emprunts mutables du langage Rust en une valeur courante et une valeur future cet emprunt. L'assistant de preuve Coq a été utilisé pour formaliser cet encodage prophétique et prouver la correction de la génération d'obligation de preuves associée. Par ailleurs l'approche a été mise en œuvre dans une implémentation d'un logiciel de vérification pour Rust qui automatise la génération des obligations de preuve et fait appel à des solveurs externes pour valider ces obligations. Afin de supporter les itérateurs de Rust, une extension a été développée pour manipuler les clôtures ainsi qu'une technique de vérification pour les itérateurs et combinateurs. L'implémentation a été évaluée expérimentalement sur des exemples d'algorithmes et structures de données pertinentes. Elle a été également validée par une étude de cas conséquente: la vérification d'un solveur de satisfiabilité modulo theories (SMT).

Title: Deductive Verification of Rust Programs
Keywords: Rust, ownership, verification, iterators, closures

Abstract: Rust is a programming language introduced in 2015, which provides the programmer with safety features regarding the use of memory. The goal of this thesis is the development of a deductive verification tool for the Rust language, by leveraging the specificities of its type system, in order to simplify memory aliasing management, among other things. Such a verification approach ensures the absence of errors during the execution of the considered programs, as well as their compliance with a formal specification of the expected functional behavior. The theoretical foundation of the approach proposed in this thesis is to use a notion of prophecy that interprets the mutable borrows in the Rust language as a current value and a future value of this borrow. The Coq proof assistant was used to formalize this prophetic encoding and prove the correctness of the associated proof obligation generation. Furthermore, the approach has been implemented in a verification software for Rust that automates the generation of proof obligations and relies on external solvers to validate these obligations. In order to support Rust iterators, an extension has been developed to manipulate closures, as well as a verification technique for iterators and combinators. The implementation has been experimentally evaluated on relevant algorithm and data structure examples. It has also been validated through a significant case study: the verification of a satisfiability modulo theories (SMT) solver.

Résumé Français

La vérification formelle des logiciels système à grande échelle reste aujourd'hui un défi. Une partie de cette difficulté découle des langages traditionnels (C/C++) utilisés pour ces tâches, qui obligent à considérer des défis supplémentaires liés à la sureté de la mémoire. En 2015, le langage de programmation Rust a été publié avec l'objectif de résoudre les problèmes de sureté de la mémoire dans les logiciels système. Pour ce faire, Rust utilise un nouveau *système de types de possession* capable de raisonner statiquement sur la sureté des pointeurs et de la mémoire. En gérant cela au niveau du système de types, Rust peut alléger la charge de la vérification, lui permettant d'atteindre une plus grande échelle et complexité. Cette thèse explore cette possibilité en développant un vérificateur formel appelé Creusot, qui tire parti des invariants du système de types Rust pour pousser la vérification vers des programmes plus grands et plus complexes.

Le reste de cette thèse décrit la conception, la mise en œuvre et l'utilisation du vérificateur Creusot. À ce stade de la thèse, Creusot peut vérifier du code Rust *sûr* en utilisant la majorité des fonctionnalités de langage disponibles. Creusot est distribué sous forme de logiciel libre sous une licence LGPL et peut être trouvé en ligne[1].

Le reste de ce manuscrit est structuré de la manière suivante. Après un aperçu des concepts de base dans Chapter 2, nous présentons Creusot du point de vue de l'utilisateur, en parcourant la preuve d'une petite routine dans le Chapter 3.

Dans Chapter 4, nous présentons une description complète de l'approche de Creusot pour la génération de conditions de vérification. Nous essayons de ne pas simplifier la traduction mais de décrire précisément ce qui est implémenté. Cette traduction est le cœur de Creusot et constitue la contribution la plus significative de cette thèse. Nous présentons également les défis associés à diverses fonctionnalités avancées de Creusot comme le *reborrowing logique* et les problèmes d'implémentation rencontrés lors de la manipulation du *système de traits* et des *fermetures* de Rust.

Dans Chapter 5, nous discutons de la formalisation d'une partie centrale de la traduction de Creusot à travers le projet RustHornBelt. Ce travail étend le projet précédent RustBelt pour raisonner sur la *correction* des programmes Rust lorsqu'ils sont équipés de spécifications. RustHornBelt fournit une preuve sémantique et extensible de la validité et justifie l'utilisation des spécifications de Creusot pour les bibliothèques implémentées en utilisant du code non sécurisé. Nous discutons de la connexion entre Creusot et RustHornBelt dans § 5.5.

Chapter 6 montre comment en utilisant Creusot, nous pouvons fournir une spécification puissante et flexible pour le trait `Iterator`, qui est à la base des boucles `for` de Rust. La contribution significative de ce chapitre est de montrer comment cette spécification peut gérer des effets secondaires complexes comme l'itérateur `IterMut`, qui renvoie des pointeurs mutables vers un conteneur. Le chapitre se termine en présentant une preuve de correction pour l'itérateur `Map`, qui applique une fermeture aux éléments d'un autre itérateur ; cette fermeture peut capturer à la fois des éléments mutables et effectuer des effets secondaires.

[1] https://github.com/xldenis/github

Enfin, dans Chapter 7, nous présentons une preuve d'un solveur SMT écrit en Rust. Ce solveur met en œuvre l'algorithme SMT de pointe CDSAT [15], que nous décrivons et formalisons dans le cadre du travail de vérification. Bien que ce solveur soit inefficace, l'architecture globale de la preuve ouvre la voie à une vérification plus complète et performante d'un solveur SMT. L'évaluation pour cette preuve comprend un *rapport d'expérience* décrivant l'expérience subjective de l'utilisation de Creusot pour vérifier un programme Rust étendu.

Acknowledgements

When I started my thesis in 2020, I had no idea I would not meet my advisors for the first six months nor be allowed into my lab for the first year due to the pandemic. These initial solitary months were challenging, but thanks to my advisors, they were made bearable.

Thus, first, I would like to thank my advisors Jacques-Henri Jourdan and Claude Marché, for their invaluable support and guidance, especially considering the challenges of supervising during a pandemic. Jacques-Henri's unrelenting attention to detail and deep insights into programming language semantics made me a much more rigorous researcher and forceed me to develop my skill as a writer of proofs. Additionally, Jacques-Henri was always available and ready to discuss any problem. We often hear about the perils of advisors who ask their students to work at all hours of the day; perhaps we should talk more about advisees who torture their supervisors at 2 AM. Thank you, Jacques-Henri, for answering my many late-night questions, your patience, and our many discussions on Framateam. Claude provided the perspectives and balances needed to progress with my PhD and always seemed to have a perfect paper to tackle any new issue. Claude supervised my work with equanimity, counteracting my anxious tendencies. I would also like to thank him for his "Proofs of Programs" course at the MPRI without which I may never have begun this PhD.

Further, I would like to thank my referees, Sylvain Boulmé and Peter Müller for reading my thesis and providing me with valuable feedback to highlight the results of these past three years. I would also like to thank the rest of my fantastic jury, who took the time to read my thesis and attend my defense: Ralf Jung, Francois Pottier, and Mihaela Sighireanu. My defense will always be a fond memory, and I am grateful to you all for your role in making it so.

I had the incredible luck of being a member of the TOCCATA team at the then-newly formed LMF, a group of fascinating people. Our discussions at lunch and the ensuing coffee breaks were among the highlights of my time there, whether discussing pastries, woodworking, or the latest math challenge Jean-Christophe had cooked up for us. Thank you to Andrei, Jean-Christophe, Guillaume, Sylvie, Kim, and Armael. A research group is incomplete without its students, post-docs, and engineers. Thank you to all I had the pleasure of sharing my time with, among others, Houda, Clément, Josué, Paul, Paul, Matteo. Perhaps my favorite parts of my PhD were the opportunities to collaborate with fascinating and brilliant people, so I would like to thank my co-authors Yusuke Matsushita and Derek Dreyer.

An unexpected development of the thesis was that it attracted the attention of terrific students who wanted to base *their* work on my highly unfinished, experimental project. Particularly, Sarek who wrote his master thesis using a tool that could not even handle a 'Hello World' program. His energy and willingness to play the guinea pig for my experiments was an invaluable help to my thesis. Similarly, I would like to thank Johannes and David, who used Creusot as a basis for their work and provided valuable feedback. Finally, thank you to Dominik, who came to spend four months with Jacques-Henri and me in Paris. Who was the 'simple' task of developing type invariants in Creusot, "It should only take two weeks," we said until Dominik unearthed the many finer points of the problem. Thank you all for being willing to put your diplomas on my work. I

feel honored for the trust and responsibility you placed in me.

I had the pleasure of organizing the Rust Formal Methods Interest Group during my thesis, and I would like to thank all the participants for their enthusiasm and the many exciting discussions we had. Through the Rust formal methods community, I met many people, some of whom I can now consider my friends: Sacha, Andrea, Vytautas, Frederico, Nico, and many others; thank you all.

Taking a small step away from academics, thank you to the "Verif' Squad" Denis, Son and Aymeric for the evenings and drinks philosophizing about the nature of verification, computing and the world's future. I still remember the first time Denis invited me to have lunch outside INRIA Paris where he encouraged me to start my PhD and introduced me to the rest of the squad. Denis remains an inspiration for his humanistic approach to computer science, always seeking to apply it for the betterment of all society. Somehow, after all his brilliant research, he finds time to read and write, and I hope to be half as good a writer as he is one day. Son has always been someone I admire, an incredibly brilliant and prolific researcher who is always focused on the most important problems. I hope one day be able to collaborate with him.

I want to thank my friends from the MPRI, who were there from the start and with whom I had the 'misfortune' of starting our PhDs in the middle of a pandemic: Pablo and Ada. Thank you also to Pablo for introducing me to the ReFL, and its members: Boris, Tito, Valentin, Sidney, Davide, Jérémy, Kostia, and the others.

A thesis is a marathon; reaching the end requires many people's encouragement. Some of my friends, some more recent and some less so, have provided me with that support. I want to distinguish Bouke, who has traveled with me since our meeting in Mexico and for the past *8 years* has been forced to listen to my *endless* rants on so many topics. By accepting this burden, you have spared many others. You and Ellen have always provided a reprieve from the various anxieties of academia. So, where are we going for our next trip?

Thank you to Camille, Erik, Mattie, Lauren, Esther, Meg, and Ian. My evenings would have been much less enjoyable without you, and I look forward to sharing many more now that I'm freed of this manuscript.

My PhD would have been entirely impossible without the help and support of my family, who have been forced to sit through innumerable and interminable explanations of my work. I hope to repay you for all my absences and half-presences soon. To Syd, Daphnée, and Milena, I am so glad that doing my Ph.D. in Paris allowed me to spend more time with you, and I hope to continue to do so. Those who know me also know that my family is huge and very important to me, I will spare you the list of names, but know that I am individually grateful to every one of you. However, I would like to thank my aunt Isabelle, with whom I lived upon arriving in Paris and who has always been present and supportive of me.

I would not be here without my parents, who have always nurtured my love of learning and knowledge. Their eternal support and encouragement (and occasional home-cooked meal) made this possible, so thank you to my mother and father. The same applies to my siblings: Agathe, Étienne, and Antoine.

Finally, I would like to thank Natasha, who brings me joy every day. She has put up with evening after evening of late work and frustrations and agreed to apply her much greater literary talents to the editing of my more pedestrian academic treatise. I'm lucky to have you by my side.

Contents

Chapter 1

Introduction

In 1843, Ada Lovelace published her notes on the Analytical Engine; in those notes, she corrected a mistake in the program for calculating Bernoulli numbers, the first software bug. Almost a century before the first computer ever ran, programmers were locked in struggle with their nemesis: their own programs.

Fast forward a century, and the father of modern computer science, Alan Turing, proposes a solution: apply mathematics and logic to formally demonstrate the correctness of programs. His 1949 paper 'Checking a large routine' [22] exemplifies this approach through a 'large' function for calculating the factorial of a number n. Yet, by the time of Turing's paper, programs were already thousands of lines long, and his example was already a toy. In that juxtaposition is found the essential tension of program verification: *the programs we want to verify are too large and complex to verify.*

Like Tantalus reaching for the fruit that always eludes him, researchers have striven to expand the scope of program verification to encompass the programs of their day. The works of Floyd [37], Hoare [44], Dijkstra [29] and others transformed the fastidious manual calculations of Turing into a systematic and automated process. Yet, each time the scope of program verification expanded, the day's programs had already outgrown it.

Not everything is hopeless, however. Part of the reason programs have been able to grow so large is due to the development of better, safer programming languages. From machine code to assembler to FORTRAN to C, Java, and Haskell, languages have developed ways to make it easier to write safe programs. At each stage of evolution, the invariants introduced by newer languages provide leverage for verification by reducing the number of possible states a program can be in. We can help the programmer and the verifier reach new heights through careful language design.

The advances in language development have not affected all programmers equally. The field of *systems programming*, which encompasses everything from firmware to operating systems is characterized by its heavy usage of languages like C and C++. These languages offer features crucial to systems programmers like low-level memory management and direct access to hardware. However, these features come at the cost of extreme unsafety. Microsoft estimates that 70% of their security vulnerabilities are due to *memory safety issues* [1], a class of bugs that is pervasive in C and C++. Despite providing a perfect opportunity for program verification to make a meaningful impact on the safety of software, systems software has stayed out of reach: the programs are too large and their memory behavior too complex to verify.

The solution came in 2015 with the 1.0 release of the Rust [82] programming language. Rust is a *safe* systems programming language, which, despite providing the low-level control of memory expected by *systems programmers*, makes the memory safety bugs of C/C++ impossible to write.

This feat is achieved through a powerful *ownership type system*, which enables the compiler to statically verify the memory behavior of programs. Borrows – the safe pointers of Rust – are tracked by the compiler, which ensures that they always point to initialized memory and that *mutable borrows* do not alias with other borrows. Ensuring the non-aliasing of mutable borrows makes programs more predictable by avoiding 'spooky action at a distance' in which parts of a program can affect each other unexpectedly.

A safer language can alleviate the burden placed on verification, enabling it to reach greater scale and complexity. The type system of Rust provides the ideal opportunity to finally achieve systems software verification.

1.1 The challenges of systems software verification

Systems software forms a prime candidate for software verification: it encompasses software whose failure could lead to loss of life or massive property damage. Though not universally so, systems software is often relatively stable, undergoing few changes (thus needing comparatively less re-verification). However, verified systems software projects are few and far between.

A reason for this can be found in the widespread usage of *pointers* in systems software languages. Pointers are essential for efficiently implementing data structures and are used to communicate and share data between software components. However, pointers also introduce many new ways for bugs to creep into software. Memory safety bugs in which *uninitialized memory* is accessed are a common source of crashes and security vulnerabilities. Historically, they have also been challenging to eliminate because pointers can go to arbitrary destinations in the heap. Reasoning about memory safety required untangling the webs of memory indirection and forced reasoning about the entire heap at once. Without first establishing memory safety, it is impossible to prove further the correct properties of software, as we cannot determine their behavior if they are unsafe.

In 2002, the introduction of *separation logic* [76] provided a theoretical solution to this problem. Separation logic is constructed around the idea of *ownership*; individual variables can exclusively own the memory that backs them. This ownership makes it possible to *separate* the heap into independent chunks that cannot affect each other. Separation logic allows recasting the global property of memory safety into a local one; each variable can be reasoned about individually. Despite providing the correct vocabulary for expressing the safety of C-like programs, that very safety remains *fiendishly* challenging to state. When combined with various forms of *aliasing* or *sharing* in which multiple parts of a program may modify a common data structure, stating the specification in separation logic can introduce a zoo of new mathematical terminology.

This leads to an observation certain readers of this thesis may find objectionable: *separation logic is too hard to read and write*. The relative paucity of separation logic literate people, especially compared to the number of systems programmers, places a natural upper limit on the impact of separation logic on verified systems software. From this, we can glean an initial requirement: a tool for the verification systems software must have *concise and readable specifications*. The overhead of specifying code should ideally be less than one line of specification per line of code, and more subjectively, the vocabulary of the specifications should be *unsurprising* and require (relatively) little specialized training to understand.

A second major contributing factor to the paucity of verified software is the economic cost of verification, which we can quantify by considering the invested workforce effort. The seL4 [55] kernel, one of the flagship accomplishments of formal verification, estimated the cost of verification at 11 person-years compared to approximately 2.2 person-years for the writing of the kernel itself. This materialized itself in roughly 200,000 lines of proofs verifying the correctness of 8,700 lines of C code. If formal verification is to find purchase in systems software, the benefit it brings must

be commensurate to its cost. The high proof burden has other second-order effects; when a proof takes a long time to write and verify, experimentation becomes less frequent as validating changes takes longer. This leads to a second requirement for verification: proof effort should be low, and proofs should be verified rapidly to allow faster incremental iteration and evolution of code and verification.

1.2 Verifying Rust programs

The requirements set above are nothing new, and of course, no one sets out to create a slow tool with illegible specifications and a proof burden that would make the most Kafkaesque bureaucrat blush. These tools are developed in reaction to the existing systems software and the languages they are written in. The flexibility and unreliability of the C language have forced verifiers into this position.

Rust promises to eliminate the safety issues of C through its strong, static type system while preserving the *useful* flexibility. The *borrow checker* of Rust ensures that *borrows* (safe pointers) are guaranteed to point to initialized memory, and that mutable borrows are unique. In C, establishing these properties would require significant specification and verification work; in Rust, they are provided out of the gate. If a verifier can trust and *leverage* the borrow checker, it could avoid the need to prove memory safety entirely. The uniqueness of mutable borrows opens up the possibility that a Rust verifier could avoid separation logic *entirely*, as they already guarantee the locality of reasoning. A Rust verifier would benefit from the more powerful automation and familiarity of classical logic by avoiding separation logic.

This possibility is complicated by the presence of `unsafe` Rust. The reality of systems programming occasionally requires going beyond what Rust can statically verify. Unsafe code allows a programmer to lift restrictions of the Rust type system and is an essential component for the core libraries of Rust.

If Rust is to change the fate of systems program verification, a tool must reconcile the safe and unsafe. It should leverage the safety guarantees of Rust to provide efficient verification of safe code, while accounting for the presence of `unsafe` code. The tool, Creusot, is a complete verification environment for safe Rust with strong theoretical foundations that allow soundly integrating unsafe libraries. Creusot supports the verification of *real world* Rust programs, supporting the myriad of complex, and essential features of Rust like *traits*, *closures*, and *iterators*. The design of Creusot is centered around a *type-driven*, *compositional* approach to Rust. We believe that this approach produces a more *predictable* tool; when something is possible in one situation, it is usable *everywhere* in Creusot. Leveraging the type-system of Rust allows Creusot to provide powerful proof automation, shortening the iteration cycles of verification.

1.3 Contributions

The rest of this thesis describes the design, implementation, and usage of the Creusot verifier. As of this thesis, Creusot can verify *safe* Rust code using the majority of available language features. Creusot is distributed as free software under an LGPL license and can be found online[1]. A short overview of key concepts needed throughout the thesis can be found in Chapter 2.

We start with dessert in Chapter 3 by showing how Creusot can be used today to verify a simple Rust program. We introduce Creusot from a user's perspective, showing the various functionalities available for specification and verification of Rust programs.

[1] https://github.com/xldenis/github

In Chapter 4 we present a complete description of Creusot's approach to verification conditions generation. We try not to simplify the translation but to describe precisely what is implemented. This translation is the core of Creusot and is the most significant contribution of this thesis. We also present the challenges associated with various advanced features of Creusot like *logical reborrowing* and the implementation issues encountered when dealing with Rust's *trait system* and *closures*.

In Chapter 5 we discuss the formalization of a core portion of Creusot's translation through the RustHornBelt project. This work extends the prior RustBelt project to reason about the *correctness* of Rust programs when equipped with with specifications. RustHornBelt provides a semantic, extensible proof of soundness and justifies the usage of Creusot specifications for libraries implemented using unsafe code. We discuss the connection between Creusot and RustHornBelt in § 5.5.

Chapter 6 shows how using Creusot we can provide a powerful and flexible specification for the `Iterator` trait, which is the basis for Rust's `for` loops. The significant contribution of this chapter is showing how this specification can handle complex side-effects like the `IterMut` iterator, which returns mutable pointers into a container. The chapter concludes by presenting a proof of correctness for the `Map` iterator, which applies a closure to the elements of another iterator; this closure can have both mutable captures and perform side effects.

Finally, in Chapter 7 we present a proof of an SMT solver written in Rust. This solver implements the state-of-the-art SMT algorithm CDSAT [15], which we describe and formalize as part of the verification work. While this solver is inefficient, the overall proof architecture opens the door to a more complete and performant verification of an SMT solver. The evaluation for this proof includes an *experience report* describing the subjective experience of using Creusot to verify an extensive Rust program.

Publications The preparation of this thesis included the publication of several articles in peer-reviewed conferences and journals.

- "The Creusot Environment for the Deductive Verification of Rust programs" [27], appeared at ICFEM 2022, and introduced Creusot.
- "RustHornBelt: A Semantic Foundation for Functional Verification of Rust Programs with Unsafe Code" [70], appeared at PLDI'22 and won Distinguished Paper. This work forms the basis for Chapter 5.
- "Specifying and Verifying Higher-Order Rust Iterators" [26], appeared at TACAS'23. The work in Chapter 6 is a significantly revisited and expanded version of this article.

Chapter 2

Background Materials on Program Verification, Rust and Prophecies

Before introducing how we can verify Rust programs, we will provide an overview of both *program verification* and *Rust* independently. The remainder of this chapter will be structured as follows: §2.1 will briefly overview Hoare logic, the weakest precondition calculus, and predicate transformers. § 2.2 introduces Why3, the program verification framework we will use later in this thesis. § 2.3 introduces the Rust programming language and its ownership system. Finally, § 2.4 presents an overview of *prophetic verification*, the technique we will use to verify Rust programs.

2.1 Program Verification

The field of program verification is concerned with showing that programs are *correct* for a specification, often with the aid of a computer. Many different kinds of *specifications* and techniques can be used to verify them. We concern ourselves exclusively with the *functional correctness* of programs: establishing that a program produces the correct outputs for all possible inputs.

From a technique standpoint, we will focus on *deductive verification*. In deductive verification, we interpret the correction of a program as a *logical statement*, and we then construct a *proof* of that statement's validity. In practice, we construct dedicated logics for reasoning about individual languages and use (semi) automated tools to discharge the obligations. We will follow the classic route by introducing a simple imperative language and then exploring how to verify programs in that language.

Definition 1 (IMP)**.** *The following grammar defines the IMP language:*

$$s \ni \mathit{Stmt} ::= x \leftarrow e \mid s_1; s_2 \mid \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi} \mid \texttt{while}\ e_1\ \texttt{do}\ s\ \texttt{done} \mid \texttt{skip} \mid \texttt{panic}$$
$$e \ni \mathit{Exp} ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1/e_2 \mid e_1 \leq e_2 \mid e_1 = e_2 \mid \texttt{true} \mid \texttt{false}$$

where x is a variable and n is an integer. The following big-step semantics are given to expressions:

$$[\![x]\!]_\Sigma = \Sigma(x) \qquad [\![n]\!]_\Sigma = n \qquad [\![e_1 + e_2]\!]_\Sigma = [\![e_1]\!]_\Sigma + [\![e_2]\!]_\Sigma \qquad [\![e_1 - e_2]\!]_\Sigma = [\![e_1]\!]_\Sigma - [\![e_2]\!]_\Sigma$$
$$[\![e_1 * e_2]\!]_\Sigma = [\![e_1]\!]_\Sigma * [\![e_2]\!]_\Sigma \qquad [\![e_1/e_2]\!]_\Sigma = [\![e_1]\!]_\Sigma / [\![e_2]\!]_\Sigma \qquad [\![e_1 \leq e_2]\!]_\Sigma = [\![e_1]\!]_\Sigma \leq [\![e_2]\!]_\Sigma$$
$$[\![e_1 = e_2]\!]_\Sigma = [\![e_1]\!]_\Sigma = [\![e_2]\!]_\Sigma \qquad [\![\texttt{true}]\!]_\Sigma = \texttt{true} \qquad [\![\texttt{false}]\!]_\Sigma = \texttt{false}$$

while statements are given a small-step semantics, where the state Σ is a mapping from variables to either booleans or integers.

$$\langle \Sigma \mid x \leftarrow e \rangle \rightsquigarrow \langle \Sigma[x \mapsto [\![e]\!]_\Sigma] \mid \texttt{skip} \rangle \qquad \langle \Sigma \mid \texttt{skip}; s \rangle \rightsquigarrow \langle \Sigma \mid s \rangle \qquad \frac{\langle \Sigma \mid s_1 \rangle \rightsquigarrow \langle \Sigma' \mid s_1' \rangle}{\langle \Sigma \mid s_1; s_2 \rangle \rightsquigarrow \langle \Sigma' \mid s_1'; s_2 \rangle}$$

$$\frac{[\![e]\!]_\Sigma = \texttt{true}}{\langle \Sigma \mid \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi} \rangle \rightsquigarrow \langle \Sigma \mid s_1 \rangle} \qquad \frac{[\![e]\!]_\Sigma = \texttt{false}}{\langle \Sigma \mid \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi} \rangle \rightsquigarrow \langle \Sigma \mid s_2 \rangle}$$

$$\frac{[\![e]\!]_\Sigma = \texttt{true}}{\langle \Sigma \mid \texttt{while } e \texttt{ do } s \texttt{ done} \rangle \rightsquigarrow \langle \Sigma \mid s; \texttt{while } e \texttt{ do } s \texttt{ done} \rangle} \qquad \frac{[\![e]\!]_\Sigma = \texttt{false}}{\langle \Sigma \mid \texttt{while } e \texttt{ do } s \texttt{ done} \rangle \rightsquigarrow \langle \Sigma \mid \texttt{skip} \rangle}$$

The IMP language is a simple imperative language with variables, integers, and arithmetic expressions. It has the usual control flow constructs: assignment, sequencing, conditionals, and while loops. We have also added a `panic` statement representing a program that crashes. Our objective will be to prove that panic does not occur, so do not give it a step, causing evaluation to get stuck.

We use boolean-typed expressions as our *assertion language* to write our specifications. Let P be an assertion, we say that a state Σ satisfies P (denoted $\Sigma \models P$) if $[\![P]\!]_\Sigma = \texttt{true}$.

2.1.1 Hoare Logic

There are many varying grades of safety properties we can establish for our programs. The simplest one is ensuring *panic safety*, that is demonstrating that a `panic` never occurs. This may require establishing *functional correctness* which establishes the input-output relation for a program. We may be unable to correctly execute our program for every possible input state, like in the program `if x` $\neq$ `0 then x` $\leftarrow$ `10 / x else panic fi`, which will panic if $x = 0$. To handle this, we consider the correctness of a program relative to a *precondition* P and *postcondition* Q. A program is correct if, given a state satisfying P, it will produce a state satisfying Q.

Definition 2 (Hoare Triples)**.** *A* Hoare Triple $\{P\}\, s\, \{Q\}$ *is defined as follows:*

$$[\![\{P\}\, s\, \{Q\}]\!] = \forall\, \Sigma\ \Sigma',\ \langle \Sigma \mid s \rangle \rightsquigarrow^* \langle \Sigma' \mid \texttt{skip} \rangle \rightarrow \Sigma \models P \rightarrow \Sigma' \models Q.$$

To prove the validity of such a triple, we can construct a *derivation* of the triple using a *Hoare logic* composed of atomic rules for each statement in our language.

Definition 3 (Axiomatic Semantics of IMP)**.** *The following rules define the axiomatic semantics of IMP:*

$$\{P\}\, \texttt{skip}\, \{P\} \qquad \{\texttt{false}\}\, \texttt{panic}\, \{Q\} \qquad \{Q[x \mapsto e]\}\, x \leftarrow e\, \{Q\}$$

$$\frac{\{P\}\, s_1\, \{Q\} \qquad \{Q\}\, s_2\, \{R\}}{\{P\}\, s_1; s_2\, \{R\}} \qquad \frac{\{P \wedge e\}\, s_1\, \{Q\} \qquad \{P \wedge \neg e\}\, s_2\, \{Q\}}{\{P\}\, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi}\, \{Q\}}$$

$$\frac{\{P \wedge e\}\, s\, \{P\}}{\{P\}\, \texttt{while } e \texttt{ do } s \texttt{ done}\, \{P \wedge \neg e\}} \qquad \frac{P \rightarrow P' \qquad \{P'\}\, s\, \{Q'\} \qquad Q' \rightarrow Q}{\{P\}\, s\, \{Q\}}$$

The rules of Definition 3 cover each syntactic case of IMP, with one extra rule for *consequence*, which allows us to strengthen the precondition and weaken the postcondition of a triple.

Using these rules, we can prove the correctness of simple programs:

$$\texttt{if } x \neq 0 \texttt{ then } x \leftarrow 10/x \texttt{ else panic fi}$$

Specifically, we want to show that the following triple holds:

$$\{x \neq 0\}\,\texttt{if } x \neq 0 \texttt{ then } x \leftarrow 10/x \texttt{ else panic fi}\,\{x = 10/x\}$$

which we can do by applying the If rule, and then the Assign rule:

$$\text{IF}\ \frac{\text{ASGN}\ \dfrac{}{\{x \neq 0 \wedge x \neq 0\}\ x \leftarrow 10/x\,\{x = 10/x\}} \qquad \dfrac{}{\{x \neq 0 \wedge x = 0\}\,\texttt{panic}\,\{x = 10/x\}}\ \text{PANIC}}{\{x \neq 0\}\,\texttt{if } x \neq 0 \texttt{ then } x \leftarrow 10/x \texttt{ else panic fi}\,\{x = 10/x\}}$$

The derivation above establishes the validity of the triple, but apart from intuition, nothing relates this to the *semantics* of the program. The missing property is a proof that our axioms are *sound*, that if a triple $\{P\}\,s\,\{Q\}$ is provable, then given an input state satisfying P, all states produced by executing s satisfy Q. This is the usual definition of *partial correctness*; it does not require the existence of the output state s; the program could diverge instead.

With this notion of semantics for our triples, we can state the soundness theorem for our Hoare logic:

Theorem 2.1.1 (Soundness of IMP Logic)**.** *If $\{P\}\,s\,\{Q\}$ is provable, then $[\![\{P\}\,s\,\{Q\}]\!]$ holds.*

We do not go over the proof of this theorem, but a version can be found in Software Foundations [78]. Often, we may also prove the converse *completeness* theorem, which states that if $[\![\{P\}\,s\,\{Q\}]\!]$ then $\{P\}\,s\,\{Q\}$ is provable.

2.1.2 Predicate Transformers

Though we have constructed a sound (and complete) logic for IMP programs, how do we know it is the '*best*' one? There are many possible sets of axioms for a language, but generally, we are interested in the most flexible and least constraining ones. One way to achieve this is to find the *weakest* precondition for a program. The *Weakest Precondition Calculus*, written $\text{wp}(s, Q)$ takes as input a program s and a postcondition Q and transforms it into a *precondition* for s.

For example, we can easily define wp for simple commands:

$$\text{wp}(\texttt{skip}, Q) \triangleq Q$$
$$\text{wp}(\texttt{panic}, Q) \triangleq \texttt{false}$$

while more complex definitions like `while` are more complex and cannot be given a functional form in first-order logic. Instead, `while` requires the introduction of an *invariant* which must be user-provided.

The WP calculus is one example of the more general technique of *predicate transformers*. An instance of $\text{wp}(s, Q)$ transforms the postcondition Q into a precondition by examining the program s. Other possible systems exist, such as the dual *strongest postcondition* calculus.

2.2 The Why3 verification environment

Why3 applies the ideas for deductive verification we have discussed until now to a more complete and realistic language. It is designed both as an intermediate language for other verification tools and as a tool to directly write verified programs in. Internally, Why3 uses a mix of weakest-precondition

```
let divide (x : int)
  requires {x <> 0}
  ensures {result = 10 / x}
  = 10 / x
```

(a) A simple WhyML function

```
Sub-goal
  Check division by zero of goal divide'vc.
Prover result is: Valid (0.01s, 497 steps).

Sub-goal
  Postcondition of goal divide'vc.
Prover result is: Valid (0.01s, 889 steps).
```

(b) Output from `why3 prove`

Figure 2.1: A simple WhyML function and its proof using Why3

and strongest-postondition calculi to generate the verification conditions for a program. These can then be discharged with the help of automated solvers like *satisfiability modulo theory* (SMT) solvers or other Automated Theorem Provers (ATP). In Figure 2.1 run the program on the left-hand side through Why3 to obtain the result on the right-hand side.

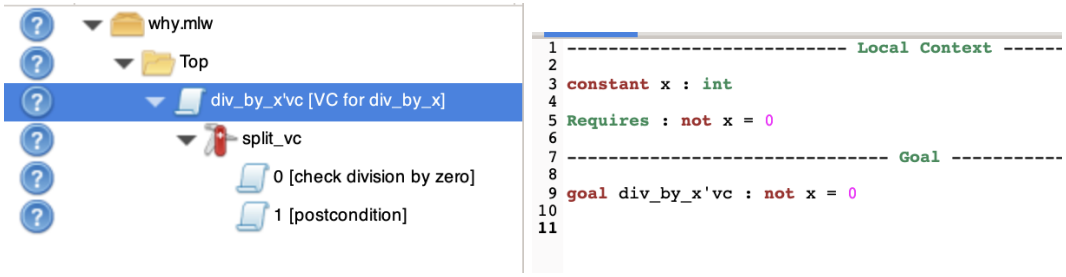
Why3 has several features that clearly distinguish it from its contemporaries (Dafny [60], Viper [74], Boogie [8]). While most tools of this class are developed around one specific solver backend, usually the Z3 [73] SMT solver, Why3 is foundationally based on the opposite notion. Why3 currently supports 15 or more different backend solvers including major SMT solvers (Z3, CVC4 [10], CVC5 [6], Alt-Ergo [23]), first-order and higher-order logic solvers (Eprover [84], Vampire [57]) and interactive theorem provers (ITP) like Coq [24] and Isabelle [77]. This is possible through a *driver* mechanism that allows quickly adding new provers. When performing proofs with Why3, it is common – expected even – to use multiple provers simultaneously. Combining provers in a 'portfolio' allows them to compensate for each other's weaknesses and provide a more consistent overall experience.

The second significant difference when compared to similar tools is Why3's interface. Why3 is most directly comparable to tools like Dafny or Viper, which facilitate 'auto-active' verification. These tools allow users to write programs and assertions and attempt to verify them in the background, providing a binary 'yes/no' with the verification results. In this interaction model, a user's primary tool is the assertions they write; if a proof fails, they must write a series of assertions, which should entail the failing assertion. Why3 takes a different perspective on interaction, similar to ITPs like Coq. Individual proof obligations are visible in a 'Task Tree' (Figure 2.2a) where users can apply *transformations*, which perform various simplifications and steps of logical deduction. Each proof obligation can also be displayed as a 'Task' (Figure 2.2a) showing the hypotheses and required conclusion.

While sometimes awkward to use, tasks and transformation add much flexibility to Why3 and help place users in control of the automation; they have the agency to interrupt, explore, and guide the automated solvers.

2.2.1 WhyML

Why3 is built around the WhyML language, the primary input format. WhyML is an ML-family language designed for program verification; it includes several concepts that serve only to facilitate verification. As we will use WhyML later in this thesis (Chapter 4), we will briefly introduce some of the more relevant features now. At the highest level, it supports the usual features of a functional language: algebraic data types, pattern matching, and polymorphism. Though it does not fully support higher-order functions, they are limited to *pure* functions, which do not have any side



(a) Why3 task tree and transformation (b) Why3 Task view

Figure 2.2: The Why3 IDE GUI

effects. Why3 has a *region typing system* [40], which enables the usage of *mutable fields*, so long as they are not recursive and are not aliases. Mutable fields allow defining types like OCaml's arrays and references.

To support verification, WhyML functions can be given *contracts* through `requires` and `ensures` clauses; we showed an example of these in Figure 2.1a. Additionally, loops can be given `invariant` clauses to express the invariants in the sense of §2.1. Types can also be given invariants, a property that must be maintained by values at function entrance and exit.

```
1   type positive = { x : int } invariant { x > 0}
2
3   let dec (a : positive) : positive
4     requires { a.x > 1 }
5     ensures { result.x < a.x }
6     = { x = a.x - 1 }
```

We may need to use auxiliary logical definitions in contracts; these can be defined using the `function` and `predicate` keywords. These definitions must be *pure* and can be used in contracts and other logical definitions. Interestingly, these definitions are *total*, even things like division-by-zero are allowed; such operations return an *unspecified* value instead.

```
1   function divides0 (x : int) : int = x / 0
```

We can use *ghost code* to help guide the proof when proving code, especially loop invariants. Ghost code is only visible to the verifier and does not affect the execution of the program. It allows the user to spell out steps of reasoning which are implicitly happening in the program and help the solvers find a proof.

Modules and Cloning

Code and proof organization in WhyML is handled using *modules* [35]. A module contains definitions and declarations, including types, functions, and predicates. Modules can include *declarations*: definitions without a body.

```
1   module Sort
2     type t
3     function le (x y : t) : bool
4     axiom le_trans : forall x y z. le x y -> le y z -> le x z
5     axiom le_refl : forall x. le x x
6     predicate sorted (a : array t) =
7       forall i j. 0 <= i <= j < length a -> le a[i] a[j]
```

```
8
9       let sort (a : array t) : unit
10        ensures { sorted a }
11        = { ... }
12    end
```

The module `Sort` above defines a generic procedure for sorting arrays. It is parameterized with the type of elements `t` (line 2) and the comparison function `le` (line 3). The module assumes two axioms about the provided comparison function, and using those can define a procedure `sort`, which sorts an array in place and ensures that the array is sorted.

A module can be used through *cloning*, which copies the contents of a module into the current environment. A clone can apply a *substitution*, instantiating some or all declarations in the cloned module:

```
1    clone Sort with type t = int, function le = (<=)
```

During this clone, axioms are turned into proof obligations, and the user must provide proof of them. This mechanism allows for high-level modularity in proofs as work can be split into different modules and then composed together. It is important to note that cloning is *generative*: performing the exact clone twice will introduce duplicate definitions, declarations, and types, which Why3 cannot understand as identical. Cloning is a fully manual process: Why3 cannot automatically determine what to clone and what substitutions to apply.

2.3 Rust

First publicly released in 2015, Rust is a systems programming language that provides the same control as C or C++ but guarantees memory safety without introducing a runtime or garbage collector. Rust combines a novel *ownership* type system with strong encapsulation. The ownership system ensures each value has a single, exclusive owner at any time and thus can free memory when that owner is dropped. Seemingly restrictive, the ownership discipline is often unobtrusive. However, sometimes it becomes a problem, like when handling cyclic data. For these cases, Rust provides *interior mutability*, types that allow various forms of aliased mutable states like mutexes.

Interior mutable types are implemented using Rust's *unsafe* features that allow bypassing the ownership system. However, this does not give library authors carte blanche; using unsafe code to implement a *safe* API comes with responsibilities: no undefined behavior (UB) can be exposed to the user. Despite being partially a social convention, this feature is perhaps the most critical driver to Rust's success; unlike its competitors, C/C++, there is no 'wrong way to hold the stick.' Rust provides a strong nominative type system to help the authors of unsafe code uphold their promise of UB-free interfaces. By encapsulating their unsafe code carefully, authors can ensure that clients will always maintain the invariants of their types.

Below, we show a simple example of Rust code, which we will use to introduce some of the language's features.

```
1    enum X { A(usize), B { x : bool }, C }
2
3    fn match_X(x: X) {
4      let mut a = 0;
5      match x {
6        X::A(_) => a = 1,
7        X::B { .. } => a = 2,
8        X::C => a = 3,
9      }
```

```
10      println!("{}", a);
11    }
```

As shown in the code above, Rust has an ALGOL-inspired syntax similar to that of C-family languages. However, beyond superficialities, Rust has many syntactic features usually found in functional languages. On line 1, we declare a new *enum* or tagged union, `X` with three variants, `A`, `B`, and `C`, which may contain fields. We can also declare `struct` types containing multiple possibly named fields. On line 3, we declare a function `match_X`, which takes a value of type `X` and prints a number depending on the variant. We declare a new local, mutable variable on line 4 with the syntax `let mut`. Bindings are immutable by default in Rust and have *inferred types*. On line 5, we exhaustively evaluate the possible values of `x` through a `match` expression, storing a different value into `a` as a function of `x`. The Rust compiler checks the exhaustiveness of `match` expressions, raising an error if a case is forgotten. Finally, on line 10, we call a *macro* to print the value stored in `a`. Rust includes an extensively used powerful macro system: even printing output goes through a macro.

2.3.1 Ownership

As mentioned earlier, Rust has a notion of *ownership*; each variable has exclusive control of its contents: no other alias can exist. This has immediate consequences on the semantics of common operations:

```
1   let mut x = Vec::new();
2   let y = x;
3   x.push(0); // ERROR: `x` has been moved into `y`
```

To preserve ownership, Rust uses *move* semantics, so when we assign `x` to `y`, we move the contents out of the name `x` and thus can no longer use `x`. This applies even to *functions*:

```
1  fn do_something<T>(mut v : Vec<T>) { v.push(4); }
2
3  fn main() {
4    let x = vec![1,2,3];
5    do_something(x);
6    assert_eq!(x[3], 4); // ERROR `x` has been moved
7                         // during call to `do_something`
8  }
```

On line 4, we create a vector containing the elements $1, 2, 3$ we then call `do_something` using it. From `main`, we can no longer observe that `do_something` appended a new element to `x` as we no longer own the value. In the parlance of imperative programming, Rust employs *pass-by-value*.

Certain types like integers can be declared as *copyable*, in which case the ownership discipline is lifted and, instead of moving assignments and function calls, will perform a byte-by-byte copy of the value:

```
1   let x = 1;
2   let y = x;
3   assert!(x == y);
```

Types with enjoyable ownership like `Vec` or `HashMap` cannot be made copyable as copies would share underlying memory, which could lead to double-free or other errors.

A consequence of this strict ownership discipline is that Rust types can only naturally express tree-like structures; anything more complex must rely on unsafe code or libraries like `Rc` (reference counted pointers, implemented with unsafe code).

Borrowing & Lifetimes

Though surprisingly expressive, as shown, the ownership disciple of Rust would be overly restrictive: it is not even possible to *just* read a value, and even operations like *indexing an array* are impossible to express. To solve this, Rust uses safe pointers called *borrows*, objects which temporarily *borrow* the ownership of a value, and after their *lifetime* ends restore that ownership to the lender. Borrows come in two flavors: immutable (`&`) and mutable (`&mut`), characterized by the idiom "Aliasing XOR Mutation" (AXM). At any time, we can *either* create a unique mutable borrow to a value, or we can create arbitrary amounts of immutable borrows:

```
1   let mut a = 0;
2   assert_eq!(&a, &a); // OK
3   let b = &mut a; // OK, previous borrows are ended
4   *b = 5;
5   assert_eq!(&mut a, &mut a) // ERROR
```

As the name implies, borrows are *temporary*; each has a *lifetime* by which they must be dropped so permissions can be restored. A borrower's lifetime must be shorter than the lifetime of the lender, thus ensuring it always points to initialized memory. Lifetimes are inferred by the Rust compiler, which also checks the *AXM* discipline.

Rust borrows are flexible; we can use them to create *interior pointers* to types:

```
1   struct Pair<A, B>{a : A, b: B }
2   ...
3
4   fn split(x : &mut Pair<A, B>) -> (&mut A, &mut B) {
5     (&mut x.a, &mut x.b)
6   }
```

On line 5, we *reborrow* our borrow `x` into disjoint borrows for each field of our type `Pair`.

We can give explicit lifetimes to borrows using the syntax `&'a mut T` (or `&'a T`) where `'a` is the lifetime of the borrow. Reborrowing can be used to shorten lifetimes:

```
1   fn shorten<'b, 'a : 'b, T>(a : &' a mut T) -> &'b mut T { &mut * a }
```

The syntax `'a : 'b` is an *outlives constraint*, it requires that `'a` is some lifetime which is *longer* than `'b`.

Interior Mutability

There are situations where even the flexibility of borrows is insufficient, where we need to mutate through a shared borrow. For example, consider a multi-threaded application using a *mutex* to synchronize shared values. Each thread must have a reference to the mutex, but then no thread can fully *own* the mutex and thus mutate the stored values. Instead, a mutex can be mutated with only a *shared borrow* as the mutex *dynamically* guarantees the uniqueness of mutable borrows: only one thread at a time can lock the mutex and mutate the shared value.

This behavior is called *interior mutability* and provides an escape hatch for the cases where the borrowing rules are too restrictive. Rust provides types for interior mutability: `Cell<T>` and `RefCell<T>`[1]. These types represent shared mutable memory and enforce the safety properties of Rust at runtime. The `Cell<T>` type allows you to store and read values of `copyable` types. The `get` function returns a copy of the value in the cell, and the API provides no mechanism to obtain a borrow. The `RefCell<T>` type is more flexible; it provides two methods `borrow` and `borrow_mut`,

[1] These are implemented using a *primtive* `UnsafeCell<T>` type, but we will ignore this.

which return immutable borrows or mutable borrows, respectively, subject to runtime checks. The program will panic if a mutable borrow is attempted while an immutable borrow is active (and vice-versa).

By combining `RefCell` with duplicable pointers like `Rc` (reference counted values), we can recover the mutable, aliasable memory of other languages, and use this to create cyclic data structures, ordinarily impossible in safe Rust:

```
1   struct Node {
2     next : Option<Rc<RefCell<Node>>>,
3   }
4
5   fn main() {
6     let a = Rc::new(RefCell::new(Node { next : None }));
7     let b = Rc::new(RefCell::new(Node { next : Some(a.clone()) }));
8     a.borrow_mut().next = Some(b.clone());
9     let a = (a.borrow_mut(), a.borrow_mut()); // ERROR: 'already borrowed'
10  }
```

Because line 11 has no active borrow of the cell in `a`, we can safely mutate it to point to `b`. In contrast, line 9 attempts to create two mutable borrows of `a`, and thus panics.

Despite the presence of this escape hatch, which would free programmers from the constraints of ownership, interior mutability is a niche (but essential) feature in Rust. It finds usage in synchronization primitives (like `Mutex<T>`), in asynchronous programming (again for synchronization), and in memoization (for caching expensive computations). The relatively limited usage of interior mutability is a testament to the flexibility of the ownership system and the power of the Rust type system.

2.3.2 Unsafe Code

Unsafe code is an often misunderstood feature of Rust. A common view is that `unsafe` is a keyword that gives programmers *carte blanche*, freeing them from the onerous restrictions of the type system. In fact, unsafe code is subject to the *same* typing and ownership rules as safe code, but it permits additional operations that cannot be *statically checked* by the compiler. The need for `unsafe` arises from the conservative nature of the Rust compiler: it is impossible to prove the safety of all programs statically.

When a library author uses `unsafe`, they pass a contract with the compiler to avoid the undefined behavior the compiler cannot statically eliminate and are granted permission to use additional operations. They are expected to package this unsafe code behind a *safe abstraction* with a safe API that respects the invariants of the unsafe code. Though this contract is not formally enforced, it delineates responsibility for bugs between library authors and users. If a Rust programmer encounters a segfault in safe code, the problem lies with the libraries used, not their code, no matter how torturous.

Unsafe code is declared through the `unsafe` keyword which appears in two principal places:

- **Unsafe blocks**: The most common usage of `unsafe` is to declare an *unsafe block*, which allows performing unsafe operations in a safe function. Of particular importance, it is possible to use *raw pointers* in unsafe blocks, which are not tracked for ownership by the type system.
- **Unsafe functions**: A function may be marked `unsafe`, which indicates that the caller must uphold additional invariants. Calling an unsafe function is always an unsafe operation. It is expected that unsafe functions are accompanied by a `SAFETY` comment explaining the additional invariants expected of the caller.

In all of these cases, the authors of unsafe code must keep in mind Rust's *undefined behaviors* so they may successfully avoid them. This task is complicated by the lack of a comprehensive list of undefined behaviors, though it is actively being worked on [88]. Though Rust's UB is still in flux, the Rust project provides Miri [81], an interpreter for Rust, which can detect some UB in unsafe code.

2.3.3 Traits

Industrial languages need to be able to abstract common behaviors, and Rust is no exception. Consider an equality function: We cannot write a single function that works for all types, as we must know how to compare the values. We also do not want to rely on an ad-hoc collection of monomorphic functions, as this would be inflexible and hard to extend. Instead, we use a single function `eq` for which the compiler can determine the appropriate implementation. This problem of *ad-hoc polymorphism* is solved in Rust through *traits*, also known as *typeclasses* [93] in Haskell. A trait associates a set of functions and types to a *subject* type. For example, the `Eq` trait is defined as follows:

```
trait Eq {
  fn eq(&self, other : &Self) -> bool;
}
```

The first argument of a trait function can use the special `self` name, designating that function as a `method`, which can be accessed through the dot-syntax `a.eq(..)`. The `self` parameter can come in three different forms: `self`, `&self`, and `&mut self` describing the different ways of referring to values in Rust. As the name implies, the `self` argument is of the same type as the trait subject. In a trait, the `Self` type refers to the subject of the trait or implementation.

To state that a type `T` implements `Eq`, we must provide an *implementation*. For example, we could provide an implementation for `bool`:

```
impl Eq for bool {
  fn eq(&self, other : &Self) -> bool {
    match (*self, *other) {
      (true, true) => true,
      (false, false) => true,
      _ => false
    }
  }
}
```

Instances do not need to be monomorphic; we can also provide implementations for generic types by using *constraints* to specify that the type parameter must implement specific traits, *e.g.,* on generic pairs:

```
impl<T : Eq, U : Eq> Eq for (T, U) {
  fn eq(&self, other : &Self) -> bool {
    self.0.eq(&other.0) && self.1.eq(&other.1)
  }
}
```

Then when, if we call `(true, false).eq(&(true, false))`, the solver will know how to construct an instance for `(bool, bool)` from the previous implementations. Constraints can also be added to ordinary functions, allowing us to write generic functions which can only be called on types that implement specific traits:

```
fn neq<T : Eq>(x : T, y : T) -> bool { !x.eq(&y) }
```

Coherence Because the mechanism to pick which trait implementation is out of the user's control, it must remain *predictable*, it would be problematic if equality for pairs of booleans meant two things in different contexts. This requirement extends to an *ecosystem* level, where libraries may provide traits and implementations. It is desirable that adding or removing a library does not change the behavior of unrelated code. To solve this, Rust uses a *coherence* rule; there must be at most one instance applicable to a type for any given trait. This rule is verified through an *overlaps check*, which ensures that no two implementations are *unifiable*. This would mean that we cannot also provide an implementation of `Eq` just for `(bool, bool)`.

2.3.4 Closures

Like most contemporary languages, Rust supports *closures*, functions that can capture variables from their environment. Closures are a powerful tool for abstraction and find frequent usage in Rust code bases. They can be used to transform, filter, or select from streams of data using `map`, `filter`, and `fold`, respectively, or used to encode callbacks. Rust closures are written using the syntax `|a,...,z| { ... }`, the braces are optional if the body is a simple expression.

For users of traditional functional programming languages, the closures of Rust can be a little disorienting. Unlike most other higher-order languages, Rust does not use a universal 'arrow' or 'function' type to describe closures. Instead, every closure has a unique, anonymous type, which users cannot write. This is a consequence of Rust's compilation and ownership models.[2]

Rust uses trait bounds to write higher-order functions that accept closure arguments. A function accepts a closure as an argument by accepting a generic parameter of type `F`, which implements an `Fn`-trait. There are three such traits `FnOnce`, `FnMut` and `Fn`, which each contain a single `call_*` method that invokes the closure function while accepting the environment by value, mutable and immutable borrow respectively. The `Fn` traits form a hierarchy: every `Fn` is an `FnMut`, and every `FnMut` is an `FnOnce`. The different steps in the `Fn`-hierarchy trade-off restrictions on the closure with flexibility in how it can be called.

For example, we can implement `Option::map` as follows:

```
impl<T, U> Option<T> {
  fn map<F : Fn(T) -> U>(self, f : F) -> Option<U> {
    match self {
      Some(x) => Some(f(x)),
      None => None
    }
  }
}
```

The `map` function above accepts a variable of type `F` which can be called as a function from `T` to `U` (i.e. `F` implements the `Fn`-trait). The expression `f(x)` is desugared by rust into `f.call(x)` invoking the method of the `Fn` trait. The syntax `Fn(T) -> U` is a sugar for the trait bound `Fn<T, Output = U>`, available in Rust to make closures more intuitive.

When we attempt to call `Option::map` with a closure, Rust will automatically provide the appropriate definitions of `Fn`, `FnMut`, or `FnOnce` as appropriate.

[2] Rust compiles closures via *closure conversion*, generating a struct for each closure holding its captured variables and a function for each closure method. Two closures with the same method signature can have entirely different captures and, thus, different sizes. Rust requires each type to have a single size, so it cannot use a universal 'arrow' type for closures.

```
fn maybe_mod(o : Option<u32>, n : &u32) -> Option<bool> {
  o.map(|x| x % n == 0) // OK
}
```

The closure above would generate an *environment struct* like the one below:

```
struct ClosureEnv<'a> {
  n : &'a u32
}
```

The closure function body will then determine how the environment needs to be passed, either by value, mutable borrow, or immutable borrow. In this case, as we only need to read the value of `n`, the closure can be called with an immutable borrow, making it `Fn`.

The `FnOnce` trait At the top of the hierarchy is the `FnOnce` trait. This trait accepts the closure environment by value, consuming it during the closure call. This trait is the most flexible for the closure as it can manipulate its environment however it wants. In contrast, it is the most restrictive for the client as the closure is consumed during the call. Below we define the `FnOnce` trait:

```
trait FnOnce<Args : Tuple> {
  type Output;
  extern fn call_once(self, args : Args) -> Self::Output;
}
```

It is parameterized with a type `Args`, which captures the function arguments of a closure as a tuple. The associated type `Output` is the return type of the closure. The single method `call_once` consumes the closure environment and returns the result of the closure call.

To understand what we can do with this, consider the example below:

```
fn ex_call_once<F : FnOnce(u32) -> bool>(f : F) -> bool {
  f.call_once(0)
}

fn client() {
  let v = Vec::new();
  ex_call_once(|x| {drop(v); true}) // OK
}
```

In our small `client` example, we drop a vector `v` during the closure call. Dropping requires full ownership over the value, thus forcing the closure to capture `v` by value. Since `drop` consumes `v`, calling it twice would be a use-after-free, so the compiler must ensure that the closure is only called once.

The `FnMut` trait Often, we do not need to consume the closure environment, but we still want to mutate it. For example, consider a closure, which counts the number of times it is called. We can implement this using the `FnMut` trait, which accepts the closure environment by mutable borrow.

```
trait FnMut<Args> : FnOnce<Args> {
  fn call_mut(&mut self, a : Args) -> Self::Output;
}
```

The trait inherits from `FnOnce`, the closure function itself has the same `Args` to `Output` signature as `FnOnce`, the only change is that `self` is passed behind a mutable borrow. In the following example, we use this to determine our closure was called twice:

```
fn ex_call_mut<F : FnMut(u32) -> bool>(mut f : &mut F) -> bool {
  f.call_mut(0); f.call_mut(0); f.call_mut(0)
}
fn client_mut() {
  let mut cnt = 0;
  ex_call_mut(&mut |x| {cnt += 1; true}) // OK
  assert_eq!(cnt, 3);
}
```

In this example, our closure captures `cnt` by mutable borrow and thus can be made `FnMut`: we do not need to consume `cnt` entirely.

Note, that since every instance of `FnMut` is also an instance of `FnOnce` we could also pass this closure to the prior `ex_call_once` function:

```
  fn client_mut_as_once() {
    let mut cnt = 0;
    let mut c = |x| {cnt += 1; true}
    ex_call_once(c) // OK
    // ex_call_once(c); // ERROR
    assert_eq!(cnt, 1);
  }
```

The call to `ex_call_once` consumes the closure; uncommenting the second call would be an error. This highlights a key detail of Rust closures: their kind (`FnOnce`, `FnMut`, `Fn`) does not determine how their captures are made, but only how the environment is handled during the call.

The `Fn` trait The final kind of closure is `Fn`, which accepts the closure environment by immutable borrow. An `Fn` closure places the most significant restrictions on the closure's environment but provides the caller the greatest flexibility. `Fn` closures are often used whenever a predicate-like function is needed, such as filtering or searching a collection.

```
  fn contains<T, F : Fn(T) -> bool>(x : Option<T>, f : F) -> bool {
    match x {
      Some(x) => f(x),
      None => false
    }
  }
  fn client_mut() {
    let r = contains(None, |x| x % 2 == 0); // OK
    assert!(r);

    let mut cnt = 0;
    let r = contains(Some(0), |x| {cnt += 1; x % 2 == 0}); // ERROR
  }
```

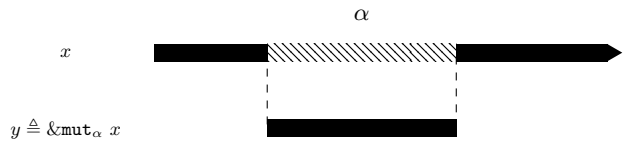
In the example above, we implement a function `contains`, which checks if an `Option` contains a value satisfying a predicate. The second call to `contains` raises an error as it attempts to mutate `cnt`, which would require the environment to be mutably borrowed.

2.4 Prophetic verification

Compared to the IMP language covered in § 2.1, Rust's borrows introduce a complication. Borrows inherently cause aliasing, as there are now two names that can refer to a value: the borrow

and the original variable from which the borrow was created. Many techniques exist for handling this difficulty, most notably *separation logic*, which was made explicitly for this case. These approaches ignore everything about the ownership system of Rust, and we must encode and verify all the properties usually provided by the Rust type system, significantly increasing the difficulty of verification.

If we could lean the other way and *leverage* the ownership guarantees, we might simplify verification. RustHorn[69] proposed a method to accomplish just this in 2020 by introducing the technique of *prophetic verification* for mutable borrows. In this approach, each borrow is represented by a pair of values: its *current* value, and its prophecized *final* value. Before we explain the prophecies' details, let us explore their motivation and how they fit into Rust. We need to find a specification for three operations: creating a borrow, writing to a borrow, and dropping a borrow. When a borrow is created, it holds the value it was lent. Then, while the borrow is active, it has exclusive access to that value; any mutations to it must go through the borrow. Finally, when the borrower's lifetime ends, the lender recovers its permissions and can observe the mutations made by the borrower.



That final step implies a form of *synchronization* between the borrow and lender, which is problematic because when a borrow expires, we may not know which variable it was obtained from, and thus, not who should be updated with a new value.

This is where RustHorn's prophecies intervene. Rather than trying to update the lender when the borrow is dropped, we instead *prophecize* the value it will have after the lifetime and update it when the borrow is created. Then the borrow merely needs to fulfill its prophecy; it suffices to observe that when it falls out of scope, its current value must be equal to its prophecized *final* value. We call this *resolving* the prophecy.

A reasonable reaction to learning about prophetic verification is suspicion: How can it possibly work? While the more formal answer is provided in Chapter 5, let's begin by considering a simple example.

```
1  let mut x = 0;          x ↦ 0
2  let y = &mut x;         x ↦ β, y ↦ ⟨0, β⟩
3  *y = 1;                 x ↦ β, y ↦ ⟨1, β⟩
4  drop(y);                x ↦ β, y ↦ ⟨1, β⟩, 1 = β
5  assert_eq!(x, 1);
```

To the right of each line of code, we annotate the program's state seen using RustHorn's prophecies. On line 2 we create a borrow and update the lender to point to the prophecy of the borrow. Then, on line 3, we update the borrow with a new value. Finally, on line 4, we drop the borrow and resolve the prophecy. At this point, we learn that the current value of the prophecy is equal to its final value and thus can prove the assertion on line 5.

By using a prophetic encoding in our example, the reasoning performed about the memory is purely local: manipulations to the borrow only affect the borrow's value. The resolution of the borrow introduces an equality that can be propagated until it encounters the lender, without requiring the knowledge of *who* the lender is to perform resolution.

Rust's borrows include operations other than merely writing: we can *split* them and *reborrow* them. Both these operations are naturally expressed in the vocabulary of prophecies. Splitting a borrow of a pair produces a pair of borrows; this operation can be expressed as the natural extension of splitting an owned pair: we split both the current and final values.

```
1    let x : &mut (T, U) = ..;
2    let (y, z) : (&mut T, &mut U) = z;
```
$x \mapsto \langle (x_1, x_2), (\alpha_1, \alpha_2) \rangle$
$y \mapsto \langle x_1, \alpha_1 \rangle,\ z \mapsto \langle x_2, \alpha_2 \rangle$

Reborrowing is a more complex operation in Rust. It can do many things, from shortening a borrow's lifetime to collapsing levels of indirection in nested borrows. Both patterns can be captured in prophetic reasoning:

```
1    let x = &'a mut T = ... ;
2    let y : &'b mut T = &mut * x;
```
$x \mapsto \langle x, \alpha \rangle$
$y \mapsto \langle x, \beta \rangle, x \mapsto \langle \beta, \alpha \rangle$

Shortening the lifetime can be treated the same way as borrowing the value pointed to by the original borrow `x`. When `y` expires, we recover `x` which will point to whatever the last value written by `y` was.

Collapsing indirection through a reborrow is a more complex operation, for which we will give interpretations in Chapter 4 and Chapter 5.

Hoare rules for prophecies We can make a step towards formalizing the prophetic reasoning we introduced by giving Hoare triples for the core operations of borrows in Rust.

BORROW

$$\frac{}{\{\forall \alpha,\ Q[y \mapsto \langle \mathtt{x}, \alpha \rangle, x \mapsto \alpha]\}\ \mathtt{y = \&mut\ x}\ \{Q\}}$$

RESOLVE

$$\frac{}{\{\mathtt{x.1} = \mathrm{x.0} \to Q\}\ \mathtt{drop(x)}\ \{Q\}}$$

WRITE

$$\frac{}{\{Q[x \mapsto \langle \mathtt{y}, \mathtt{x.1} \rangle]\}\ \mathtt{* x = y}\ \{Q\}}$$

SPLIT

$$\frac{}{\{Q[a \mapsto \langle \mathtt{x.0.0}, \mathtt{x.1.0} \rangle, a \mapsto \langle \mathtt{x.1.0}, \mathtt{x.1.1} \rangle]\}\ \mathtt{let\ (a,\ b)\ =\ x}\ \{Q\}}$$

SHORTEN

$$\frac{}{\{Q[y \mapsto \langle x, \beta \rangle, x \mapsto \langle \beta, \alpha \rangle]\}\ \mathtt{let\ y\ =\ \&mut\ *\ x}\ \{Q\}}$$

Combined with standard rules like those of IMP for basic operations, we are provided with a deductive system for core Rust. However, proving the soundness of these additional rules is significantly trickier and requires demonstrating that there *exists* a possible prophecy for the quantifier in BORROW. Proving this will be the subject of Chapter 5, which provides a fully mechanized proof of these rules.

Chapter 3

Introduction to the Creusot verifier

Creusot is a *deductive verifier* for Rust programs; it attempts to establish that a program is *correct* according to its specification. It receives as input a Rust library, which may feature custom *annotations* and generates a series of verification conditions meant to be verified with SMT and other automated verifiers. It builds upon the pre-existing Why3 verification platform by targetting an input format of Why3 called MLCFG. Why3 then generates the verification conditions and calls out to an extensive array of solvers for verification.

While in Chapter 4 and Chapter 5 we cover the mechanics of Creusot's translation and its soundness, in this chapter, we will focus on its *usage* and interface. In § 3.1 we introduce the basic usage of Creusot and its interface. Then § 3.2 introduces the Pearlite language Creusot uses to write specifications. In § 3.3 we use Pearlite to prove the functional correctness of a simple program. § 3.4 covers how to specify and prove properties of code involving *traits*. § 3.6 discusses the particularities of *closures* in Rust and Creusot. § 3.7 introduces *external specifications* which can be used to specify code belonging to external libraries. Finally, § 3.8 gives an initial evaluation of Creusot's performance on small benchmarks.

3.1 First steps with Creusot

We will consider an implementation of 'Gnome Sort', given in Figure 3.1. The Gnome Sort algorithm sorts an array using a single loop; it scans it from left to right, and whenever an element is out of order, it swaps it with its predecessor and then works from right to left until the element is in its place. This algorithm is inefficient but has the advantage of being relatively simple to implement.

To verify the program, we open our program in VSCode with the WhyCode[1] extension installed along with the configuration for VSCode provided with Creusot. When we open our file, we see the verification conditions generated by Why3 underlined in red in Figure 3.2a. Saving the file will launch automated provers on each obligation, proving them valid.[2]

When we run an unannotated Rust program through Creusot, it checks the *panic safety* of the program; in the case of our Gnome Sort, this means ensuring that the arithmetic does not

[1] Why3 also has a custom IDE built in GTK, which is more stable but offers a less beginner-friendly introduction to verification.

[2] Why3 features *transformations* (proof tactics), which can be manually applied to help verification when automated approaches stumble.

```
fn gnome_sort(v: &mut [i32]) {
    let mut i = 0;
    while i < v.len() {
        if i == 0 || v[i - 1] <= v[i] {
            i += 1;
        } else {
            v.swap(i - 1, i);
            i -= 1;
        }
    }
}
```

Figure 3.1: Gnome Sort

overflow and that the slice accesses are in-bounds. The Rust language helps us through its strict type system, which ensures slice indices are always of type `usize` and that only `v` can mutate the slice. Leveraging these guarantees and combining them with the loop guard is sufficient to prove the safety of this program.

In contrast, a similar proof in C would require additional specifications to guarantee properties not provided statically by the language.

3.1.1 Proving functional correctness

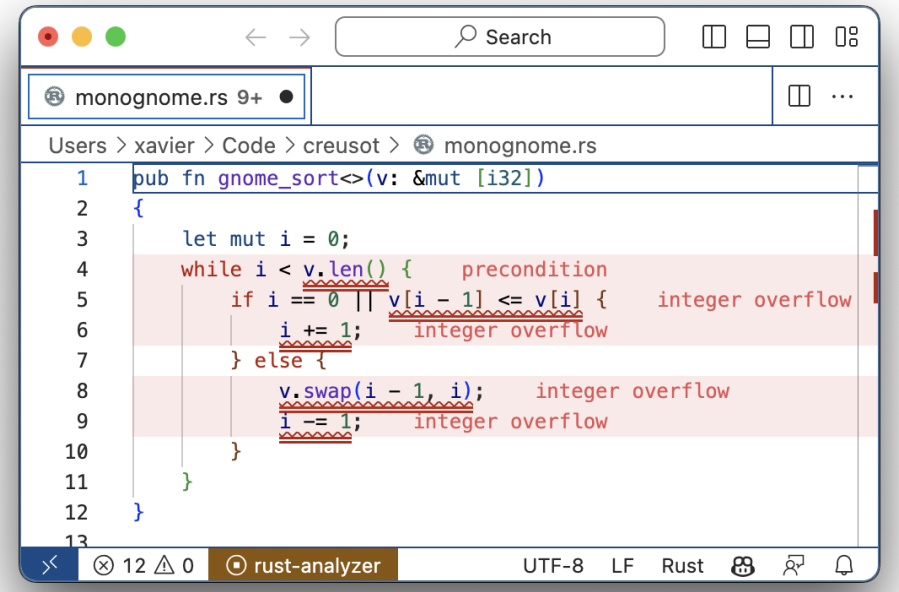
The current proof of Figure 3.1 shows the absence of panics but says nothing about *what* `gnome_sort` does. Proving the semantic properties of our function will require adding preconditions and post-conditions to the function, constraining the circumstances in which it can be called. In Creusot, we provide the `requires` and `ensures` clauses for this purpose. These clauses contain Pearlite expressions, a specification language provided by Creusot. Before we describe how to specify Figure 3.1, we will look at some simpler uses of Pearlite.

3.2 The Pearlite specification language

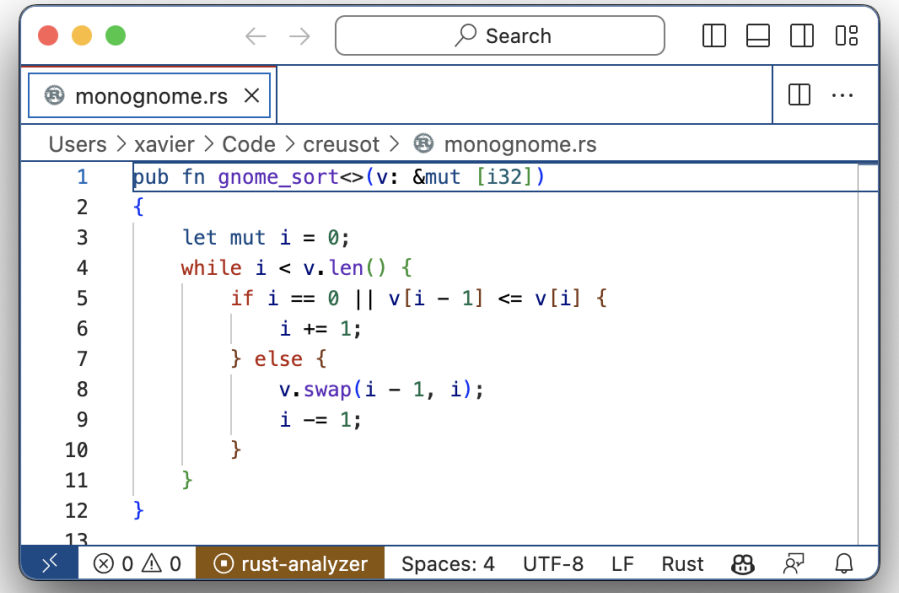
Pearlite is a non-executable, ownership-free specification language for Rust, its background logic is a classical, first-order, multi-sorted logic. Each Rust type corresponds to a sort in this logic. The logical connectives denoted by `&&`, `||` and `!` mirror their Rust counterparts, but Pearlite also introduces `==>` for implication, and the quantifiers `forall<v:t> formula` and `exists<v:t> formula`. Atomic predicates can be built using custom *logic functions and predicates*, constant literals, variables, and built-in symbols, a central case being the logical equality denoted by a double equal sign (`==`) and defined on any sort. This logical equality is the symbol interpreted to the set-theoretic equality in a set-based semantics. This distinguishes it from the *program equality* of Rust, which is sugar for `PartialEq::eq`. To reason about mutable borrows Pearlite introduces the *final* operator (`^`), which provides access to the prophecy of a mutable borrow. Using this operator, we could give and prove the following specification

```
#[ensures(^x == *x)]
fn drop_mut_bor(x : &mut T) { }
```

stating that the final value of `x` equals its initial value.



(a) Gnome Sort before verification



(b) Gnome Sort after verification

Figure 3.2

Pearlite has support for logical sorts that do not exist in Rust, like `Int`, the unbounded mathematical integers, or like `Seq<T>`, the generic sort of mathematical sequences. A syntactically valid formula is thus, for example:

```
forall<x: Int> x >= 0 ==> exists<y: Int> y >= 0 && x * x == x + 2 * y
```

Pearlite formulas are type-checked by the Rust compiler's front end but not borrow-checked. Hence, values can be used in logic functions or predicates, even if the Rust ownership rules forbid copying them.

Contract Clauses Pearlite can be used in several specification clauses:

1. `requires`: Attached to functions or closures, allows specifying a precondition to function calls. This clause is evaluated in the *prestate* of the function. When attached to a *trait declaration*, this specifies the *strongest precondition* that implementations can require. Each trait implementation must refine this precondition.
2. `ensures`: Attached to functions or closures, specifies the postcondition of a function call. Unlike in many other verification tools, this clause is also evaluated in the *prestate* of the function[3]. When attached to a *trait declaration*, this specifies the *weakest postcondition* that implementations can provide.
3. `variant`: Used to prove the termination of functions in both Rust and Pearlite, it accepts an expression whose result must be a strictly decreasing well-founded order. The `variant` clause can be attached to loops and functions to show their termination.
4. `trusted`: Indicates that a contract should be assumed, and that Creusot should not generate any proof obligations.

The `decr_even` function below uses a `requires` and `ensures` clause to decrement an integer if it contains an even value:

```
#[requires(x % 2 == 0)]
#[ensures(result == x - 1)]
fn decr_even(x : i32) -> i32 { x - 1 }
```

Other Pearlite clauses Besides contracts of declarations, Pearlite can be used within definitions to help in the proof of properties, in particular:

1. `invariant`: Can be attached to loops and states a property that must be valid before each iteration.
2. `proof_assert`: Inserts a *ghost* assertion which is checked for validity. This assertion does not exist in the compiled binary.
3. `ghost`: Inserts a *ghost* expression in the code, which can be used in further ghost expressions, assertions or invariants. This expression does not exist in the compiled binary.

The `proof_assert` annotation is often used to debug proofs; it allows the user to verify that intermediate properties hold at specific program points and is a reasonable manner of understanding why a specific proof is not passing.

[3] An interesting consequence of the Rust type system is that since function arguments are passed by value and with ownership, they are effectively consumed by the function. Given a function `fn omg(x: T) x` does not meaningfully exist at the end of the execution of `omg`, in fact, `x` may have been moved or dropped at an arbitrary point in `omg`'s execution. The primary manner of performing or observing side-effects in Rust is through *mutable borrows*, and when using those, Pearlite offers the `^` operator to speak of the final value of a borrow. This property is equally valid before or after function execution.

Pearlite definitions Finally, we often wish to write auxiliary definitions in Pearlite to be used in the previous clauses. For this reason, we can write a function with the `predicate` or `logic` attribute to state that these definitions are Pearlite symbols and not Rust functions. In particular, these symbols can use Pearlite constructs like quantifiers and mathematical types.

```
#[logic]
fn sqr(x: Int) -> Int { x * x }

#[logic]
#[variant(x)]
fn sum_of_odd(x: Int) -> Int {
  if x <= 0 {
    0
  } else {
    sum_of_odd(x - 1) + 2 * x - 1
  }
}
```

Pearlite definitions must be *total*, and thus we must provide a `variant` clause for the `sum_of_odd` example above.

Lemmas A useful feature of Pearlite is the introduction of user lemmas using the so-called *lemma function* construction. To achieve this, one provides a contract to a logical function. By proving the contract valid, one obtains a lemma stating that for all values of arguments, the preconditions imply the postconditions. This construction is even able to prove lemmas by induction. Here is an example:

```
#[logic]
#[requires(x >= 0)]
#[variant(x)]
#[ensures(sum_of_odd(x) === sqr(x))]
fn sum_of_odd_is_sqr(x:Int) { if x > 0 { sum_of_odd_is_sqr(x-1) } }
```

This code is automatically proven to conform to its contract. A user can then use the lemma inside of a ghost or proof assertion to introduce a new hypothesis: $\forall x, x \geq 0 \Rightarrow$ `sum_of_odd`$(x) = x^2$ in the current proof context. Lemma functions allow users to provide specific hints to the verifiers, which can be used to prove more complex properties.

3.3 Proving 'Gnome Sort' correct

Using Pearlite we can state what it means for `gnome_sort` to be *correct*. An essential, expected property of a sorting function is that after a call, the array is *sorted*. We start by defining a predicate `sorted_range` as follows:

```
#[predicate]
fn sorted_range(s: Seq<i32>, l: Int, u: Int) -> bool {
    pearlite! {
        forall<i : Int, j : Int> l <= i && i < j && j < u ==> s[i] <= s[j]
    }
}

#[predicate]
fn sorted<(s: Seq<i32>) -> bool {
    sorted_range(s, 0, s.len())
}
```

The predicate `sorted_range` states that any two ordered pairs of indices `i`, `j` are ordered in the sequence `s`. A sequence is a mathematical array type with `Int` length. Recall that Pearlite functions are *total*, and there is no requirement to check that sequence accesses are in bounds, if the index `i` is out of bounds, then `s[i]` produces an unspecified value. We then define the `sorted` predicate using `sorted_range`, providing the lower and upper bounds.

In both these definitions, we used *mathematical sequences*, but in our `gnome_sort` function, we use a `slice`. We distinguish the two types in Creusot, but we can convert a slice into a sequence using a *model function*. Given a slice `[T]`, we can canonically build a sequence `Seq<T>`, which is just a direct mapping of the slice. Lifting program types into their mathematical abstractions is an essential operation of verification in Creusot and Pearlite provides a dedicated, post-fix operator for this `@` (pronounced *model*). On line 1, we use this to convert the final value of `v` into a sequence in the code below:

```
#[ensures(sorted((^v)@))]
#[ensures((^v)@.permutation_of(v@))]
pub fn gnome_sort(v: &mut [i32])
{
    let old_v = ghost! { v };
    let mut i = 0;
    #[invariant(sorted_range(v, 0, i@))]
    #[invariant(v@.permutation_of(old_v@))]
    while i < v.len() {
        if i == 0 || v[i - 1] <= v[i] {
            i += 1;
        } else {
            v.swap(i - 1, i);
            i -= 1;
        }
    }
}
```

We must also use a *loop invariant*, a property maintained by each loop iteration, to prove this postcondition. On line 7, our invariant is that `v` is sorted from index 0 to `i`. To prove this invariant valid, we must first ensure it is true when we enter the loop (also called *initialization*); here, it is trivially the case when $i = 0$ as the interval $[0, 0)$ is empty. Then, the invariant must also be upheld by each iteration of the loop (called *preservation*); we have two cases:

1. If `i` was 0, then after one iteration the singleton slice `v[0..1]` will be sorted. Otherwise, if `v[i - 1] <= v[i]`, then given `sorted_range(v, 0, i@)` (from the previous iterations), `sorted_range(v, 0, i@ + 1)` must also be sorted, since `v[i]` must be bigger than all previous elements.

2. Otherwise, we've found a pair `v[i-1] > v[i]`. In this case, we swap both values, restoring the sorted property between them and *decrement* `i`. Since `sorted_range(v, 0, i@)`, we also have `sorted_range(v, 0, i@ - 1)`.

On line 2, we state the second fundamental property of a sorting algorithm: that it does not delete or duplicate elements. To prove this, we use a second invariant, which relates the state of `v` at each iteration to `old_v`, a *ghost* copy of `v` at the start of the function. Because the only modification we make to `v` is swapping elements, proving that we always have a permutation is easy.

The `ghost!` macro allows us to insert additional *ghost code* [34] to help with verification. Ghost code and ghost values must satisfy an *erasure* property: the program's behavior with the ghost code removed must be identical to the program with ghost code. Ghost code can be used to construct values of mathematical types like `Int` and `Seq`. The `ghost!{x}` macro evaluates to values of type `Ghost<T>` where `T` is the type of `x`. This can store ghost values inside types and variables, a valuable feature for more considerable verification efforts (Chapter 7).

3.4 Working with traits

Rust traits can be specified like normal Rust functions and include logical predicates and functions. Any contract provided to a trait will generate *refinement* obligations for individual implementations. They must demonstrate that the implementation's contract is at least as strong as the generic one given to the overarching trait. This contract does not have to be *identical*; we can give more robust contracts or contracts stated in a manner easier to work with for a specific type.

Traits are often used to abstract over operations that should respect specific identities, like `Ord` [86], which is expected to implement a total order. Users should be allowed to rely on these properties for correctness, even in generic contexts. In a trait definition, we can include lemma functions to require specific behaviors from the implementations of traits.

```
1   trait Ord {
2     fn cmp(&self, rhs: &Self) -> Ordering;
3
4     #[law]
5     #[requires(a.cmp(b) == o && b.cmp(c) == o)]
6     #[ensures(a.cmp(c) == o)]
7     fn transitivity(a : &Self, b : &Self, c : &Self, o : Ordering);
8
9     // ...
10  }
```

To avoid increasing the size of contexts, Creusot will only include a lemma if it is called at least once in the verified function. However, when working generically with traits like `Ord`, we want to ensure that the *transitivity* (line 7) is always available. Creusot allows specifying these identities or *laws* in traits by through `#[law]` definitions. Laws are particular kinds of logic functions that are *auto-loaded* by Creusot when a trait is used.

3.5 Verifying a generic program

Let's consider the verification of a more challenging program than `gnome_sort`: a different, generic version of `find` on a *singly linked-list*. Our `find_mut_tail` function accepts a list of type `List<T>` and an element of `T` and returns a mutable pointer to the first cell containing this element if any is

```
enum List<T> {
    Cons(T, Box<List<T>>),
    Nil,
}

impl<T> List<T> {
  #[logic] fn get(self, i: Int) -> Option<T> { .. }

  #[logic] fn len(self) -> Int {
    match self {
        Nil => 0,
        Cons(_, tl) => tl.len() + 1,
    }
  }
}

use List::*;

fn find_mut_tail<'a, T: Eq>(list: &'a mut List<T>, x: &T) -> Option<&'a
    mut List<T>> {
    match list {
        Nil => None,
        Cons(y, _) if y == x => Some(list),
        Cons(_, tl) => find_mut(tl, x),
    }
}
```

Figure 3.3: Generic `find` on a linked list

```
1  trait PartialEq<Rhs> {
2    #[ensures(result == (self.deep_model() == rhs.deep_model()))]
3    fn eq(&self, rhs: &Rhs) -> bool
4    where Self: DeepModel,
5          Rhs: DeepModel<DeepModelTy = Self::DeepModelTy>;
6  }
7
8  trait DeepModel {
9      type DeepModelTy;
10     #[logic] fn deep_model(self) -> Self::DeepModelTy;
11 }
```

Figure 3.4: The specification of `PartialEq`

present. The code can be found in Figure 3.3. As with `gnome_sort`, passing this program through Creusot will only check the absence of panics, and does so automatically.

Note, we also define a few *logical functions*, `get` (line 7) and `len` (line 9), which we will use in the specification of `find_mut_tail`. We provide the code for illustration for `len`, showing how we treat `List` as the ordinary list type in our logic. The definition for `get` is analogous; it is the standard form of `get` on a list.

The implementation of `find_mut_tail` is generic over the element type `T`, but requires that `T` implements `Eq`. The specification of `PartialEq` is given in Figure 3.4, and makes use of the `DeepModel` trait. In Creusot, the `DeepModel` trait is used to describe the view of Rust types taken by `PartialEq`, `PartialOrd`, and `Hash`. To help implement these traits, Creusot provides a `#[derive(DeepModel)]` macro, which automatically generates a `DeepModel` implementation for a type.

Using `DeepModel`, we can give a generic specification for this function. We provide the complete specification in Figure 3.5, which we will now cover in detail. First, if the result was `None` (line 4), then no element in the list can have a deep model equal to our target `x`. The specification is more interesting when the result is `Some(n)`. Then, we state that the first element of `n` is equal to `x` (line 8). We then state that the initial elements of `n` are a suffix of our input list (line 12). Finally, we also state that the *final* elements of `n` are a suffix of the *final* elements of `list` (line 14).

This specification, especially the `Some` case, highlights one of the unique aspects of proving programs using Creusot: we never reason about *pointer values* themselves, but exclusively about their contents at different points in the program. With this specification, the proof passes automatically.

3.6 Higher-order functions

Closures can be specified using `#[requires]` and `#[ensures]` attributes like other functions.

```
1    iter.map(#[requires(x <= 10)] |x| { x + 1 })
```

These pre and postconditions can refer to the closure's arguments, return value, and *captures*. Unlike the parameters of standard functions, the captures of a closure alias their environment. In particular, all modifications to a captured variable are visible outside the closure. Consider a normal Rust function `fn inc(mut x: i32) { x += 1; }`, the modification to the parameter `x` is not visible in the callee as Rust uses call-by-value semantics. However, in the closure `|| { x += 1; }`, the modification to the capture `x` is visible outside of the closure.

```
#[ensures(match result {
  None => ^list == *list &&
    (forall<i : _> 0 <= i && i < list.len() ==>
      list.get(i).deep_model() != Some(x.deep_model())),
  Some(n) => {
    let offset = list.len() - n.len();
    n.len() <= list.len() &&
    n.get(0).deep_model() == Some(x.deep_model()) &&
    (forall<i : _> 0 <= i && i < offset ==>
        list.get(i) == (^list).get(i)) &&
    (forall<i : _> 0 <= i && i < n.len() ==>
        list.get(i + offset) == n.get(i)) &&
    (forall<i : _> 0 <= i && i < (^n).len() ==>
        (^list).get(i + offset) == (^n).get(i)) &&
  },
})]
fn find_mut<'a, T: Eq + DeepModel>(list: &'a mut List<T>, x: &T) ->
    Option<&'a mut List<T>> { .. }
```

Figure 3.5: Specification of `find_mut_tail`

This behavior unique to closures requires introducing a new contract operator to distinguish the value of a closure capture before and after the call. Creusot uses the `old` pseudo-function for this role.. The prior closure could be given the specification `#[ensures(old(x) + 1 == x)]` to state that the closure increments its argument by one. When used in a postcondition of a closure `old(x)` refers to the value of `x` upon entry to the closure call.

3.6.1 Specifying clients of closures

The consumers of closures also need a mechanism to reason about the behavior of the closures they receive. Closures can have arbitrary preconditions or postconditions, and the clients must ensure they have sufficient permissions to call the closure. Earlier, in § 2.3.4, we showed how Rust uses the `Fn`-traits to abstract over closures. In Creusot, we extend those traits to abstract over the pre and postconditions of closures.

For example, suppose we wish to call a *mutable* closure with negative integer values; we can write the following function:

```
#[requires(forall<f : F, x: i32> x < 0 ==> f.precondition((x,)))]
fn call_with_neg<F: FnMut(i32)>(mut f: F) {
  f(-1);
}
```

We use the `precondition` predicate to state that all values of the closure type `F` must accept negative values. Note, `precondition` accepts the arguments of the closure as a *tuple* (even if the closure only has one argument), we use `(x,)` to create a 1-tuple.

Handling the converse is more complicated: as covered in § 2.3.4, there are three different ways to call a closure, each manipulating the environment differently. In consequence, we do not have a single `postcondition` predicate but *three*, one for each of the `Fn`-traits.

```
trait FnOnce<Args> {
    type Output;

    #[predicate] fn precondition(self) -> bool;

    #[predicate] fn postcondition_once(self) -> bool;

    #[requires(self.precondition(args))]
    #[ensures(self.postcondition_once(args))]
    fn call_once(self, args: Args) -> Self::Output;
}
```

Figure 3.6: `FnOnce` extended with specifications

We implemented these additional predicates by extending the `Fn`-trait hierarchy, beginning with `FnOnce` in Figure 3.6. We introduce two predicates: `precondition` capturing the precondition of the closure, and `postcondition_once`, which describes the postcondition of calling a closure through `call_once`. Creusot automatically provides the bodies of these predicates for closures using the `requires` and `ensures` clauses the user provided. The bodies of these predicates are precisely the contents of the clauses.

The `FnMut` trait (Figure 3.7) introduces another two predicates- `postcondition_mut` and `unnest`, along with several laws, including `fn_mut_once` relating `postcondition_mut` to `postcondition_once`. The contract of an `FnMut` closure has two possible interpretations, depending on whether it is called through `call_mut` or `call_once`. However, these specifications should be *compatible*, we link them through the `fn_mut_once` law:

```
self.postcondition_once(args, res) ==
      exists<s: &mut Self> *s == self &&
        s.postcondition_mut(args, res) && (^s).resolve()
```

This law states that calling an `FnMut` closure through `call_once` is equivalent to calling it through `call_mut` and then resolving (throwing away) the closure state.

The second part of `FnMut` is the `unnest` predicate. This predicate captures a property unique to closure types: we can only mutate the values pointed by the captures, not the captures themselves. Consider the following simple example:

```
1  let mut cnt = 0;
2  let mut c = || { cnt += 1;};
3  c();
4  c();
5  assert!(cnt == 2);
```

The closure `cnt` borrows the variable `c`, and its value is tested on line 5. If Creusot treats the closure in a fully opaque manner, it has no way of knowing that after each call the environment holds a reference to the *same* variable `cnt`. To work around this, for each `FnMut` we generate an `unnesting` predicate which relates the mutable captures of the closure before and after the call. This predicate allows Creusot to know that the closure is mutating the same variable at each call.

The unnesting predicate is essential to link closure states throughout repeated calls. Without it, we would lose track of the contained mutable borrows. This property is an example of a *historical invariant* [94], a property that holds throughout the lifetime of a closure, and thus we impose that this predicate is reflexive and transitive using two laws: `unnest_refl` and `unnest_trans`.

```
trait FnMut<Args>: FnOnce<Args> {
    #[predicate]
    fn postcondition_mut(&mut self, _: Args, _: Self::Output) -> bool;

    #[predicate]
    fn unnest(self, _: Self) -> bool;

    #[law]
    #[requires(self.postcondition_mut(args, res))]
    #[ensures((*self).unnest(^self))]
    fn postcondition_mut_unnest(&mut self, args: Args, res:
    Self::Output);

    #[law]
    #[ensures(self.unnest(self))]
    fn unnest_refl(self);

    #[law]
    #[requires(self.unnest(b))]
    #[requires(b.unnest(c))]
    #[ensures(self.unnest(c))]
    fn unnest_trans(self, b: Self, c: Self);

    #[law]
    #[ensures(
      self.postcondition_once(args, res) ==
        exists<s: &mut Self> *s == self &&
          s.postcondition_mut(args, res) && (^s).resolve()
    )]
    fn fn_mut_once(self, args: Args, res: Self::Output);

    #[requires(self.precondition(args))]
    #[ensures(self.postcondition_mut(args))]
    fn call_mut(&mut self, args: Args ) -> Self::Output;
}
```

Figure 3.7: The `FnMut` trait extended with specifications

```
trait Fn<Args>: FnMut<Args> {
    #[predicate]
    fn postcondition(&self, _: Args, _: Self::Output) -> bool;

    #[law]
    #[ensures(self.postcondition_mut(args, res) == (self.resolve() &&
    self.postcondition(args, res)))]
    fn fn_mut(&mut self, args: Args, res: Self::Output);

    #[law]
    #[ensures(self.postcondition_once(args, res) == (self.resolve() &&
    self.postcondition(args, res)))]
    fn fn_once(self, args: Args, res: Self::Output);
}
```

Figure 3.8: The `Fn` trait extended with specifications

Finally, `Fn` (Figure 3.8) imposes that the closure is immutable and introduces the final form of postcondition- `postcondition`.

In a `Fn` closure, we can no longer use `old` as the closure is immutable.

Two laws, `fn_mut` and `fn_once` relate `postcondition` to each of `postcondition_mut` and `postcondition_once` respectively. These state that the postconditions of `FnMut` and `FnOnce` closures are equivalent to the postcondition of `Fn` closures, but only if the closure is resolved.

3.7 Interfacing with the real world

Creusot uses a modular approach to verification, meaning it cannot look inside the bodies of functions during calls. This means the behavior of functions is solely modeled by the contract attached to that function. However, to verify real-world Rust code, we must at least be able to call standard library functions, for which we cannot (directly) prove a specification. To attach contracts to standard library functions, Creusot provides the `extern_spec!` macro.

The `extern_spec!` macro allows users to write contracts for arbitrary Rust functions, which will be used by Creusot in the translation. The argument to `extern_spec!` is a forest of Rust paths, eventually ending in Rust *functions*. Each function can be given a contract, which the translation will use by Creusot. Below, we show an example of a specification for `Vec::new`:

```
extern_spec! {
  mod std {
    mod vec {
      impl<T, A: Allocator> Vec<T, A> {
        #[ensures((@result).len() == 0)]
        fn new() -> Self;
      }
    }
  }
}
```

Simply attaching a contract to a function is often insufficient; we may also need to require trait implementations for generic parameters. Consider the case of the function

```
1    fn binary_search<T : Ord>(self: &[T], x: &T) -> Result<usize, usize>
```

from `std::slice`. For this function to work correctly, it needs a precondition that the slice is sorted. However, since this function is generic over `T`, we need a way to express a generic order over `T`. In § 3.4 we showed how to provide a specification to `Ord`. Thus, to write our extern specification, we must have a `T` that satisfies our `Ord` specification trait. For this reason, when a user writes an `extern_spec`, we allow the bounds to be *stronger* than for the original function. We check that the bounds in the `extern_spec` imply the original bounds. Additionally, we verify that these additional bounds are satisfied each time the specification is used (at call sites).

3.8 Evaluation

With Creusot, we value verification performance; a short verification time allows faster iteration on proofs and verified code. We ran Creusot on a wide range of benchmarks to demonstrate that our approach allows efficient verification[4]. These benchmarks use *polymorphism* and *traits*. We adapted and generalized programs from the Prusti [4] benchmark suite, strengthening the verified properties. Other examples were inspired from the Why3 gallery [12], Rosetta Code [71] or RustHorn [69].

Proof Strategy Note that Why3 supports a wide range of manual proof tactics that allow users to set up the proof structure before handing off obligations to provers. As these can dramatically help verification, we avoid them in our evaluation and instead apply a standard proof strategy to all examples. Each example is proved using Why3's "Auto Level 3" strategy, a common first step when verifying programs with Why3. One benchmark required a small number of additional manual proof steps, "Sparse Array", to prove a complex lemma about injections between sequences.

Our evaluation used a 2016 Macbook Pro running macOS 11.6 installation with an Intel Core i7-7920HQ CPU and 16GB of RAM. We relied on a combination of Alt-Ergo 2.4.1, Z3 4.8.17, and CVC4 1.8 as back-ends to Why3.

3.8.1 Discussion

The selected results are presented in Table 3.1, where benchmarks are grouped by origin. The first group comes from RustHorn's evaluation [69, §4.3], where we added specifications of the intended functional behavior. The second group of benchmarks is adapted from Prusti's evaluation [4, §7.2]. The third group is novel examples contributed as part of Creusot's test suite. "Filter Vector" is a challenging example regarding reasoning on memory separation [45]. "Sparse Array" is an example from the VACID-0 benchmarks [61]. The proof involves a mathematical lemma with a few steps of manual proof [25] before sending the sub-goals to SMT solvers. "In Place List Rev." is the in-place linked-list reversal procedure classically used to illustrate reasoning in separation logic. Remarkably, the Rust code for that can be verified without separation logic.

Our RustHorn tests show that we maintain the verification performance of RustHorn, as our provers rapidly verify these examples. While some manual annotation is required, even for safety, the overhead is low and primarily consists of stating the properties we wanted to prove in the first place.

The Prusti examples listed here are derived from their introductory paper in 2019 [4]. Their paper provides two versions for their functions, the first proving only safety while the second proving portions of functional correctness.

[4]The benchmarks can be found at https://github.com/xldenis/creusot/tree/master/creusot/tests/should_succeed

Name	LOC	Spec. LOC	# of VCs	Time (s)	Additional Properties
Inc Some List	25	22	4	0.98	Func. correctness
Inc Max	12	3	2	0.53	Func. correctness
Inc Max Many	13	3	2	0.74	Func. correctness
Binary Search	21	20	31	2.15	Func. correctness
Knapsack 0/1	32	52	81	3.94	—
Knapsack 0/1	32	106	113	5.96	Func. correctness
Knuth Shuffle	9	11	1	0.30	Permutation
100 doors	18	6	3	1.08	—
Heap Sort	30	71	125	14.6	Func. correctness
Selection Sort	15	27	30	2.14	Func. correctness
Gnome Sort	11	17	31	2.06	Func. correctness
Filter Vector	21	39	6	0.98	—
Sparse Array†	47	75	37	4.86	Func. correctness
In place List Rev.	12.	10	1	0.55	Func. correctness
All Zero List	11	10	1	0.64	Func. correctness
Index Mut List	18	53	3	0.59	Func. correctness
Swap Pair	9	3	2	0.48	Func. correctness
HashMap	50	111	71	5.43	Func. correctness
Red Black Tree	275	596	1029	31.27	Func. correctness

Table 3.1: Selected results of our evaluation. The column "LOC" indicates the lines of program code (excluding blank lines) we verify. The column "Spec. LOC" measures the lines of specifications (excluding blank) used. "# of VCs" measures the number of verification conditions sent to CVC4 or Z3 as proof tasks. "Time (s)" measures the time Why3 takes to run the provers. Tests marked with † required manual proof steps in Why3 IDE [25].

The difference in verification performance is evident by the "Knapsack 0/1" example of Prusti. This example solves the 0/1-Knapsack problem using the traditional dynamic programming approach. Prusti takes over 2 minutes to verify the safety of the problem, whereas our proof of safety passes in approximately 4 seconds. This difference in performance helps us go further; checking proofs rapidly allows for faster iteration, which enabled us to extend this example with a complete proof of functional correctness. Our version of the Knapsack Problem with functional correctness takes longer to verify, with the proof passing in approximately 6 seconds.

The 'Index Mut List' Benchmark

The benchmark 'Index Mut List' is a simple example that demonstrates the power of Creusot's prophecies. It traverses a singly-linked list, returning a borrow to the `ix`-th element. The code is shown below:

```
1    pub struct List(u32, Option<Box<List>>);
```

```
2
3     pub fn index_mut(mut l: &mut List, mut ix: usize) -> &mut u32 {
4       while ix > 0 {
5         l = l1.as_mut_ref().unwrap();
6         ix -= 1;
7       }
8
9       &mut l.0
10    }
```

The specification of this function seems rather simple, the returned borrow should correspond to the 'projection' of the `ix`-th cell of the borrow on the whole list. Specifically, the client should be able to modify the returned value, and when that borrow is dropped recover the original list with the modification. We can state these properties with the Pearlite specification below:

```
1  #[requires(ix@ < l.len())]
2  #[ensures(Some(*result) == l.get(ix@))]
3  #[ensures(Some(^result) == (^l).get(ix@))]
4  #[ensures((^l).len() == (*l).len())]
5  #[ensures(forall<i:Int> 0 <= i && i < l.len() && i != ix@ ==> l.get(i) ==
       (^l).get(i))]
```

Proving this specification however is subtle, due to the usage of a loop. Let's consider the simplest case: we wish to establish that the loop leaves the list's length unchanged, that is: `(*l).len() == (^l).len()`. Unlike a normal loop invariant, this property depends on *future information*, proving this true requires knowledge of the value of `^l`. Supposing this property is true for the following iterations, we can establish it for the current one. We formalize this through an implication: if the property holds for the current value of `l` then it also holds for the *orignal* one.

```
1  let old_l = ghost! { l };
2  #[invariant((^l).len() == l.len() ==> (^old_l).len() == old_l.len())]
3  while ix > 0 {
4    l = l1.as_mut_ref().unwrap();
5    ix -= 1;
6  }
```

At the entry of the loop, we can initialize the invariant since `l = old_l`. Then we must establish the preservation of this invariant. Since we peeled off one element of the list, we know that the length of `l` has decreased by one. By supposing the property holds for the tail of the list (the antecedent of the invariant), we can demonstrate that it must have held at the start of the iteration, and thus can prove the property.

This sort of backwards reasoning is commonly found when working with *separation logic* as a *magic wand*. The magic wand $P \mathrel{-\!\!*} Q$ encodes the idea of a property Q with a P shaped hole, exactly like the kind of reasoning we were just performing. The approach Creusot uses to verification effectively subsumes magic wands within safe Rust, turning them into a *design pattern* for specifications. The remaining postconditions can be proved through similar invariants.

3.8.2 Limitations & Unsupported Features

Just as important as the features Creusot does support are those that are not yet implemented or otherwise unsupported:

1. *Drop*: Creusot does not yet support reasoning about the `Drop` trait. The challenge of drop is two-fold: 1) the Rust borrow checker special-cases certain behaviors in the presence of drop (also called the 'dropck eyepatch'), and 2) the specification to give `Drop` is unclear.

2. *Mutually recursive functions*: Though Creusot has long supported simple recursive functions, there are engineering challenges to supporting mutually recursive functions.
3. *Trait Objects*: Currently, trait objects can be present in functions, but we cannot meaningfully specify them; as such, they can only be used in traits with no specification like `Display` or `Debug`.
4. *Async*: Though we support the similar notion of *iterators*, Creusot does not yet support futures or `async`/`await`.
5. *Unsafe Code*: As we have already stated, Creusot is a verifier for *safe* Rust and crucially leverages the safety guarantees of the Rust type system for its soundness.

Chapter 4

Implementing a Rust verifier

> This chapter is based on the "Creusot: A Foundry for the Deductive Verication of Rust Programs" article about Creusot, but the content has been significantly expanded and revisited.

This chapter presents the design and implementation of Creusot, a verifier for Rust programs built on top of the Why3 [31] verification platform. It uses a prophetic approach inspired by RustHorn [69] but targets *pure*, *functional* languages rather than predicate transformers. Despite its academic origins, the purpose of Creusot is not merely to demonstrate the possibility of verifying Rust programs but to provide a practical tool for verifying real-world Rust programs. Much work has gone into expanding the subset of Rust supported by Creusot, allowing more idiomatic Rust code to be used.

The remainder of the chapter will be structured in the following manner. In § 4.1, we will introduce MIR, the source language of Creusot, and the kernel language of Rust. In § 4.2, we will introduce MLCFG, the target language of Creusot. With the source and destination defined, we will present the translation of Creusot in § 4.3. We will focus on Creusot's handling of *prophecies* (§ 4.3.2) and *traits* (§ 4.3.4). We finish by discussing related projects (§ 4.6).

4.1 The MIR language

Despite being a Rust verifier, Creusot does not start its work from the source Rust code. Instead, it uses the *Mid-Level Intermediate Representation (MIR)* as the source language for our translation. This language can be considered the 'kernel' language of Rust; it has a drastically simplified syntax, is structured as a CFG, and makes the semantics of Rust programs explicit[1]. In particular, the flagship analysis of 'borrow checking,' responsible for enforcing Rust's 'Mutability XOR Aliasing' discipline, is performed on the MIR. Since the introduction of non-lexical lifetimes, it is not clear that reasoning about borrows could be performed on a more traditional tree-structured language. Furthermore, the Rust project has strongly indicated that MIR should be considered the interface for tools external to the compiler.[2]

[1] This is a white lie, as there remain open questions about many details of MIR semantics, which the Rust project is actively working on closing.

[2] The 'Stable MIR' project aims to officialize this by offering an official, public API that permits reading the MIR of Rust programs.

$$
\begin{aligned}
\textit{Place} \ni \mathtt{p} &::= \ell \mid \mathtt{*\ p} \mid \mathtt{p\ [\ }\ell\mathtt{\ ]} \mid \mathtt{(p\ as\ }C\mathtt{).}f \\
\textit{Operand} \ni \mathtt{o} &::= \mathtt{move\ p} \mid \mathtt{copy\ p} \mid n \\
\textit{RValue} \ni \mathtt{r} &::= \mathtt{o} \mid \mathtt{\&\ p} \mid \mathtt{\&mut\ p} \mid \mathtt{o} \oplus \mathtt{o} \mid C\ \overline{\mathtt{o}} \\
\textit{Statement} \ni \mathtt{S} &::= \mathtt{p} = \mathtt{r} \\
\textit{Terminator} \ni \mathtt{T} &::= \mathtt{switch\ o}\ \overline{C \to \mathtt{BB}_k} \mid \mathtt{goto\ BB}_k \mid \mathtt{assert\ o} \to \mathtt{BB}_k \\
&\quad \mid \mathtt{call}\ f\mathtt{(\overline{o})}\ \to\ \mathtt{BB}_k \mid \mathtt{return} \mid \mathtt{unreachable} \\
\textit{BasicBlock} \ni B &::= \mathtt{BB}_k\ \{\overline{\mathtt{S}};\ \mathtt{T}\} \\
\textit{VarDecl} \ni V &::= \mathtt{let}\ \ell : \tau\ ; \mid \mathtt{let\ mut}\ \ell : \tau\ ; \\
\textit{Function} &::= \mathbf{fn}\ \phi\mathtt{<}\overline{\alpha}, \overline{\xi}\mathtt{>(}\overline{\mathtt{b} : \tau}\mathtt{)}\ \to\ \tau = \overline{V}\ \overline{B}
\end{aligned}
$$

Figure 4.1: Syntax of MIR (simplified)

4.1.1 Syntax

We present a slimmed-down version of MIR, used by Creusot as its source language. The MIR language contains additional syntactic constructs used for error reporting, type inference, and dynamic analysis, which are irrelevant to the translation. One simplification is made by Creusot: it does not support *call-stack unwinding* (used in the implementation of panics), and thus all relevant control-flow edges are removed. The other significant simplification is *drop*, the Rust *destructor* function, Creusot does not support `drop`, and thus we erase the relevant terminators from MIR. The remaining language is presented in Figure 4.1.

A MIR function ($\mathbf{fn}\ \phi\mathtt{<}\overline{\alpha}, \overline{\xi}\mathtt{>(}\overline{\mathtt{b} : \tau}\mathtt{)} \to \tau_1$) has a name ($\phi$), a set of lifetime parameters ($\overline{\alpha}$), a set of type parameters ($\overline{\xi}$), a set of function parameters ($\overline{\mathtt{b} : \tau}$) and produces a value of an output type τ_1. Each function has a series of local variables ($\overline{V}$) and a series of basic blocks ($\overline{B}$). Local variables are either mutable ($\mathtt{let\ mut}\ \ell : \tau$) or immutable ($\mathtt{let}\ \ell : \tau$), and their scope ranges over the whole function. A *basic block* ($\mathtt{BB}_k\ \{\overline{\mathtt{S}};\ \mathtt{T}\}$) is composed of assignments $\overline{\mathtt{S}}$, a terminator $\mathtt{T}$, and labeled with a unique identifier k. By convention, the entry point to a function is $\mathtt{BB}_0$. Assignments are of the form $\mathtt{p} = \mathtt{r}$, where $\mathtt{p}$ is *place* and $\mathtt{r}$ is a *right-hand side value* (RValue). A *place* ($\mathtt{p}$) is either a local variable (ℓ), a dereference ($\mathtt{*\ p}$), an array access ($\mathtt{p\ [\ }\ell\mathtt{\ ]}$) or a field access ($\mathtt{(p\ as\ }C\mathtt{).}f$) where C is a constructor name and f is a field index within that constructor. An RValue ($\mathtt{r}$) is either an *operand* ($\mathtt{o}$), an immutable borrow ($\mathtt{\&\ p}$), a mutable borrow ($\mathtt{\&mut\ p}$), a primitive binary operation ($\mathtt{o} \oplus \mathtt{o}$) which includes arithmetic and comparison, or a constructor ($C\ \overline{\mathtt{o}}$). An operand ($\mathtt{o}$) is either a *move* ($\mathtt{move\ p}$), a *copy* ($\mathtt{copy\ p}$) or a constant (n).

4.1.2 Informal semantics for MIR

We will not present formal semantics for MIR here because the official semantics are under active development, and the fragment we are considering in Creusot has unsurprising semantics. MIR is a call-by-value language; calling a function with an argument of type τ requires handing ownership of the value to the callee. This remains true for pointer types: calling a function that takes a `&mut T` argument transfers ownership of the borrow. A *place* evaluates to an address in memory reachable from a local variable. When a move operand is used, the source of the move is *uninitalized*. Using a *copy* operand, the bytes of the variable are duplicated into the destination, leaving the source intact. By convention, MIR uses the variable `_0` to hold the function's return value.

Example: Incrementing a Borrow To illustrate the syntax and semantics of MIR, we consider the following simple `incr` function.

```
1 fn incr(_1 : &mut i32) {
2   let mut _0 : ();
3   let _1 : &mut i32;
4   BB0 {
5     *_1 = (copy *_1) + 1;
6     _0 <- ();
7     return;
8   }
9 }
```

Upon entry, it copies the value pointed to by `i32`, adds one, and then writes back into `*_1`. The *place* `*_1` occurs on both the left and right-hand sides of the assignment, but all places occurring as rvalues must be prefixed by either `move` or `copy`. Only `Copy` types may occur behind a `copy` operator; the MIR type checker ensures this property.

4.2 The MLCFG language

Rather than translate directly from MIR into WhyML, Creusot makes an intermediate step by targetting MLCFG. The MLCFG language is syntactic sugar over WhyML, allowing inputting programs in the form of *control-flow graphs* (CFGs) rather than classic functional programs. This is a natural fit for MIR, which is already structured as a CFG and allows Creusot to focus on other aspects of the translation. We present the syntax of MLCFG and the WhyML fragment we use in Figure 4.2.

$$
\begin{aligned}
\mathit{Expr} \ni e &::= x \mid n \mid e \oplus e \mid \mathtt{let}\ \rho = e\ \mathtt{in}\ e \mid C\ \overline{e} \mid (e, e) \mid e_{.0} \mid e_{.1} \\
&\quad \mid \mathtt{match}\ e\ \mathtt{with}\ \overline{\rho \to e}\ \mathtt{end} \mid \mathtt{any} \\
\mathit{Term} \ni t &::= x \mid n \mid t \oplus t \mid t = t \mid t \wedge t \mid t \vee t \mid t \Rightarrow t \mid \forall\, x, t \mid \exists\, x, t \\
&\quad \mid \mathtt{let}\ \rho = t\ \mathtt{in}\ t \mid C\ \overline{t} \mid \mathtt{match}\ t\ \mathtt{with}\ \overline{\rho \to t}\ \mathtt{end} \\
\mathit{Pattern} \ni \rho &::= C(\overline{\rho}) \mid x \mid \_ \\
\mathit{Statement} \ni S &::= x \leftarrow e \mid x \leftarrow \mathtt{call}\ \ \phi(\overline{e}) \mid \mathtt{assert}\ t \mid \mathtt{assume}\ t \mid \mathtt{invariant}\ t \\
\mathit{Terminator} \ni T &::= \mathtt{goto}\ \ \mathtt{BB}_k \mid \mathtt{switch}\ e\ \mathtt{with}\ \overline{\rho \to \mathtt{BB}_k} \mid \mathtt{return}\ e \mid \mathtt{absurd} \\
\mathit{BasicBlock} \ni B &::= \mathtt{BB}_k\ \{\overline{S};\ T\} \\
\mathit{VarDecl} \ni V &::= \mathtt{var}\ x : \tau \\
\mathit{Cfg} &::= \mathtt{let\ cfg}\ \phi\ (\overline{\mathtt{b} : \tau}) : \tau \\
&\qquad \mathtt{requires}\ t \\
&\qquad \mathtt{ensures}\ t \\
&\qquad = \overline{V}\ \overline{B}
\end{aligned}
$$

Figure 4.2: Syntax of MLCFG

4.2.1 Syntax

We let identifiers (x), constructor names (C) and block identifiers (k) range over an infinite set. The class *Expr* contains arithmetic, comparisons, constructors, let bindings, and match expressions. These are used as program expressions in MLCFG to perform individual atomic operations. The class *Term* describes *logical terms* used in specifications; it includes arithmetic, constructors, let bindings, and match expressions but also universal and existential quantifiers and propositional operators for conjunction, disjunction, and implication. The class of *Pattern* includes standard functional pattern matching forms: constructor patterns, variable bindings, and wildcards.

Now we consider the syntax unique to MLCFG; programs are structured as a series of basic blocks consisting of zero or more statements followed by a single terminator. A statement (*Statement*) can either evaluate an expression and store it in a local variable, assert the truth of a term, or assert a cycle invariant. A cycle invariant works like a loop invariant in a structured language, except it refers to *cycles* in the graph rather than while-loops. A terminator can either unconditionally jump to a block, conditionally jump, return from the function, or assert unreachability. A basic block (*BasicBlock*) is a series of statements followed by a terminator. Variable declaration (*VarDecl*) introduces a new variable whose scope ranges over the whole function. Finally, an MLCFG function (*Cfg*) assembles these components; the signature consists of a name, arguments, and a contract, while the body is a series of variable bindings followed by a series of blocks.

Informal Semantics for MLCFG As MLCFG is defined purely as an alternate syntax for WhyML, its semantics are inherited from WhyML. Apart from generalities, we will not present the semantics of MLCFG here: it is call-by-value and supports mutable regions, though we will not use those in Creusot. In Figure 4.3, we give an example MLCFG program for calculating Fibonacci numbers.

```
1   let cfg fib (n : int) =
2     requires { n >= 0 }
3     var x : int; var y : int; var z : int; var t : int;
4     BB0 { x <- n; y <- 1; z <- 0;
5       goto BB0
6     }
7     BB1 { switch x with
8       | 0 -> goto BB2
9       | _ -> goto BB3
10    };
11    BB2 { return y };
12    BB3 { t <- y;
13      y <- y + z;
14      z <- t;
15      x <- x - 1;
16      goto BB1
17    }
```

Figure 4.3: MLCFG program for computing Fibonacci numbers

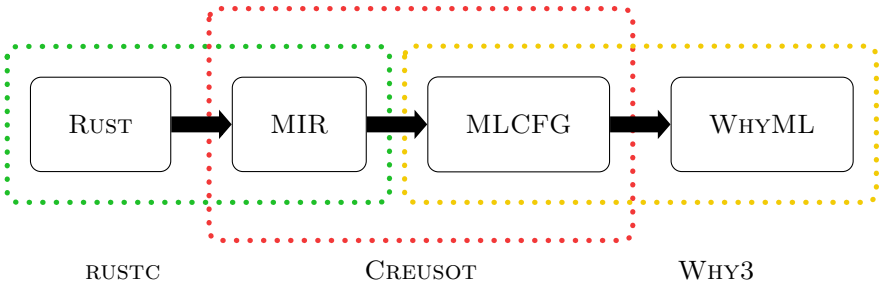


Figure 4.4: The verification pipeline of Creusot

4.3 Translation from Rust to MLCFG

Creusot is a toolchain for verifying Rust programs operating on the MIR level. It hooks into the Rust compiler to extract the MIR form of programs and translates those to MLCFG, an input language for Why3. In turn, Why3 will translate MLCFG to its internal language and then pass it to various backends. The connection between these different stages is recapitulated in Figure 4.4. The primary contribution of this chapter is the translation from MIR to MLCFG. At the very core of Creusot is the *prophetic* translation of mutable borrows, pioneered by RustHorn [69]. This model represents mutable borrows as pairs of a current and final (prophetic) value. When a borrow is created, the prophecy is given to the *lender*, and when the borrow expires, that prophecy is then *resolved*, fixing a specific value for the prophecy. This is made possible by the ownership typing of Rust: mutable borrows have exclusive permissions to mutate their borrowed resource, meaning if we have `b = &mut a`, then only `b` can modify the value of `a`, and when `b` is thrown away, we *know* that `a` must have whatever the final value of `b` was. In Creusot, this approach is encoded through *any/assume* non-determinism: `any` is used to create a prophecy, and `assume` is used to resolve it.

To perform this encoding, Creusot's translation operates in two phases: the first is a dataflow analysis used to identify the *resolution points* of every borrow, and the second is a syntax-directed translation of MIR to MLCFG, which uses the dataflow analysis to insert resolutions at the appropriate positions. On top of this kernel, Creusot also includes several extensions to support more idiomatic Rust code; in particular, we support *traits*, *generic functions*, and closures.

4.3.1 Interpretation of Rust Types

Before explaining how to translate Rust code, it is essential to understand how to translate *types*. Each Rust type will be interpreted as a specific MLCFG type; in most cases, it is straightforward, with the only exceptions being for *pointer types*. Base types are given their natural interpretations:

$$\lfloor \texttt{bool} \rfloor \triangleq \texttt{bool} \qquad \lfloor \texttt{u32} \rfloor \triangleq \texttt{uint32} \qquad \lfloor \texttt{f32} \rfloor \triangleq \texttt{float32} \qquad \ldots$$

Creusot always uses machine arithmetic to interpret Rust integer types. Additionally, for float types, we use Why3's theories for IEEE floats. Type constructors are given their natural interpretation: `structs` as products and `enums` as sums.

$$\lfloor \texttt{struct X(T, U)} \rfloor \triangleq \texttt{type X = X } \lfloor \texttt{T} \rfloor \ \lfloor \texttt{U} \rfloor \qquad \lfloor \texttt{enum X \{ T, U \}} \rfloor \triangleq \texttt{type X = I } \lfloor \texttt{T} \rfloor \ \texttt{I } \lfloor \texttt{U} \rfloor$$

The interpretation of pointers is how Creusot avoids the need for an explicit heap model.

$$\lfloor \texttt{Box<T>} \rfloor \triangleq \lfloor \texttt{T} \rfloor \qquad \lfloor \texttt{\&'a T} \rfloor \triangleq \lfloor \texttt{T} \rfloor \qquad \lfloor \texttt{\&'a mut T} \rfloor \triangleq (\lfloor \texttt{T} \rfloor, \lfloor \texttt{T} \rfloor)$$

Boxes and immutable borrows are interpreted as their underlying type, while mutable borrows are interpreted as a pair of the underlying type and a prophecy. Mutations to mutable borrows are interpreted as local updates to the *current* value. When the borrow is dropped, the prophecy is *resolved*, synchronizing the value with the original lender. This approach is made possible by the *ownership typing* of Rust, in particular, the uniqueness of *mutable borrows* combined with the existence of *lifetimes* for these borrows, ensuring prophecies are always well-formed.

4.3.2 Translation of MIR

The translation consists of five families of rules, all parameterized by a MIR typing context $\mathbf{T}$ binding locals to types. The simplest family is for *place reading* ($\mathbf{T} \vdash \texttt{p} \overset{\text{pl}}{\hookrightarrow} e$), found in Figure 4.5, accesses the value in a MIR place. This operation distinguishes accesses to *mutable borrows* from those to other pointer types. The dual *place writing* judgment ($\mathbf{T} \vdash S \hookleftarrow \texttt{p}/e$) writes an MLCFG expression e into a MIR place $\texttt{p}$, producing a MLCFG statement S encoding the update. In Figure 4.6 operand judgments ($\mathbf{T} \vdash \texttt{o} \overset{\text{op}}{\hookrightarrow} e/\overline{S}$) use the place reading judgments to translate *operands* into pure expressions. The operand jugements return a list of statements that clear any places that were *moved* by the operand. The translation of RValues ($\mathbf{T} \vdash \texttt{r} \overset{\text{rval}}{\hookrightarrow} e/\overline{S}$) is also found in Figure 4.5. As RValues contain operands, they forward the auxiliary statements the operands produce. The statement judgment ($\mathbf{T} \vdash \texttt{S} \overset{\text{stmt}}{\hookrightarrow} \overline{S}$) handles assignments. The final family in Figure 4.7 is for terminators ($\mathbf{T} \vdash \texttt{T} \overset{\text{term}}{\hookrightarrow} \overline{S}; T$) includes match expressions, function calls, and Rust assertions.

During this translation, we interpret MIR locals, blocks, and function names as MLCFG variables, basic blocks, and functions of the same name.

Simple example

To illustrate the translation, we will consider the following simple example:

```
1  fn main(x, y) -> ()
2  let _0 : ()
3  let mut x : (T, bool)
4  let y : (u32, T)
5  BB0 {
6    x.0 = move y.1;
7    _0 <- ();
8    return
9  }
```

```
1  let cfg main x y =
2  var x : (T, bool) = x;
3  var y : (u32, T) = y;
4  BB0 {
5    x <- let (_, a) = x in
6      (let (_, b) = y in (b, a));
7    y <- let (a, _) = y in (a, any);
8    return ()
9  }
```

On line 6, we *move* the second component of `y` into `x`'s first component. The expressions `x.0` and `y.1` are *place expressions* and are used to access the first and second components of the tuple. The `move` operand allows us to use a place expression on the right-hand side of an assignment by specifying the desired semantics for the read, either by `move` or `copy`.

The resulting MLCFG on the right-hand side is obtained by applying the rules S-Assign with R-Use and Op-Move. Place expressions are interpreted as functional lenses: places on the left-hand side of an assignment become setters, while those on the right are getters. Because we *move* `y.1`, after updating `x`, we empty the corresponding place in `y` by assigning it a fresh value `any`.

Place Reading $\boxed{\mathbf{T} \vdash \mathtt{p} \overset{\text{pl}}{\hookrightarrow} e}$

$$\textsc{Rd-Deref-Mut}\ \frac{\mathbf{T} \vdash \mathtt{p} \overset{\text{pl}}{\hookrightarrow} e \qquad \mathbf{T} \vdash \mathtt{p} : \mathtt{\&'a\ mut}\ \tau}{\mathbf{T} \vdash \mathtt{*\ p} \overset{\text{pl}}{\hookrightarrow} e_{.0}} \qquad \textsc{Rd-Deref-Imm}\ \frac{\mathbf{T} \vdash \mathtt{p} \overset{\text{pl}}{\hookrightarrow} e \qquad \mathbf{T} \vdash \mathtt{p} : \mathtt{\&'a}\ \ \tau}{\mathbf{T} \vdash \mathtt{*\ p} \overset{\text{pl}}{\hookrightarrow}\ \ e}$$

$$\textsc{Rd-Field}\ \frac{\mathbf{T} \vdash \mathtt{p} \overset{\text{pl}}{\hookrightarrow} e}{\mathbf{T} \vdash \mathtt{(p\ as}\ C\mathtt{)}.f \overset{\text{pl}}{\hookrightarrow}\ \mathtt{let}\ C\mathtt{(..,}f\mathtt{,..)\ =}\ e\ \mathtt{in}\ f} \qquad \textsc{Rd-Index}\ \frac{\mathbf{T} \vdash \mathtt{p} \overset{\text{pl}}{\hookrightarrow} e}{\mathbf{T} \vdash \mathtt{p[}\ \ell\ \mathtt{]} \overset{\text{pl}}{\hookrightarrow}\ \mathtt{(}e\mathtt{)\ [}\ \ell\ \mathtt{]}}$$

$$\textsc{Rd-Var}\ \frac{}{\mathbf{T} \vdash \ell \overset{\text{pl}}{\hookrightarrow} \ell}$$

Place Writing $\boxed{\mathbf{T} \vdash S \hookleftarrow \mathtt{p}/e}$

$$\textsc{Wrt-Deref-Mut}\ \frac{\mathbf{T} \vdash S \hookleftarrow \mathtt{p}/(e,\ e'_{.1}) \qquad \mathbf{T} \vdash \mathtt{p} \overset{\text{pl}}{\hookrightarrow} e' \qquad \mathbf{T} \vdash \mathtt{p} : \mathtt{\&'a\ mut}\ \tau}{\mathbf{T} \vdash S \hookleftarrow \mathtt{*\ p}/e}$$

$$\textsc{Wrt-Field}\ \frac{\mathbf{T} \vdash S \hookleftarrow \mathtt{p/let}\ C(\overline{xs},\ f,\ \overline{ys})\ \mathtt{=}\ e_2\ \mathtt{in}\ C(\overline{xs},\ e,\ \overline{ys}) \qquad \mathbf{T} \vdash \mathtt{p} \overset{\text{pl}}{\hookrightarrow} e_2}{\mathbf{T} \vdash S \hookleftarrow \mathtt{(p\ as}\ C\mathtt{)}.f/e}$$

$$\textsc{Wrt-Var}\ \frac{}{\mathbf{T} \vdash \ell \leftarrow e \hookleftarrow \ell/e}$$

Figure 4.5: Translation of MIR to MLCFG: Place Reading and Writing

Operand $\boxed{\mathbf{T} \vdash \mathtt{o} \overset{\text{op}}{\hookrightarrow} e/\overline{S}}$

$$\textsc{Op-Copy}\ \frac{\mathbf{T} \vdash \mathtt{p} \overset{\text{pl}}{\hookrightarrow} p}{\mathbf{T} \vdash \mathtt{copy\ p} \overset{\text{op}}{\hookrightarrow} p/\epsilon} \qquad \textsc{Op-Move}\ \frac{\mathbf{T} \vdash \mathtt{p} \overset{\text{pl}}{\hookrightarrow} p \qquad S \hookleftarrow \mathtt{p/any}}{\mathbf{T} \vdash \mathtt{move\ p} \overset{\text{op}}{\hookrightarrow} p/S} \qquad \textsc{Op-Const}\ \frac{}{\mathbf{T} \vdash n \overset{\text{op}}{\hookrightarrow} n/\epsilon}$$

RValue $\boxed{\mathbf{T} \vdash \mathtt{r} \overset{\text{rval}}{\hookrightarrow} e/\overline{S}}$

$$\textsc{R-Use}\ \frac{\mathbf{T} \vdash \mathtt{o} \overset{\text{op}}{\hookrightarrow} e/\overline{S}}{\mathbf{T} \vdash \mathtt{o} \overset{\text{rval}}{\hookrightarrow} e/\overline{S}} \qquad \textsc{R-Ref-Imm}\ \frac{\mathbf{T} \vdash \mathtt{p} \overset{\text{pl}}{\hookrightarrow} e}{\mathbf{T} \vdash \mathtt{\&\ p} \overset{\text{rval}}{\hookrightarrow} e/\epsilon}$$

$$\textsc{R-BinaryOp}\ \frac{\mathbf{T} \vdash \mathtt{o}_1 \overset{\text{op}}{\hookrightarrow} e_1/\overline{S}_1 \qquad \mathbf{T} \vdash \mathtt{o}_2 \overset{\text{op}}{\hookrightarrow} e_2/\overline{S}_2}{\mathbf{T} \vdash \mathtt{o}_1\ \oplus\ \mathtt{o}_2 \overset{\text{rval}}{\hookrightarrow} e_1 \oplus e_2/\overline{S}_1 \cdot \overline{S}_2} \qquad \textsc{R-Aggregate}\ \frac{\forall \mathtt{o} \in \overline{\mathtt{o}}, \mathbf{T} \vdash \mathtt{o} \overset{\text{op}}{\hookrightarrow} e/M}{\mathbf{T} \vdash C\ \ \overline{\mathtt{o}} \overset{\text{rval}}{\hookrightarrow} C\ \overline{e}/\overline{M}}$$

$$\textsc{R-Ref-Mut}\ \frac{\mathbf{T} \vdash \mathtt{p}_2 \overset{\text{pl}}{\hookrightarrow} e \qquad \mathbf{T} \vdash b \hookleftarrow \mathtt{p}_1/(e, \mathtt{any}) \qquad \mathbf{T} \vdash \mathtt{p}_1 \overset{\text{pl}}{\hookrightarrow} p \qquad \mathbf{T} \vdash S \hookleftarrow \mathtt{p}_2/p_{.1}}{\mathbf{T} \vdash \mathtt{p}_1\ \mathtt{=\ \&mut\ p}_2 \overset{\text{stmt}}{\hookrightarrow} b/S}$$

Figure 4.6: Translation of MIR to MLCFG: Operands and RValues

Statement $\boxed{\mathbf{T} \vdash \mathtt{S} \overset{\text{stmt}}{\hookrightarrow} S}$

$$\text{S-Assign}\quad \frac{\mathbf{T} \vdash \mathtt{r} \overset{\text{rval}}{\hookrightarrow} e/\overline{S} \qquad \mathbf{T} \vdash p \hookleftarrow \mathtt{p}/e}{\mathbf{T} \vdash \mathtt{p} \leftarrow \mathtt{r} \overset{\text{stmt}}{\hookrightarrow} p; \overline{S}}$$

Terminator $\boxed{\mathbf{T} \vdash \mathtt{T} \overset{\text{term}}{\hookrightarrow} \overline{S}; T}$

$$\text{T-goto}\quad \frac{}{\mathbf{T} \vdash \texttt{goto BB}_k \overset{\text{term}}{\hookrightarrow} \texttt{goto BB}_k}$$

$$\text{T-switch}\quad \frac{\mathbf{T} \vdash \mathtt{o} \overset{\text{op}}{\hookrightarrow} o/\overline{S} \qquad \rho_i \text{ contains \_ for each field in } C_i}{\mathbf{T} \vdash \texttt{switch o } \overline{C \rightarrow \texttt{BB}_k} \overset{\text{term}}{\hookrightarrow} \texttt{switch } o \texttt{ of } \overline{C(\rho) \rightarrow \overline{S}\texttt{; goto BB}_k}}$$

$$\text{T-Call}\quad \frac{\forall \mathtt{o} \in \overline{\mathtt{o}},\ \mathbf{T} \vdash \mathtt{o} \overset{\text{op}}{\hookrightarrow} o/M}{\mathbf{T} \vdash \ell = \phi(\overline{\mathtt{o}}) \rightarrow \texttt{BB}_k \overset{\text{term}}{\hookrightarrow} \ell \leftarrow \texttt{call } \phi(\overline{o}); \overline{M}; \texttt{goto BB}_k}$$

$$\text{T-Assert}\quad \frac{\mathbf{T} \vdash \mathtt{o} \overset{\text{op}}{\hookrightarrow} \mathrm{o}/\overline{S}}{\mathbf{T} \vdash \texttt{assert o} \rightarrow \texttt{BB}_k \overset{\text{term}}{\hookrightarrow} \texttt{assert } o; \overline{S}; \texttt{goto BB}_k} \qquad \text{T-Return}\quad \frac{}{\mathbf{T} \vdash \texttt{return} \overset{\text{term}}{\hookrightarrow} \texttt{return \_0}}$$

$$\text{T-Unreachable}\quad \frac{}{\mathbf{T} \vdash \texttt{unreachable} \overset{\text{term}}{\hookrightarrow} \texttt{absurd}}$$

Figure 4.7: Translation of MIR to MLCFG: Rvalues and Terminators

Mutable borrows

The previous example showed how to translate trivial Rust code, but any real-world code will require dealing with *mutable borrows*. Creusot uses a prophetic encoding similar to the one presented in §2.4; each mutable borrow is encoded as a pair of a current and final value. The final value is given to the lending variable at creation. When the borrow is *resolved*, we ensure that the final value is correct and thus indirectly update the lender. This highly compositional approach allows us to handle the various borrowing operations found in Rust easily.

Creation To create a mutable borrow in MLCFG, we use R-Ref-Mut (repeated below for convenience):

$$\text{R-Ref-Mut}\quad \frac{\mathbf{T} \vdash \mathtt{p}_2 \overset{\text{pl}}{\hookrightarrow} e \qquad \mathbf{T} \vdash b \hookleftarrow \mathtt{p}_1/(e, \mathtt{any}) \qquad \mathbf{T} \vdash \mathtt{p}_1 \overset{\text{pl}}{\hookrightarrow} p \qquad \mathbf{T} \vdash S \hookleftarrow \mathtt{p}_2/p.1}{\mathbf{T} \vdash \mathtt{p}_1 \ \mathtt{=}\ \mathtt{\&mut}\ \mathtt{p}_2 \overset{\text{stmt}}{\hookrightarrow} b/S}$$

This rule does two things: first, it creates the borrow in $\mathtt{p}_1$, giving it a pair of the value currently in $\mathtt{p}_2$ and a fresh prophecy. We then prepare a statement S which will write this prophecy into the borrowed place $\mathtt{p}_2$. The rule R-Ref-Mut allows us to make arbitrary borrows, including *reborrows* (borrowing from a pre-existing borrow) and is *lifetime-independent*: it requires no information coming from the Rust borrow checker to be correctly implemented.

Resolution The second part of the borrow translation: *resolution* is more subtle. In particular, it is essential to determine the location and order in which borrows should be resolved; getting either wrong leads to unsoundness. Informally, Creusot resolves borrows when they 'fall out of scope,' but there is no notion of scope because we operate on a CFG. By combining primitive analyses, we can recreate the notion of scope and thus determine the correct location to insert resolution statements. We construct the relation $\text{IsResolved}(-) : \text{Loc} \times \text{Var} \to \mathbb{B}$ which characterizes the places at which a variable has *already* been resolved[3].

$$\text{Is-Resolved}(G) = \neg\text{Live}(G) \wedge \text{Initialized}(G)$$

The relation is the intersection between a standard liveness analysis and initialization analysis (the Rust compiler provides both). The *resolution points* are where Is-Resolved changes value from false to true. By using this analysis, we end up with a more fine-grained notion of scope than mere lexical scope; consider the following example:

```
1  let mut x = 0;
2  let y = &mut x;
3  * y = 10;
4  assert!(x == 10);
5  x += 1;
6  assert!(x > 10);
```

The *lexical* scope of `y` continues for the entire block, while the *resolution point* of `y` ends at the first assertion. Once the resolution point of a borrow p has been determined, Creusot can instrument MLCFG code with *resolution statements*:

```
1    assume { p.0 = p.1 };
```

[3] During the preparation of this thesis, the characterization of resolution points was changed to accommodate the presence of *type invariants*. As this is not the subject of the author's work, we use the older interpretation of the resolution point here.

The semantics of assume statements ensure that only traces in which we non-deterministically chose the correct value can proceed with execution, effectively constraining the past non-deterministic choice made at borrow creation.

Why not lifetimes? A reader familiar with Rust may wonder why we cannot use the lifetimes provided by the Rust compiler to determine resolution points. To understand why lifetimes are insufficient to perform resolution, suppose we have two borrows `x, y : &'a mut T` in the following example:

```
1  if b { z = x } else { z = y }
```

The lifetime ends at the end of the `if` expression, but which borrow needs to be resolved will depend on the execution of the program. While the lifetime provides an *upper bound* on where the borrow can be resolved, it is not enough to determine the exact position. It does not allow us to disambiguate the order between two borrows of the same lifetime.

Putting it into practice To better illustrate the handling of mutable borrows, let us consider the case of the following simple Rust program:

```
1  #[ensures(^x == *x && ^y == 0 || ^x == 0 && ^y == *y)]
2  fn clear_max(x: &mut u32, y: &mut u32) {
3    if *x > *y {
4      *y = 0;
5    } else {
6      *x = 0;
7    }
8  }
```

We want to prove that either `x` is unchanged and `y` is zero, or `y` is unchanged and `x` is zero.

First, the Rust program is lowered into MIR, shown on the left-hand side of Figure 4.8. Then, on the right-hand side, we have the output MLCFG code. Note that upon entry to `BB1`, we resolve `x`, and upon entry to `BB2`, we resolve `y`. Then, after the update to the corresponding variable, we resolve the remaining borrows before jumping to the exit block. Thus, we can prove the postcondition we desire: if we passed through `BB1`, then `x` is unchanged and `y` is zero, and if we passed through `BB2`, then `y` is unchanged and `x` is zero.

Generalized Resolution

We have shown how Creusot handles the resolution of simple mutable borrows. However, we cannot yet resolve the borrows in a variable of type `Vec<&mut T>`. To address this, we would like to generalize the resolution of borrows to assume an arbitrary predicate depending on a type `T`. Then, we could ensure that resolving a vector resolves every element in the vector:

$$\text{resolve}_{\texttt{Vec<\&mut T>}}(v) \triangleq \forall\, i, 0 \leq i < |v| \to \text{resolve}_{\texttt{T}}(v_i)$$

In Rust, to define properties in terms of types, we typically use traits, and thus, in Creusot, we introduce a `Resolve` trait for this reason:

```
#[ensures(
      ^x == *x && ^y == 0
  || ^x == 0 && ^y == *y
)]
fn clear_max(x, y)
let _0 : ()
let x : &mut u32
let y : &mut u32
let _1 : bool
BB0 {
  _1 <- *x > *y;
  switch _1 {
  | true -> BB1
  | false -> BB2
  }
}
BB1 {
  *y <- 0;
  goto BB3
}
BB2 {
  *x <- 0;
  goto BB3
}
BB3 {
  _0 <- ();
  return
}
```

```
let cfg main x y =
  ensures {
       x.1 = x.0 && y.1 = 0
    || x.1 = 0 && y.1 = y.0
  }
var x : (u32, u32) = x;
var y : (u32, u32) = y;
var _1 : bool;
BB0 {
  _1 <- x.0 > y.0;
  match _1 with
  | True -> goto BB1
  | False -> goto BB2
}
BB1 {
  assume { x.0 = x.1 };
  y <- (0, y.1);
  assume { y.0 = y.1 };
  goto BB3
}
BB2 {
  assume { y.0 = y.1 };
  x <- (0, x.1);
  assume { x.0 = x.1 };
  goto BB3
}
BB3 {
  return ()
}
```

Figure 4.8: MIR representation of `clear_max` and its MLCFG translation.

```
#[trusted] trait Resolve {
  #[predicate] fn resolve(self) -> bool;
}

#[trusted] impl Resolve for T {
  #[predicate]
  default fn resolve(self) -> bool {
    true
  }
}
```

This trait definition uses the `#[trusted]` annotation, Creusot's equivalent to Rust's `unsafe` keyword. While `unsafe` is used to indicate that the implementation of a trait may violate memory safety, `#[trusted]` indicates that the implementation of a trait may violate logical soundness. The following implementation would allow us to prove `false` but violates the contract the library and author and Creusot have passed:

```
struct Evil;

#[trusted] impl Resolve for Evil {
  #[predicate] fn resolve(self) -> bool {
    false
  }
}

#[ensures(false)]
fn boom(x: Evil) {}
```

To help users avoid jeopardizing the soundness of their verified code, Creusot offers a *derive macro* for the `Resolve` trait, which is guaranteed to generate a sound implementation for the types it is applied to.

We use Rust's experimental *specialization* to define `Resolve`, providing a default implementation for every type. The implementation of `resolve` on `T` is marked as `default`, meaning that we are allowed to write alternative instances that refine this definition, so long as the refining implementation is 'more specific.' For example, we can refine the implementation on `T` by providing one for `&mut T`:

```
#[trusted] impl Resolve for &mut T {
  #[predicate]
  fn resolve(self) -> bool {
    pearlite! { ^ self == * x }
  }
}
```

The usage of specialization is critical to the usability of Creusot to verify real Rust code; we would otherwise be required to leak `Resolve` into the bounds of every function. Using specialization, we can resolve variables of any type, extending the language with a new primitive operation.

4.3.3 Handling polymorphism

To encode Rust-level polymorphism, Creusot uses the *module system* (introduced in § 2.2.1) of Why3. Each function becomes an independent module in this approach, and it *clones* any symbols it needs into its local context. This approach allows Creusot to control each function's proof context precisely and simplifies encoding traits in § 4.3.4.

However, modules are not a silver bullet, clones introduce the problem of *generativity*. A clone is equivalent to copy-pasting the cloned module source with the relevant substitution applied. This becomes a problem if we accidentally clone the same symbol twice like in the example below:

```
1  module Cmp
2    type t
3    function cmp : t -> t -> ordering
4  end
5  module User
6    clone Cmp with type t = int as Cmp0
7    clone Cmp with type t = int as Cmp1
8
9    (* unprovable *)
10   goal : forall i . Cmp0.cmp i = Cmp1.cmp i
11 end
```

The clones performed on lines 6 and 7 introduce two distinct symbols `cmp`; their results and types are incomparable. If `cmp` corresponds to a Rust function, this is a problem: in Rust two identical instantions of the same function should be identical. This problem can become more problematic when working with *nested* definitions. In the example below, though both `UserA` and `UserB` use `cmp` instantiated with `int`, the postcondition of `userb` is unprovable.

```
1  module UserA
2    clone Cmp with type t = int as Cmp0
3    val usera a b
4      ensures { Cmp0.cmp a b = Cmp0.cmp b a}
5  end
6  module UserB
7    clone Cmp with type t = int as Cmp0
8    clone UserA as UserA
9    val userb a b = UserA.usera a b
10     ensures { Cmp0.cmp a b = Cmp0.cmp b a}
11 end
```

To work around this problem, we show Why3 that there is only one `cmp` for `int` by applying an appropriate *substitution* to clones and deduplicating identical clones.

Clone Graph For each Rust function `f`, we construct a *clone graph* in which nodes are instantiations of Rust functions (pairs of a function and type substitution) and edges represent dependencies. An edge $a \to b$ means that b was used inside of a. The substitutions are all relative to the root function. We perform an in-depth traversal of the call graph to construct the graph, adding dependency edges and nodes as needed. Once the graph has been generated, we can perform a topological traversal to generate the sets of clones for `f`; each dependency edge indicates a substitution that must be provided to guarantee sharing.

Frequently, it is necessary to hide the body of a Rust definition from its clients. To make this possible, we translate Rust functions using *two modules*, an 'interface' module containing only the symbol and its contract and a 'body' module containing its definition. An interface module only refers to further interface modules; it does not attempt to perform substitutions on its symbols; those will be formed by the root function using this module. When constructing the clone graph, we use the appropriate 'view' of the Rust symbol; by using an interface module, we can avoid introducing dependencies on symbols that appear exclusively in the body.

This approach to handling polymorphism has the benefit of eliminating polymorphism from Why3, all usages of types monomorphic instances of types. Avoiding polymorphism allows Why3

```
trait Index<I> {
  type Item = T;

  #[predicate] fn in_bounds(self, i : I) -> bool;

  fn index(&self, ix: I) -> &Self::Item;
}

impl<T> Index<usize> for Vec<T> {
  type Item = T;

  #[predicate] fn in_bounds(self, i : usize) -> bool { i < self@.len() }

  #[requires(ix < self@.len())]
  fn index(&self, ix: usize) -> &Self::Item {
    &self[ix]
  }
}
```

Figure 4.9: A simplified `Index` and accompanying implementation

to use a more efficient encoding into SMT.

4.3.4 Translating traits declarations

The cloning mechanism used to handle polymorphism in Creusot naturally extends to handling traits and their functions. Rather than considering traits as objects in their own right, Creusot only considers the individual items making up the trait declaration: the functions, constants, or associated trait types. These items can be translated in the same way as any other Rust item. When code uses a method a trait provides, Creusot clones that *specific* method, not the trait itself. Creusot attempts to resolve trait methods at each call-site, replacing the generic method with a concrete method provided by an implementation, if possible.

4.3.5 Translating traits implementations

Traits in Creusot can be given contracts, including logical functions and default definitions. Implementations must then *refine* those contracts and ensure they remain correct, even if they only changed some of the default definitions. This is a form of *behavioral subtyping* ; we must ensure that implementations weaken preconditions and strengthen the postconditions. For each implementation, we generate refinement obligations verifying this property.

To illustrate this, in Figure 4.9, we have a simplified `Index` trait and an implementation for `Vec<T>`. For this trait implementation to be valid, we must show that it refines the trait contract. We achieve this by generating a module for each implementation containing the trait contract and the implementation contract, then proving that the implementation contract refines the trait contract. In the case above, we would generate the following refinement condition:

```
  self.in_bounds(ix) ==> ix < self@.len()
```

More generally, for a trait method $\{P\}\, f\, \{Q\}$, and an implementation $\{P'\}\, f'\, \{Q'\}$, we generate the following goal:

$$\forall\, \overline{a},\ P(\overline{a}) \to (P'(\overline{a}) \land Q'(\overline{a}) \to Q(\overline{a}))$$

where $\overline{a}$ is the parameters to f and f'.

Default methods and Specialization The Rust trait system also allows for *default* and *specializable* definitions. We can provide definitions that further trait implementations can override. When dealing with these definitions, generating the correct verification conditions is subtle: we must ensure that the default definition is correct for all possible implementations, including those that may override it. Consider the following example with a default implementation of `foo` which is used in the contract for the law `less_than_ten`

```
1  trait DefaultMethod {
2      #[logic]
3      fn foo() -> Int { 5 }
4
5      #[law]
6      #[ensures(Self::foo() <= 10)]
7      fn less_than_ten() { }
8  }
9
10 impl DefaultMethod for () {
11     #[logic]
12     fn foo() -> Int { 15 }
13 }
```

Contracts are proved at the definition site, meaning that for `less_than_ten` we prove it at the trait declaration site. Since later implementation on line 10 doesn't redefine the method, we won't generate fresh proof obligations for it. However, that implementation does change the definition of `foo`, which is used in the contract for `less_than_ten`.

Creusot treats all default and specializable methods as *opaque* during verification, forcing proofs to reason generically over the possible implementations of these methods. This ensures that the correctness of `less_than_ten` cannot depend on the implementation of `foo`.

4.3.6 Closures in Rust

Building off our support for traits and polymorphism, closures pose no challenge to Creusot. However, interpreting their specifications is slightly more subtle (§ 4.4.1).

4.4 Translation of Pearlite

We have established how to translate Rust programs to WhyML but have not yet attacked the question of their *specifications*. While the interpretation of Pearlite is simple in WhyML, embedding it into Rust is another matter. The primary difficulty is the *ownership typing* of Rust. Pearlite terms denote formulas in a *classical*, *first-order* logic; this means that within Pearlite, we are free to duplicate values, even mutable borrows. This flexibility is essential for giving compact specifications and allows for an efficient encoding into SMT logic. If Pearlite were subjected to the ownership discipline of Rust, it would be impossible to state simple properties like the reflexivity of equality, and would have wide-ranging consequences across all of Creusot's logic, including forbidding the usage of multiple contract clauses which involve a non-copy variable.

Embedding Pearlite functions and terms into Rust programs requires disabling the Rust borrow checker and move checker. Since both of these operations are performed on the MIR representation of a function, we ensure that Pearlite code is never translated to MIR. Instead, Creusot recovers *THIR*, an intermediate form used by the Rust compiler before MIR, and translates that into WhyML. As the Rust compiler will request the MIR of all functions anyway, we also patch the compiler to emit an empty MIR for Pearlite functions, avoiding spurious borrow-checking errors.

With few exceptions, the translation of Pearlite is straightforward; each syntactic case is lowered to its WhyML equivalent: quantifiers to quantifiers, implication to implication, etc. The *current* operator (`* x`) is lowered either to the current value of a mutable borrow or erased (for immutable borrows and boxes). The *final* operator (`^ x`) is lowered to the *final* value of a mutable borrow and is unusable for any other type.

4.4.1 Translating the old pseudo-function

In Pearlite specifications, the `old` pseudo-function can only appear in the contracts of closures. It is used to describe the modifications made by a closure to its captured variables; the only place in Creusot's interpretation of Rust where a notion of post-state exists. Because a `old` is used to describe mutation, it can only be used in the contracts of `FnOnce` or `FnMut` closures, and only for variables captured by mutable borrow. The `old` pseudo-function has a different desugaring, based on whether the closure is being treated as `FnOnce` or `FnMut`, accounting for the extra layer of indirection introduced by `call_mut`.

Consider the following code:

```
1  let mut x = 0;
2  let b = vec![b];
3  let mut c = #[ensures(x == old(x) + 1)] || {x +=  1; drop(b)};
4  c();
5  assert_eq!(x, 1);
```

We use a vector `b` to force the closure to be `FnOnce` rather than `FnMut`. By dropping `b` it becomes impossible to call `c` again. However, the capture `x` is taken by mutable borrow, and `x` is usable after `c` has been disposed of. The closure's environment is compiled to a type resembling:

```
1  struct Closure<'a> { x : &'a mut i32, b : Vec<i32> }
```

To compile the closure contract, we must interpret `x == old(x) + 1` in terms of this closure environment. We translate the `x` to `^self.x` and `old(x)` to `*self.x`, giving us the following contract: `^self.x == *self.x + 1`.

For an `FnMut` closure the interpretation of `old` changes, consider the following example.

```
1  let mut x = 0;
2  let mut c = #[ensures(x == old(x) + 1)] || {x +=  1};
3  c();
4  c();
5  assert_eq!(x, 2);
```

The closure `c` is now `FnMut`, and we call it twice. The closure environment is now compiled to a type resembling:

```
1  struct Closure<'a> { x : &'a mut i32 }
```

The contract must be elaborated to function taking `Closure<'a>` by mutable borrow (as `call_mut` does). In this circumstance, we interpret `old(x)` as `**self.x`, the value of the capture `x` at the entrance of the call. An occurence of `x` would instead be interpreted as `* (^self).x`, the value of `x` held in `self` when the borrow of `self` expired, producing the contract:

```
*(^self).x == *(*self).x + 1
```

Combined with the *unnesting* § 3.6.1 predicate of the closure, this contract allows us to prove that successive calls to the same closure increment the same counter

4.4.2 Logical Reborrowing

A significant complication in the lowering of Pearlite to WhyML is a feature called *logical reborrowing*. Rust allows *reborrowing* portions of borrows to composite types, for example, given `x : &mut (T, U)`, the expression `&mut x.0` will have type `&mut T`. This feature is critical to the ergonomics of borrows in Rust. By analogy, *logical reborrowing* allows the same functionality to be used in specifications. Consider a function like the following:

```
fn proj<T, U>(x : &mut (T, U)) -> &mut T {
  &mut x.0
}
```

With the functionality we have seen until now, we could give it the following specifications:

```
#[ensures(^result == (^x).0)]
#[ensures(*result == (*x).0)]
```

This specification is unsatisfactory; it is unnecessarily repetitive and exposes the implementation details of borrows in Creusot. In more complex situations, this can quickly become unmanageable and sometimes not directly expressible, requiring quantifiers. With logical reborrowing, we can directly state the property we meant to express as the following:

```
#[ensures(result == &mut x.0)]
```

This is a simple example, but in § 6.2.2, we will see more complex uses of logical reborrowing.

4.4.3 Correspondence between Rust and Pearlite semantics

The *logical reborrowing* we introduced has an unexpected issue. It can sometimes have different behavior than the equivalent Rust expression. This tension is introduced by Pearlite allowing `*x` as an expression, while Rust would ordinarily complain about dereferencing borrows in this manner. The expression `&mut ** (x : &mut &mut T)` has a value equivalent to `(**x, ^*x)` in Pearlite but `(**x, *^x)` in Creusot's interpretation of the Rust code. Having identical syntax with subtly different semantics is a recipe for confusion. The overlap is not limited to reborrowing: Pearlite functions are *total*, meaning that machine addition is well-defined even for values that would overflow.[4] Worse yet, the symbol `==` has different interpretations: in Rust, it just corresponds to an implementation of `PartialEq::eq`, which *should* be an equivalence relation, while in Pearlite, it corresponds to mathematical set-equality.

An interesting question is raised: if a syntactic expression is valid in both Rust and Pearlite, should they have the same semantics? The roles of Pearlite and Rust are different, and they each operate under different constraints. In particular, Pearlite needs to be efficiently translatable to Why3 and SMT, which both use total functions and distinguish equality as a special symbol.

Unifying the semantics of Rust and Pearlite would have many advantages. Users would be able to apply their understanding of Rust to Pearlite expressions. It could feasibly allow interoperation with other Rust verifiers as every verifier must agree on the semantics of Rust itself. Finally, it would also allow defining *pure* functions, simultaneously Rust and Pearlite, a pattern which occurs frequently and requires duplication today.

[4] Currently, Pearlite does not support machine arithmetic at all; all operations must be performed on their mathematical counterparts.

4.5 From MLCFG to WhyML

At the start of this chapter, we stated that Creusot targets a *pure*, *functional* language, and yet we have just presented semantics for an imperative language with a local mutable state. Of course, MLCFG is a very *mild* imperative language; the only mutation is to local variables, and no reference-taking operations could cause aliasing or indirection. MLCFG programs can be given denotations as WhyML programs, making it evident that MLCFG is nothing more than a convenient syntax for WhyML. This has the added benefit of allowing us to reuse Why3's existing infrastructure for verification, avoiding the need to implement a precondition calculus of our own. While we will not present the complete translation to WhyML, we will highlight two crucial points: control-flow reconstruction and 'subregion' analysis.

4.5.1 CFG reconstruction

Prior work [7, 33, 75] shows how to generate verification conditions for unstructured control-flow graphs (CFGs) through an extended WP calculus. While these approaches handle the translation of general CFGs, the generated verification conditions can be unintuitive for users. The calculus generates proof obligations for each possible path through the graph, often leading to seemingly duplicated VCs. The explainability of these conditions is also degraded: loops in source code will not lead to clearly visible 'loop invariant initialization' and 'loop invariant preservation' obligations.

In one of the great original papers of programming language theory, it was established that programs can be reconstructed from control flow graphs [14, 5].[5] When a CFG is *reducible*, as is the case for structured programs like Rust, this reconstruction is linear in the size of the CFG and often quite close to the original program in structure, introducing no auxiliary variables.

We implemented a variant of the algorithm described in "An Algorithm for Structuring Flowgraphs" [5] as a backend for MLCFG. Our implementation is based on the notion of *Bourdoncle Orderings* [18], which conveniently capture the nesting structure of loops in a CFG. Given the following program:

```
1  if a then foo else while b do c done
```

After compilation to CFG, the reconstruction will provide the following program:

```
1  if a then
2    foo
3  else
4    while true do
5      match b with
6      | True -> c
7      | False -> break
8      end
9    done
```

While not syntactically identical, the reconstructed program is semantically equivalent and is quite close to the original program in structure. More importantly, the verification conditions generated from the reconstructed program are more intuitive to end users when compared to direct WP calculation on the CFG along the lines of Nguyen [75].

[5] While true, this specific result is also a classic example of a 'Folk Theorem' [42], as the original result by Böhm and Jacopini [14] merely demonstrates that every CFG can be expressed as a series of sequences, choices, and loops. Over the following years, many papers refined this result, and in 1977 Baker [5] provided the proper description of an algorithm that produces *structurally equivalent* programs to the input CFG.

4.5.2 Subregion analysis

Why3 uses region typing to determine which variables are affected by loops and thus need to be havok'ed. Our translation cannot take advantage of these regions, and as a result, Why3 is overly conservative and forgets information about variables that were not modified. This issue commonly occurs when manipulating mutable borrows like the following example:

```
1  fn foo(v : &mut Vec<T>) {
2    let old_v = ghost! { v };
3    let mut i = 0;
4    #[invariant(^old_v == ^v)]
5    while i < v.len() {
6      v[i] = 0;
7      i += 1;
8    }
9  }
```

We must add the invariant on line 4 to remember that the prophecy of `v` has not changed despite never modifying the prophecy of `v` in the generated MLCFG code.

To address this, we use a simple static analysis, which infers the unmodified portions of variables and makes that information available to the verification condition generation. This analysis is not specific to Creusot or MLCFG, though it is only enabled by default in Creusot. The analysis operates within loops, identifying which paths reachable from a variable are identical at the entrance and exit of the loop and in those cases, instrumenting the loop to preserve the value of these paths.

4.6 Related Works

Verus [58] is a Rust verifier descended from the work on Linear Dafny [65], and its translation is similar to Creusot. However, Verus does not fully account for mutable borrows; instead, specific support is provided for mutable borrows in the argument position and functions like `index_mut` cannot be handled in the Verus approach. Rather than attempting to support the whole Rust language, Verus targets the efficient verification of low-level concurrent software and has added specific features to help engineers and verifiers achieve this. The essential addition of Verus is a stratified logic: rather than having a single level of non-linear classical ghost code for their specifications, they introduce an additional *linear ghost-code* layer (called `#[proof]` mode). In particular, `#[proof]` mode allows the creation of linear ghost tokens, which can be used to encode protocols or to share access to cells *s*afely à la GhostCell [95]. Another critical feature of Verus is *performance* and the *predictability* of that performance. Significant effort has been put into tuning the SMT queries generated by Verus and avoiding features that can cause solvers to take unpredictable amounts of time.

Prusti [4] is another deductive verifier for Rust based on the Viper separation logic platform. It does not use a prophetic encoding, instead modeling ownership using *permissions*. Like Creusot, Prusti has a specification language that can be used to give contracts and invariants. Because Prusti has no notion of prophecy, it does not use the *final* operator (`^`) to specify mutable borrows, instead using *pledges*. A pledge is an assertion guaranteed to hold when the borrow expires, which is not necessarily in the function's body. When lifted into pure contracts, the semantics of Prusti specifications were designed to preserve the behavior of program assertions. In particular, arithmetic in Prusti's specifications is machine arithmetic and has to be checked for overflow. Creusot takes a different approach by using a more abstract specifications language (Pearlite), which has a

more straightforward encoding in SMT. A consequence of this difference is that Pearlite logical functions may not necessarily be executable, while Prusti's pure functions can be used in programs. While Prusti's permission system supports the common borrowing patterns of Rust, it struggles with patterns like *reborrowing in a loop* (*e.g.,* "List Index Mut" § 3.8.1), with data structures *containing borrows* like pairs of mutable borrows, or with *nested borrows*. In contrast, Creusot's translation of Rust types using prophecies for mutable borrows is general, and *compositional*: we place no restrictions on using mutable borrows or their position within types. Another noticeable difference with Prusti lies in the choice of the underlying logic. Prusti encodes specifications into separation logic and delegates verification to Viper, whereas Creusot encodes them into FOL and delegates verification to SMT solvers via Why3. Prusti chooses to verify Rust's ownership discipline with Viper. At the same time, Creusot depends on Rust's borrow checker for that, which means Creusot relies on the soundness of Rust's type system and its implementation. We believe this difference explains the significant blow-up in verification times: on simple examples, verification takes an order of magnitude more time than with Creusot. The simpler underlying logic in Creusot allows it to benefit from Why3's mature infrastructure to manage a herd of automated provers and a tactic system to provide guidance when they go astray.

Aeneas [43] is a verifier for Rust targeting interactive verification of programs in established proof assistants like F* or Coq. They also translate Rust programs to functional programs using a State-Error Monad. Instead of using prophecies, they use *backward functions* to reconstruct the value of a lender after the borrow's expiry. This approach appears to have a deep and close link to prophecies as used by Creusot, instead of using non-determinism to pull the value out of thin air, Aeneas constructs the actual *witness* of this value. The constructive approach that Aeneas takes may be better suited to interactive provers that traditionally prefer constructive logic. Aeneas also chooses to use so-called *extrinsic proofs*; all specification and proof work is done in the prover, with no annotations in Rust. While this allows them to leverage all the existing tools in the underlying prover, the proof engineer must manually sync these proofs and specifications with the Rust code as it evolves. This attests to the different audiences targeted by the tools. Aeneas seeks to enable the users of existing advanced verification tools to perform more ergonomic verification using their traditional toolkits, while Creusot seeks to bring verification to regular engineers.

Chapter 5

Soundness of the prophetic approach to Rust verification

> The work in this chapter results from a collaboration with Yusuke Matsushita, Jacques-Henri Jourdan, and Derek Dreyer and is adapted from our PLDI paper ‘“RustHornBelt: A Semantic Foundation for Functional Verification of Rust Programs with Unsafe Code”’. We elaborate upon many details not covered in the original paper.

In the previous chapter, we showed how to leverage Rust's type system to simplify the verification of *safe* Rust code. In particular, we can lean on the ownership typing and the uniqueness of mutable borrows to eliminate the need for separation logic. We demonstrated how *prophecies* can predict the mutations that will occur to a borrow and be encoded into first-order logic (FOL).

However, as we alluded, this approach has several key limitations, the first being that prophetic reasoning is limited to *safe* Rust code. In Rust, *unsafe* code is allowed to manipulate *raw pointers*, which can violate the mutable aliasing policy of the Rust type system, at least locally. However, such code is often in the implementation of safe primitives like `Vec<T>`. Consider the following implementation of the `push` [87] function from `Vec`

```
1  pub fn push(&mut self, value: T) {
2      if self.len == self.buf.capacity() {
3          self.buf.reserve_for_push(self.len);
4      }
5      unsafe {
6          let end = self.as_mut_ptr().add(self.len);
7          ptr::write(end, value);
8          self.len += 1;
9      }
10 }
```

On lines 6-8, we perform pointer arithmetic to recover a pointer to uninitialized memory, which we use to store our new vector value. However, as a safe function, `push` pledges that these operations cannot lead to undefined behavior; from the outside, we should not be able to determine the presence of unsafe code. Moreover, it should be possible to use prophetic reasoning to reason about the clients of `push`; to them we are merely adding an element to the end of the sequence already held by the vector.

However, *proving* that `push` behaves according to its prophetic specification is challenging. The proof must be able to reason about both prophecies and unsafe code and provide the link between

the two aspects.

This brings us to the second challenge for prophetic reasoning: the proof of soundness for the whole approach. While prophecies are a natural tool to write Rust specifications, it's not obvious why we should be allowed to use them; after all, they rely on being able to pull a value out of a future that has not yet occurred. Furthermore, the traditional syntactic approach to proofs of soundness is insufficient to handle unsafe code. The core of those proofs is an enumeration of syntax: we consider each syntactic form that can produce a given type and argue for its soundness. However, with unsafe code, we cannot do such an enumeration: the set of (safe) types is fundamentally open.

In this chapter, we address both these challenges by introducing RustHornBelt, a formalization of prophetic reasoning in Rust. However, the key of RustHornBelt will be its *extensible*, *semantic* proof of soundness, which can reason about unsafe code. The whole proof has been mechanized in Coq [24] using the Iris [51] separation logic framework. We will explain the high-level ideas of the proof and approach; for the full details, consult the Coq proof development[1]. In § 5.1, we present the λ_{Rust} language and type-spec system. Then, in § 5.2, we describe the Iris model of RustHornBelt, the interpretation of judgments and types, and the corresponding proof of soundness. In § 5.3, we show how to use RustHornBelt to prove specifications about unsafe code. Finally, in § 5.4, we present an evaluation of our mechanization.

5.1 The λ_{Rust} language

The Rust language is hard to study formally; its syntax is large and redundant, obscuring important and interesting semantic questions. The λ_{Rust} language is an idealized kernel language for Rust. It is modeled heavily after the *Mid-Level Intermediate Representation (MIR)* used during the compilation of Rust. During the rest of this chapter, we will use λ_{Rust} as our language of study.

5.1.1 The syntax of λ_{Rust}

λ_{Rust} programs are composed of a series of atomic instructions structured in continuation passing style. To simplify the presentation, we removed *sums* from the language, though they are in the mechanized version. The grammar of λ_{Rust} is included below; its entry point is *Instr*.

$$
\begin{aligned}
\mathit{Path} \ni p &::= \mathtt{a} \mid p.n \\
\mathit{Instr} \ni I &::= \mathtt{false} \mid \mathtt{true} \mid z \mid \mathtt{funrec}\, f(\overline{\mathtt{a}})\, \mathtt{ret}\, k := F \\
&\quad \mid p \mid p_1 + p_2 \mid p_1 - p_2 \mid p_1 \leq p_2 \\
&\quad \mid \mathtt{new}(n) \mid \mathtt{delete}(n, p) \mid {}^{*}p \mid p_1 := p_2 \\
&\quad \mid p_1 :=_n {}^{*}p_2 \\
\mathit{FuncBody} \ni F &::= \mathtt{let}\, \mathtt{a} = I\, \mathtt{in}\, F \mid \mathtt{letcont}\, k(\overline{\mathtt{a}}) := F_1\, \mathtt{in}\, F_2 \\
&\quad \mid \mathtt{newlft}; F \mid \mathtt{endlft}; F \mid \mathtt{resolve}\, p; F \\
&\quad \mid \mathtt{if}\, p\, \mathtt{then}\, F_1\, \mathtt{else}\, F_2 \mid \mathtt{assert}\, p; F \\
&\quad \mid \mathtt{jump}\, k(\overline{\mathtt{a}}) \mid \mathtt{call}\, f(\overline{p})\, \mathtt{ret}\, k
\end{aligned}
$$

Offsets n and numeric literals z range over the integers, while the indices of sums i range over the naturals. λ_{Rust} programs can be broken down into three different syntactic categories. *Paths* are either program variables $\mathtt{a}$ or offsets from one $p.n$. *Instructions* form the core of λ_{Rust}, and include base constants ($\mathtt{false}$, $\mathtt{true}$, z), arithmetic, *function definition* ($\mathtt{funrec}\, f(\overline{\mathtt{a}})\, \mathtt{ret}\, k := F$) and impure memory operations:

[1] https://gitlab.mpi-sws.org/iris/lambda-rust/-/tree/masters/rusthornbelt

- We can allocate new memory ($\texttt{new}(n)$),
- deallocate it ($\texttt{delete}(n, p)$),
- dereference it ($^*p$),
- assign untagged values ($p_1 := p_2$)
- Finally, a `memcpy`-like primitive is included ($p_1 :=_n {}^*p_2$).

Instructions are glued together using *Function Bodies*, which create local variables ($\texttt{let}\,\mathtt{a} = I\,\texttt{in}\,F$), introduce and call continuations ($\texttt{letcont}\,k(\overline{\mathtt{a}}) := F_1\,\texttt{in}\,F_2$ and $\texttt{jump}\,k(\overline{\mathtt{a}})$), create and end lifetimes ($\texttt{newlft}; F$ and $\texttt{endlft}; F$), *resolve* mutable borrows ($\texttt{resolve}\,p$), perform control flow ($\texttt{if}\,p\,\texttt{then}\,F_1\,\texttt{else}\,F_2$, assertions ($\texttt{assert}\,p$), and finally call functions ($\texttt{call}\,f(\overline{p})\,\texttt{ret}\,k$).

Differences from Rust In the interest of simplicity, λ_{Rust} has several key deviations from Rust. In particular, local variables are not mutable, unlike what can be done in Rust. Instead, we box every variable by convention, which dramatically simplifies aspects of the formalization: it means all variables are the same size, eliminating the need for mutable locals and a stack. Structurally, λ_{Rust} is organized in *continuation-passing style*; this allows a simpler formalization of the control-flow graph structure used in MIR during Rust compilation.

Differences from RustBelt As mentioned earlier, the syntax presented in this chapter excludes sums, the `match` construct, though it is present in the mechanized Coq implementation. The syntax of λ_{Rust} presented here contains a minor extension of the one found in "RustBelt: Securing the Foundations of the Rust Programming Language"[2]: we add a `assert` and a ghost `resolve` statement.

The semantics of `assert true` and $\texttt{resolve}\,p$ are the same as `skip`, while `assert false` causes execution to get stuck.

5.1.2 The λ_{Rust} type-spec system

In RustBelt [53], the authors presented a novel *semantic type system* for λ_{Rust} modeling the Rust type system and validating the 'safe abstractions' used as the foundations of Rust's safety guarantees. As a result, they demonstrated that well-typed Rust programs will not exhibit undefined behavior, even when some portions are implemented using unsafe code.

In RustHornBelt, we developed a novel 'type-spec' system that integrates verification conditions into the typing judgments of RustBelt, ensuring well-typed programs are also *well-behaved*. Judgments relate an instruction to its type and specification, written as a predicate transformer. Our system keeps the generality of RustBelt; it can still express the complex (re)borrowings found in Rust and can reason about the presence of unsafe code in a program. We begin by laying out the vocabulary to introduce our system.

Definition 4 (Representation Type)**.** *For each Rust type* T*, we define a* representation type $\lfloor T \rfloor$*, which is defined as follows:*

$$\lfloor \textbf{\textit{int}} \rfloor \triangleq \mathbb{Z} \qquad \lfloor \textbf{\textit{bool}} \rfloor \triangleq \mathbb{B} \qquad \lfloor T \times T' \rfloor \triangleq \lfloor T \rfloor \times \lfloor T' \rfloor$$

$$\lfloor \textbf{\textit{box}}\, T \rfloor \triangleq \lfloor T \rfloor \qquad \lfloor \mathcal{B}^{\alpha}_{\textbf{\textit{shr}}}\, T \rfloor \triangleq \lfloor T \rfloor \qquad \lfloor \mathcal{B}^{\alpha}_{\textbf{\textit{mut}}}\, T \rfloor \triangleq \lfloor T \rfloor \times \lfloor T \rfloor$$

[2] We strongly recommend consulting Jung et al. [52] which includes the complete set of judgments for RustBelt

In RustHornBelt, each type is associated with a *representation*, a form of refinement with peculiar handling of pointers. In particular, a mutable reference `&'a mut T`, ($\&^{\alpha}_{\mathbf{mut}}\,\mathtt{T}$ in λ_{Rust}) is represented as the pair of the current and *final* value of the pointed object. On the other hand, other pointers like `Box<T>` and shared references `&'a T` are erased and represented as their pointed objects.

Representation types are used in the specifications of individual instructions to relate the pre- and post-states logically using *predicate transformers*. Specifically, we introduce a *type-spec system* in which each typing judgment is augmented with a predicate transformer, which takes a (typed) postcondition and transforms it into a (typed) precondition.

The type-spec relation contains six different forms of judgments; we detail judgment for instructions I:

$$\mathbf{E}; \mathbf{L} \mid \mathbf{T}_0 \vdash I \dashv \mathtt{a}.\, \mathbf{T}_1 \rightsquigarrow \Phi$$

The *external* $\mathbf{E}$ and *local lifetime contexts* $\mathbf{L}$ manage lifetime information; later we use the judgment $\mathbf{E}; \mathbf{L} \vdash \alpha\ \mathsf{alive}$, to show that the lifetime α is alive (*i.e.,* has not ended).

A *type context* $\mathbf{T}$ is an ordered list of items of the form either $p \lhd \mathtt{T}$ or $p \lhd^{\dagger\alpha} \mathtt{T}$. In a type context, we primarily use the former $p \lhd \mathtt{T}$, which means that we have an object of type `T` with the name p. The latter $p \lhd^{\dagger\alpha} \mathtt{T}$ is a more *Rust-y* concept: it means that an object of type `T` assigned to p is *borrowed* (or blocked) until the *lifetime* α ends.

An *instruction* I performs a simple operation like addition $x + y$.

At the most basic level, like in RustBelt the judgment relates the typing contexts before $\mathbf{T}_0$ and after $\mathbf{T}_1$ instruction evaluation. New to the type-spec relation added in RustHornBelt is the *predicate transformer* Φ, the specification of the program fragment.

For this judgment, it has the following type:

$$\Phi\colon (\lfloor \mathbf{T}_1 \rfloor \to \mathsf{Prop}) \to \lfloor \mathbf{T}_0 \rfloor \to \mathsf{Prop}$$

It transforms a postcondition over the output $\mathbf{T}_1$ into a precondition over the input $\mathbf{T}_0$. Here, a type context's representation type $\lfloor \mathbf{T} \rfloor$ for $\mathbf{T} = \overline{p \lhd^{?} \mathtt{T}}$ (each $p \lhd^{?} \mathtt{T}$ is $p \lhd \mathtt{T}$ or $p \lhd^{\dagger\alpha} \mathtt{T}$) is defined as the ordered, heterogeneous list type $[\overline{\lfloor \mathtt{T} \rfloor}]$, consisting of the representation type of each element type `T` in the type context.

Now we consider the second significant kind of judgment, those of *function bodies* F:

$$\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F \rightsquigarrow \Phi$$

As the name indicates, a function body is used for a function's body or control flow and ends by returning from the function. This is why this judgment does not have the output type context. For this judgment, the predicate transformer Φ is typed as follows, where `T` is the return type of the function that F belongs to:

$$\Phi\colon (\lfloor \mathtt{T} \rfloor \to \mathsf{Prop}) \to \lfloor \mathbf{T} \rfloor \to \mathsf{Prop}$$

In λ_{Rust}, we can introduce new *continuations* to form complex control flows. A continuation ($\mathbf{cont}(\mathbf{L}; \overline{\mathtt{a}}.\mathbf{T}; \Phi)$) resembles a function in many respects; it has a lifetime context $\mathbf{L}$ describing the lifetimes and constraints which must be true when jumping to the continuation, it may accept arguments $\overline{\mathtt{a}}$, whose types are described in a typing context $\mathbf{T}$, along with any 'captured' variables. Compared to RustBelt, we extend continuations with a predicate transformer Φ which describes the continuation's specification. The *continuation context* $\mathbf{K}$ holds all continuations accessible when we enter F. At function entry, we only have one continuation, the *return continuation*, which only inputs the return value. The continuation context $\mathbf{K}$ associates each continuation k with a predicate transformer, which transforms the function's postcondition into a precondition for the continuation.

The return continuation is associated with the transformer $\lambda\,\Psi, [a].\Psi\,a$, asserting that the function's postcondition Ψ has to hold when applied to the return value a.

The type system includes several other kinds of judgments, primarily structural ones such as *type context inclusion*, which we can use in between instructions to perform operations such as borrowing or splitting pointers to structs into pointers on the elements of the struct. The set of judgments is included in Figures 5.1 to 5.5, but we will repeat relevant rules inline throughout the rest of the chapter. The *type context unblocking* and *continuation context inclusion* judgments are inherited from the original RustBelt work. Note that we do not include rules for sums in these figures, we also excluded the rule for subtyping functions, though all of these judgments were proved in the Coq mechanization.

Definition 5 (Syntactic Typing Judgement)**.** *We call the smallest relation engendered by the rules of Figures 5.1 to 5.5 the* type-spec relation *of* λ_{Rust}.

5.1.3 Example: Decrementing a reference

To demonstrate the function of RustHornBelt, we will type and prove the specification of a simple Rust program:

```
1 fn decr_positive(x: &mut u32) {
2   assert!(*x > 0);
3   *x -= 1;
4 }
```

The function `decr_positive` will take a mutable reference to a *positive* integer and decrement its value. It should panic otherwise.

We will call `decr_positive` from a `main` function like below:

```
1 fn main() {
2   let mut y = 43;
3   decr_positive(&mut y);
4   assert!(y == 42);
5 }
```

We wish to prove that the `main` function will terminate without failing its assertion. We will prove both functions using the RustHornBelt type-spec system to do this.

Formally, this can be written as ensuring that the following judgment holds for `decr_positive`:

$$\emptyset;\emptyset \mid \emptyset;\emptyset \vdash \texttt{decr\_positive} \dashv f.\, f \lhd \forall \alpha.\, \mathbf{fn}(\mathsf{F} : \emptyset; \&^{\alpha}_{\mathbf{mut}}\, \mathbf{int}) \to () \rightsquigarrow \lambda\,\Psi\,(x, x'),\ x > 0 \land (x' = x - 1 \to \Psi\ ())$$

In an empty context, the body of `decr_positive` types as a function taking a borrow and returning nothing and has a behavior described by the predicate transformer:

$$\lambda\,\Psi\,(x, x'),\ x > 0 \land (x' = x - 1 \to \Psi\ ())$$

The inputs of the transformer are the postcondition Ψ and the input borrow to the function (x, x') (recall that the representation of a borrow is a pair). The body of the transformer can be read as a precondition $x > 0$ along with a postcondition $x' = x - 1$.

Subtyping $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathtt{T}_1 \Rightarrow \mathtt{T}_2 \rightsquigarrow f}$

T-REFL
$$\mathbf{E}; \mathbf{L} \vdash \mathtt{T} \Rightarrow \mathtt{T} \rightsquigarrow id$$

T-TRANS
$$\frac{\mathbf{E}; \mathbf{L} \vdash \mathtt{T} \Rightarrow \mathtt{T}' \rightsquigarrow f \qquad \mathbf{E}; \mathbf{L} \vdash \mathtt{T}' \Rightarrow \mathtt{T}'' \rightsquigarrow g}{\mathbf{E}; \mathbf{L} \vdash \mathtt{T} \Rightarrow \mathtt{T}'' \rightsquigarrow g \circ f}$$

T-BOR-LFT
$$\frac{\mathbf{E}; \mathbf{L} \vdash \alpha \sqsubseteq \alpha'}{\mathbf{E}; \mathbf{L} \vdash \&_\mu^{\alpha'} \mathtt{T} \Rightarrow \&_\mu^{\alpha} \mathtt{T} \rightsquigarrow id}$$

T-UNINIT-PROD-1
$$\mathbf{E}; \mathbf{L} \vdash \lightning_{\Sigma \overline{n}} \Rightarrow \Pi \overline{\lightning_n} \rightsquigarrow \lambda a, \overline{()}$$

T-UNINIT-PROD-2
$$\Pi \overline{\lightning_n} \Rightarrow \mathbf{E}; \mathbf{L} \vdash \lightning_{\Sigma \overline{n}} \rightsquigarrow \lambda \overline{a}, ()$$

T-OWN
$$\frac{\mathbf{E}; \mathbf{L} \vdash \mathtt{T}_1 \Rightarrow \mathtt{T}_2 \rightsquigarrow f}{\mathbf{E}; \mathbf{L} \vdash \mathbf{own}_n \mathtt{T}_1 \Rightarrow \mathbf{own}_n \mathtt{T}_2 \rightsquigarrow f}$$

T-BOR-SHR
$$\frac{\mathbf{E}; \mathbf{L} \vdash \mathtt{T}_1 \Rightarrow \mathtt{T}_2 \rightsquigarrow f}{\mathbf{E}; \mathbf{L} \vdash \&_{\mathbf{shr}}^{\alpha} \mathtt{T}_1 \Rightarrow \&_{\mathbf{shr}}^{\alpha} \mathtt{T}_2 \rightsquigarrow f}$$

T-BOR-MUT
$$\frac{\mathbf{E}; \mathbf{L} \vdash \mathtt{T}_1 \Leftrightarrow \mathtt{T}_2 \rightsquigarrow id}{\mathbf{E}; \mathbf{L} \vdash \&_{\mathbf{mut}}^{\alpha} \mathtt{T}_1 \Leftrightarrow \&_{\mathbf{mut}}^{\alpha} \mathtt{T}_2 \rightsquigarrow id}$$

T-PROD
$$\frac{\forall i.\, \mathbf{E}; \mathbf{L} \vdash \mathtt{T}_i \Rightarrow \mathtt{T}_i' \rightsquigarrow f_i}{\mathbf{E}; \mathbf{L} \vdash \Pi\, \overline{\mathtt{T}} \Rightarrow \Pi \overline{\mathtt{T}'} \rightsquigarrow \lambda \overline{a_i}, \overline{f_i\ a_i}}$$

Type context unblocking $\boxed{\Gamma \vdash \mathbf{T}_1 \Rightarrow^{\dagger\alpha} \mathbf{T}_2}$

$$\emptyset \Rightarrow^{\dagger\alpha} \emptyset$$

$$\frac{\mathbf{T}_1 \Rightarrow^{\dagger\alpha} \mathbf{T}_2}{\mathbf{T}_1, p \lhd \mathtt{T} \Rightarrow^{\dagger\alpha} \mathbf{T}_2, p \lhd \mathtt{T}} \qquad \frac{\mathbf{T}_1 \Rightarrow^{\dagger\alpha} \mathbf{T}_2}{\mathbf{T}_1, p \lhd^{\dagger\alpha} \mathtt{T} \Rightarrow^{\dagger\alpha} \mathbf{T}_2, p \lhd \mathtt{T}}$$

$$\frac{\mathbf{T}_1 \Rightarrow^{\dagger\alpha} \mathbf{T}_2}{\mathbf{T}_1, p \lhd^{\dagger\alpha'} \mathtt{T} \Rightarrow^{\dagger\alpha} \mathbf{T}_2, p \lhd^{\dagger\alpha'} \mathtt{T}}$$

Continuation context inclusion $\boxed{\Gamma \mid \mathbf{E} \vdash \mathbf{K}_1 \Rightarrow \mathbf{K}_2}$

$$\frac{\mathbf{K}' \text{ is a permutation of } \mathbf{K}}{\mathbf{E} \vdash \mathbf{K} \Rightarrow \mathbf{K}'} \qquad \mathbf{E} \vdash \mathbf{K}, \mathbf{K}' \Rightarrow \mathbf{K}$$

$$\frac{\Gamma \mid \mathbf{E} \vdash \mathbf{K} \Rightarrow \mathbf{K}' \qquad \Gamma, \overline{\mathtt{a}} : \mathbf{val} \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T}' \overset{\text{ctx}}{\Rightarrow} \mathbf{T} \rightsquigarrow \Phi}{\Gamma \mid \mathbf{E} \vdash \mathbf{K}, k \lhd \mathbf{cont}(\mathbf{L}; \overline{\mathtt{a}}.\mathbf{T}; \Phi') \Rightarrow \mathbf{K}', k \lhd \mathbf{cont}(\mathbf{L}; \overline{\mathtt{a}}.\mathbf{T}'; \Phi \circ \Phi')}$$

Figure 5.1: Type-spec rules for subtyping, type context unblocking, and continuation context inclusion

Type context inclusion $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T}_1 \overset{\text{ctx}}{\Rrightarrow} \mathbf{T}_2 \rightsquigarrow \Phi}$

C-PERM
$$\frac{f \text{ is a permutation from } \mathbf{T}' \text{ to } \mathbf{T}}{\mathbf{E}; \mathbf{L} \vdash \mathbf{T} \overset{\text{ctx}}{\Rrightarrow} \mathbf{T}' \rightsquigarrow \lambda\, \Psi\ [\overline{a}],\ \Psi\ (f\ \overline{a})}$$

C-WEAKEN
$$\mathbf{E}; \mathbf{L} \vdash \mathbf{T}, \mathbf{T}' \overset{\text{ctx}}{\Rrightarrow} \mathbf{T} \rightsquigarrow \lambda\, \Psi\ (\overline{a} +\!\!+ \overline{b}),\ \Psi\ (\overline{b})$$

C-FRAME
$$\frac{\mathbf{E}; \mathbf{L} \vdash \mathbf{T}_1 \overset{\text{ctx}}{\Rrightarrow} \mathbf{T}_2 \rightsquigarrow \Phi}{\mathbf{E}; \mathbf{L} \vdash \mathbf{T}', \mathbf{T}_1 \overset{\text{ctx}}{\Rrightarrow} \mathbf{T}', \mathbf{T}_2 \rightsquigarrow \lambda\, \Psi\ (\overline{a} +\!\!+ \overline{b}),\ \Phi\ (\lambda\, \overline{b_2},\ \Psi\ (\overline{a} +\!\!+ \overline{b_2}))\ \overline{b}}$$

C-COPY
$$\frac{\mathtt{T} \text{ copy}}{\mathbf{E}; \mathbf{L} \vdash p \lhd \mathtt{T} \overset{\text{ctx}}{\Rrightarrow} p \lhd \mathtt{T}, p \lhd \mathtt{T} \ \rightsquigarrow \lambda \Psi\ [a],\ \Psi\ [a, a]}$$

C-SUBTYPE
$$\frac{\mathbf{E}; \mathbf{L} \vdash \mathtt{T} \Rightarrow \mathtt{T}' \rightsquigarrow f}{\mathbf{E}; \mathbf{L} \vdash p \lhd \mathtt{T} \overset{\text{ctx}}{\Rrightarrow} p \lhd \mathtt{T}' \rightsquigarrow \lambda \Psi\ [a],\ \Psi\ [f\ a]}$$

C-SHARE
$$\frac{\mathbf{E}; \mathbf{L} \vdash \alpha \text{ alive}}{\mathbf{E}; \mathbf{L} \vdash p \lhd \&^{\alpha}_{\mathbf{mut}} \mathtt{T} \overset{\text{ctx}}{\Rrightarrow} p \lhd \&^{\alpha}_{\mathbf{shr}} \mathtt{T} \rightsquigarrow \lambda \Psi\ [(a, a')],\ a = a' \to \Psi\ [a]}$$

C-SPLIT-OWN-1
$$\frac{\overline{\mathtt{T}} \neq [\,] \qquad \forall i.\, m_i = \sum_{j<i} \mathsf{size}(\overline{\mathtt{T}}_j)}{\mathbf{E}; \mathbf{L} \vdash p \lhd \mathbf{own}_n\, \Pi\, \overline{\mathtt{T}} \overset{\text{ctx}}{\Rrightarrow} \overline{p.m \lhd \mathbf{own}_n\, \mathtt{T}} \rightsquigarrow \lambda \Psi\ [(a_0, .., a_i)],\ \Psi\ [a_0, .., a_i]}$$

C-SPLIT-OWN-2
$$\frac{\overline{\mathtt{T}} \neq [\,] \qquad \forall i.\, m_i = \sum_{j<i} \mathsf{size}(\overline{\mathtt{T}}_j)}{\overline{p.m \lhd \mathbf{own}_n\, \mathtt{T}} \overset{\text{ctx}}{\Rrightarrow} \mathbf{E}; \mathbf{L} \vdash p \lhd \mathbf{own}_n\, \Pi\, \overline{\mathtt{T}} \rightsquigarrow \lambda \Psi\ [a_0, .., a_i],\ \Psi\ (a_0, .., a_i)}$$

C-SPLIT-BOR-IMMUT
$$\frac{\overline{\mathtt{T}} \neq [\,] \qquad \forall i.\, m_i = \sum_{j<i} \mathsf{size}(\overline{\mathtt{T}}_j)}{\mathbf{E}; \mathbf{L} \vdash p \lhd \&^{\alpha}_{\mu}\, \Pi\, \overline{\mathtt{T}} \overset{\text{ctx}}{\Leftrightarrow} \overline{p.m \lhd \&^{\alpha}_{\mu}\, \mathtt{T}} \rightsquigarrow \ \lambda \Psi\ [(a_0, ..a_i)],\ \Psi\ [a_0 :: .. :: a_i]}$$

C-SPLIT-BOR-MUT-1
$$\frac{\overline{\mathtt{T}} \neq [\,] \qquad \forall i.\, m_i = \sum_{j<i} \mathsf{size}(\overline{\mathtt{T}}_j)}{\mathbf{E}; \mathbf{L} \vdash p \lhd \&^{\alpha}_{\mathbf{mut}}\, \Pi\, \overline{\mathtt{T}} \overset{\text{ctx}}{\Rrightarrow} \overline{p.m \lhd \&^{\alpha}_{\mathbf{mut}}\, \mathtt{T}} \rightsquigarrow \lambda \Psi\ [(\overline{a_i}, \overline{a_i'})],\ \Psi\ [\overline{(a_i, a_i')}]}$$

C-SPLIT-BOR-MUT-2
$$\frac{\overline{\mathtt{T}} \neq [\,] \qquad \forall i.\, m_i = \sum_{j<i} \mathsf{size}(\overline{\mathtt{T}}_j)}{\overline{p.m \lhd \&^{\alpha}_{\mathbf{mut}}\, \mathtt{T}} \overset{\text{ctx}}{\Rrightarrow} \mathbf{E}; \mathbf{L} \vdash p \lhd \&^{\alpha}_{\mathbf{mut}}\, \Pi\, \overline{\mathtt{T}} \rightsquigarrow \lambda \Psi\ [\overline{(a_i, a_i')}],\ \Psi\ [(\overline{a_i}, \overline{a_i'})]}$$

C-BORROW
$$\mathbf{E}; \mathbf{L} \vdash p \lhd \mathbf{own}_n\, \mathtt{T} \overset{\text{ctx}}{\Rrightarrow} p \lhd \&^{\alpha}_{\mathbf{mut}}\, \mathtt{T}, p \lhd^{\dagger \alpha} \mathbf{own}_n\, \mathtt{T} \rightsquigarrow \lambda \Psi\ [a],\ \forall a',\ \Psi\ [(a, a'), a']$$

C-REBORROW-MUT
$$\frac{\mathbf{E}; \mathbf{L} \vdash \alpha' \sqsubseteq \alpha}{\mathbf{E}; \mathbf{L} \vdash p \lhd \&^{\alpha}_{\mathbf{mut}}\, \mathtt{T} \overset{\text{ctx}}{\Rrightarrow} p \lhd \&^{\alpha'}_{\mathbf{mut}}\, \mathtt{T}, p \lhd^{\dagger \alpha'} \&^{\alpha}_{\mathbf{mut}}\, \mathtt{T} \rightsquigarrow \lambda \Psi\ [(a, a')],\ \forall a'',\ \Psi\ [(a, a''), (a'', a')]}$$

Figure 5.2: Type-spec rules for type context inclusion

Well-typed function bodies $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F \rightsquigarrow \Phi}$

F-CONSEQUENCE
$$\frac{\mathbf{L} \Rightarrow \mathbf{L}' \qquad \mathbf{E}; \mathbf{L} \vdash \mathbf{T} \overset{\text{ctx}}{\Rightarrow} \mathbf{T}' \rightsquigarrow A \qquad \mathbf{E} \vdash \mathbf{K} \Rightarrow \mathbf{K}' \qquad \mathbf{E}; \mathbf{L}' \mid \mathbf{K}'; \mathbf{T}' \vdash F \rightsquigarrow \Phi \qquad \forall\, \Psi\, T, \Phi'\, \Psi\, T \to \Phi\, \Psi\, (A\, T)}{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F \rightsquigarrow \Phi'}$$

F-EQUALIZE
$$\frac{\mathbf{E}, \beta \sqsubseteq_{\mathrm{e}} \alpha, \alpha \sqsubseteq_{\mathrm{e}} \beta; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F \rightsquigarrow \Phi}{\mathbf{E}; \mathbf{L}, \beta \sqsubseteq_{\mathrm{l}} [\alpha] \mid \mathbf{K}; \mathbf{T} \vdash F \rightsquigarrow \Phi}$$

F-LET
$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{T}_1 \vdash I \dashv \mathtt{a}.\, \mathbf{T}_2 \rightsquigarrow \Phi_1 \qquad \Gamma, \mathtt{a} : \mathbf{val} \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}_2, \mathbf{T} \vdash F \rightsquigarrow \Phi_2}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}_1, \mathbf{T} \vdash \mathtt{let}\, \mathtt{a} = I\, \mathtt{in}\, F \rightsquigarrow \lambda \Psi\, (T_1 +\!\!+ T),\ \Phi_1\ (\lambda T_2,\ \Phi_2(T_2 +\!\!+ T))\ T_1}$$

F-LETCONT
$$\frac{\Gamma, k, \overline{\mathtt{a}} : \mathbf{val} \mid \mathbf{E}; \mathbf{L}_1 \mid \mathbf{K}, k \lhd \mathbf{cont}(\mathbf{L}_1; \overline{\mathtt{a}}.\, \mathbf{T}'; \Phi_1); \mathbf{T}' \vdash F_1 \rightsquigarrow \Phi_1 \qquad \Gamma, k : \mathbf{val} \mid \mathbf{E}; \mathbf{L}_2 \mid \mathbf{K}, k \lhd \mathbf{cont}(\mathbf{L}_1; \overline{\mathtt{a}}.\, \mathbf{T}'; \Phi_1); \mathbf{T} \vdash F_2 \rightsquigarrow \Phi_2}{\Gamma \mid \mathbf{E}; \mathbf{L}_2 \mid \mathbf{K}; \mathbf{T} \vdash \mathtt{letcont}\, k(\overline{\mathtt{a}}) := F_1\, \mathtt{in}\, F_2 \rightsquigarrow \Phi_2}$$

F-IF
$$\frac{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F_1 \rightsquigarrow \Phi_1 \qquad \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F_2 \rightsquigarrow \Phi_2}{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p \lhd \mathbf{bool} \vdash \mathtt{if}\, p\, \mathtt{then}\, F_1\, \mathtt{else}\, F_2 \rightsquigarrow \lambda \Psi\ (a :: T),\ (a \to \Phi_1\ \Psi\ T) \wedge (\neg a \to \Phi_2\ \Psi\ T)}$$

F-JUMP
$$\frac{\mathbf{E}; \mathbf{L} \vdash \mathbf{T} \overset{\text{ctx}}{\Rightarrow} \mathbf{T}'[\overline{\mathtt{b}}/\overline{\mathtt{a}}] \rightsquigarrow f \qquad k \lhd \mathbf{cont}(\mathbf{L}; \overline{\mathtt{a}}.\, \mathbf{T}', \Phi) \in \mathbf{K}}{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash \mathtt{jump}\, k(\overline{\mathtt{b}}) \rightsquigarrow \lambda \Psi\, \overline{b},\ \Phi\ \Psi\ (f\ \overline{b})}$$

F-CALL
$$\frac{\mathbf{E}; \mathbf{L} \vdash \overline{\alpha}\ \mathsf{alive} \qquad \Gamma, \mathtt{F} : \mathbf{lft} \mid \mathbf{E}, \mathtt{F} \sqsubseteq_{\mathrm{e}} \overline{\alpha}; \mathbf{L} \vdash \mathbf{E}' \qquad k \lhd \mathbf{cont}(\mathbf{L}; \mathtt{b}.\, \mathtt{b} \lhd \mathbf{own}\, \mathtt{T}, \mathbf{T}', \Phi) \in \mathbf{K} \qquad \Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T} \overset{\text{ctx}}{\Rightarrow} \overline{p \lhd \mathtt{T}}, \mathbf{T}' \rightsquigarrow \Phi_T}{\begin{array}{l}\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K} \\ ; f \lhd \mathbf{fn}(\mathtt{F} : \mathbf{E}'; \overline{\mathtt{T}}) \to \mathtt{T}, \mathbf{T} \vdash \mathtt{call}\, f(\overline{p})\, \mathtt{ret}\, k \rightsquigarrow \\ \lambda \Psi\, \Phi'\, T, \Phi_T\ (\lambda\, (\overline{p} +\!\!+ T'),\ \Phi\ (\lambda\, b, \Phi\ \Psi(b :: T'))\ \overline{p})\ T\end{array}}$$

F-NEWLFT
$$\frac{\Gamma, \alpha : \mathbf{lft} \mid \mathbf{E}; \mathbf{L}, \alpha \sqsubseteq_{\mathrm{l}} \overline{\alpha} \mid \mathbf{K}; \mathbf{T} \vdash F \rightsquigarrow \Phi}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash \mathtt{newlft}; F \rightsquigarrow \Phi}$$

F-ENDLFT
$$\frac{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}' \vdash F \rightsquigarrow \Phi \qquad \mathbf{T} \Rightarrow^{\dagger \alpha} \mathbf{T}'}{\mathbf{E}; \mathbf{L}, \alpha \sqsubseteq_{\mathrm{l}} \overline{\alpha} \mid \mathbf{K}; \mathbf{T} \vdash \mathtt{endlft}; F \rightsquigarrow \Phi}$$

F-ASSERT
$$\frac{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F \rightsquigarrow \Phi}{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; p \lhd \mathbf{bool}, \mathbf{T} \vdash \mathtt{assert}\, p; F \rightsquigarrow \lambda \Psi\ p :: T,\ p \wedge \Phi\ \Psi\ T}$$

F-RESOLVE
$$\frac{\mathbf{E}; \mathbf{L} \vdash \alpha\ \mathsf{alive} \qquad \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F \rightsquigarrow \Phi}{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; p \lhd \&^{\alpha}_{\mathbf{mut}}\, \mathtt{T}, \mathbf{T} \vdash \mathtt{resolve}\, p; F \rightsquigarrow \lambda \Psi\ (p, p') :: T,\ p = p' \to \Phi\ \Psi\ T}$$

Figure 5.3: Type-spec rules for well-typed functions

Type writing $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathtt{T}_1 \multimap^{\mathtt{T}} \mathtt{T}_2 \rightsquigarrow \nu, u}$

$$\textsc{Twrite-own}\ \frac{\mathsf{size}(\mathtt{T}) = \mathsf{size}(\mathtt{T}')}{\mathbf{E}; \mathbf{L} \vdash \mathbf{own}_n\, \mathtt{T}' \multimap^{\mathtt{T}} \mathbf{own}_n\, \mathtt{T} \rightsquigarrow id, (\lambda\, a\, b.\ b)}$$

$$\textsc{Twrite-bor}\ \frac{\mathbf{E}; \mathbf{L} \vdash \alpha\ \mathsf{alive}}{\mathbf{E}; \mathbf{L} \vdash \&^{\alpha}_{\mathbf{mut}}\, \mathtt{T} \multimap^{\mathtt{T}} \&^{\alpha}_{\mathbf{mut}}\, \mathtt{T} \rightsquigarrow \pi_1, (\lambda\, v\ w\ .\ (w, \pi_2\ v))}$$

Type reading $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathtt{T}_1 \circ\!\!-^{\mathtt{T}} \mathtt{T}_2 \rightsquigarrow \nu, u}$

$$\textsc{Tread-own-copy}\ \frac{\mathtt{T}\ \mathsf{copy}}{\mathbf{E}; \mathbf{L} \vdash \mathbf{own}_n\, \mathtt{T} \circ\!\!-^{\mathtt{T}} \mathbf{own}_n\, \mathtt{T} \rightsquigarrow id, id}$$

$$\textsc{Tread-own-move}\ \frac{n = \mathsf{size}(\mathtt{T})}{\mathbf{E}; \mathbf{L} \vdash \mathbf{own}_m\, \mathtt{T} \circ\!\!-^{\mathtt{T}} \mathbf{own}_m\, \lightning_n \rightsquigarrow id, (\lambda\, a.\ ())}$$

$$\textsc{Tread-bor}\ \frac{\mathtt{T}\ \mathsf{copy} \qquad \mathbf{E}; \mathbf{L} \vdash \alpha\ \mathsf{alive}}{\mathbf{E}; \mathbf{L} \vdash \&^{\alpha}_{\mu}\, \mathtt{T} \circ\!\!-^{\mathtt{T}} \&^{\alpha}_{\mu}\, \mathtt{T} \rightsquigarrow \pi_1, id}$$

Figure 5.4: Type-spec rules for type writing and reading

We begin by applying S-FN, which opens the function body and introduces our initial typing contexts:

$$\mathbf{E} = \{\mathtt{f} \sqsubseteq_{\mathrm{e}} \alpha\}$$
$$\mathbf{L} = \{\mathtt{f} \sqsubseteq []\}$$
$$\mathbf{K} = \{k \lhd \mathbf{cont}(\mathtt{f} \sqsubseteq []; \mathtt{a.a} \lhd \mathbf{own}_{()}\, ; \lambda\, \Psi\ (),\ \Psi\ ())\}$$
$$\mathbf{T} = \{x \lhd \mathbf{own}\ \&^{\alpha}_{\mathbf{mut}}\ \mathbf{int}\}$$

The external lifetime context $\mathbf{E}$ expresses that the borrow's lifetime α is longer than the function call's lifetime $\mathtt{f}$. The (internal) lifetime context $\mathbf{L}$ is used to track lifetimes that occur within the function. The continuation context $\mathbf{K}$ only contains the return continuation we use to exit the function. Finally, our type context $\mathbf{T}$ contains a binding for x.

In Figure 5.6, we provide a step-by-step typing for `decr_positive`. Because λ_{Rust} is much simpler than Rust, many seemingly atomic steps in the source Rust have been broken down, particularly the dereferencing of the mutable borrow. Next to each line of code is the set of typing rules applied to type that instruction, apart from structural rules like weakening. Below, we show the state of any contexts in which this rule has changed. We also show the predicate transformer for the rest of the program; these can be read like a Hoare weakest-precondition calculus.

Because all local variables are boxed in λ_{Rust}, we must first unbox x, which we do by performing a S-DEREF and moving the borrow out of our box with TREAD-OWN-MOVE. Doing this changes the type of x, making the box's contents uninitialized ($\lightning$). We must update the transformer each time a rule is used, like Dijkstra's predicate-transformer semantics [28].

By applying each set of rules, we can straightforwardly type the program until we get to the more unique **`resolve`** instruction of λ_{Rust}. This ghost instruction is used to syntactically mark the resolution of mutable borrows, at which point the prophecy becomes fixed. The point of resolution is *not* the same as the lifetime's end: the lifetime α of the borrow is longer than the function call.

Well-typed instructions $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{T}_1 \vdash I \dashv \mathtt{a}.\, \mathbf{T}_2 \rightsquigarrow \Phi}$

S-TRUE
$$\mathbf{E}; \mathbf{L} \mid \emptyset \vdash \mathtt{true} \dashv \mathtt{a}.\, \mathtt{a} \lhd \mathbf{bool} \rightsquigarrow \lambda\, \Psi\ [],\ \Psi\ [\mathtt{true}]$$

S-FALSE
$$\mathbf{E}; \mathbf{L} \mid \emptyset \vdash \mathtt{false} \dashv \mathtt{a}.\, \mathtt{a} \lhd \mathbf{bool} \rightsquigarrow \lambda\, \Psi\ [],\ \Psi\ [\mathtt{false}]$$

S-NUM
$$\mathbf{E}; \mathbf{L} \mid \emptyset \vdash z \dashv \mathtt{a}.\, \mathtt{a} \lhd \mathbf{int} \rightsquigarrow \lambda\, \Psi\ [],\ \Psi\ [z]$$

S-FN
$$\frac{\overline{\mathtt{T}}'\ \text{copy} \qquad \overline{\mathtt{T}}'\ \text{send} \qquad \Gamma, \overline{\alpha}, \mathtt{F} : \mathbf{lft}, f, \overline{\mathtt{a}}, k : \mathbf{val} \mid \mathbf{E}, \mathbf{E}'; \mathtt{F} \sqsubseteq_1 [] \mid k \lhd \mathbf{cont}(\mathtt{F} \sqsubseteq_1 []; \mathtt{b}.\, \mathtt{b} \lhd \mathbf{own}\, \mathtt{T}; \lambda\, \Psi\ [a],\ \Psi\ [a]);\ \overline{p} \lhd \overline{\mathtt{T}}', \overline{\mathtt{a}} \lhd \mathbf{own}\, \overline{\mathtt{T}}, f \lhd \forall \overline{\alpha}.\, \mathbf{fn}(\mathtt{F} : \mathbf{E}; \overline{\mathtt{T}}) \to \mathtt{T} \vdash F \rightsquigarrow \Phi}{\Gamma \mid \mathbf{E}'; \mathbf{L}' \mid \overline{p} \lhd \overline{\mathtt{T}}' \vdash \mathtt{funrec}\, f(\overline{\mathtt{a}})\, \mathtt{ret}\, k := F \dashv f.\, f \lhd \forall \overline{\alpha}.\, \mathbf{fn}(\mathtt{F} : \mathbf{E}; \overline{\mathtt{T}}) \to \mathtt{T} \rightsquigarrow \Phi}$$

S-PATH
$$\mathbf{E}; \mathbf{L} \mid p \lhd \mathtt{T} \vdash p \dashv \mathtt{a}.\, \mathtt{a} \lhd \mathtt{T} \rightsquigarrow \lambda\, \Psi\ p,\ \Psi\ p$$

S-NAT-OP
$$\mathbf{E}; \mathbf{L} \mid p_1 \lhd \mathbf{int}, p_2 \lhd \mathbf{int} \vdash p_1\, \{+, -\}\, p_2 \dashv \mathtt{a}.\, \mathtt{a} \lhd \mathbf{int} \rightsquigarrow \lambda\, \Psi\ [n, m],\ \Psi\ [n\, \{+, -\}\, m]$$

S-NAT-LEQ
$$\mathbf{E}; \mathbf{L} \mid p_1 \lhd \mathbf{int}, p_2 \lhd \mathbf{int} \vdash p_1 \leq p_2 \dashv \mathtt{a}.\, \mathtt{a} \lhd \mathbf{bool} z \rightsquigarrow \lambda\, \Psi\ [n, m],\ \Psi\ [n \leq m]$$

S-NEW
$$\mathbf{E}; \mathbf{L} \mid \emptyset \vdash \mathtt{new}(n) \dashv \mathtt{a}.\, \mathtt{a} \lhd \mathbf{own}_n \lightning_n \rightsquigarrow \lambda\, \Psi\ [],\ \Psi\ [()]$$

S-DELETE
$$\frac{n = \mathsf{size}(\mathtt{T})}{\mathbf{E}; \mathbf{L} \mid p \lhd \mathbf{own}_n\, \mathtt{T} \vdash \mathtt{delete}(n, p) \dashv \emptyset \rightsquigarrow \lambda\, \Psi\ [a],\ \Psi\ []}$$

S-DEREF
$$\frac{\mathbf{E}; \mathbf{L} \vdash \mathtt{T}_1 \multimap^{\mathtt{T}} \mathtt{T}_1' \rightsquigarrow \nu, u \qquad \mathsf{size}(\mathtt{T}) = 1}{\mathbf{E}; \mathbf{L} \mid p \lhd \mathtt{T}_1 \vdash {}^*p \dashv \mathtt{a}.\, p \lhd \mathtt{T}_1', \mathtt{a} \lhd \mathtt{T} \rightsquigarrow \lambda\, \Psi\ [a],\ \Psi\ [\nu\ a, u\ a]}$$

S-DEREF-SHR-BOR-OWN
$$\frac{\mathbf{E}; \mathbf{L} \vdash \alpha\ \mathsf{alive}}{\mathbf{E}; \mathbf{L} \mid p \lhd \&^{\alpha}_{\mathbf{shr}}\, \mathbf{own}_n\, \mathtt{T} \vdash {}^*p \dashv \mathtt{a}.\, \mathtt{a} \lhd \&^{\alpha}_{\mathbf{shr}}\, \mathtt{T} \rightsquigarrow \lambda\, \Psi\ [a],\ \Psi\ [a]}$$

S-DEREF-SHR-BOR-BOR
$$\frac{\mathbf{E}; \mathbf{L} \vdash \alpha\ \mathsf{alive} \qquad \mathbf{E}; \mathbf{L} \vdash \alpha \sqsubseteq \alpha'}{\mathbf{E}; \mathbf{L} \mid p \lhd \&^{\alpha}_{\mathbf{shr}}\, \&^{\alpha'}_{\mathbf{shr}}\, \mathtt{T} \vdash {}^*p \dashv \mathtt{a}.\, \mathtt{a} \lhd \&^{\alpha}_{\mathbf{shr}}\, \mathtt{T} \rightsquigarrow \Psi\ [a],\ \Psi\ [a]}$$

S-DEREF-MUT-BOR-OWN
$$\frac{\mathbf{E}; \mathbf{L} \vdash \alpha\ \mathsf{alive}}{\mathbf{E}; \mathbf{L} \mid p \lhd \&^{\alpha}_{\mathbf{mut}}\, \mathbf{own}_n\, \mathtt{T} \vdash {}^*p \dashv \mathtt{a}.\, \mathtt{a} \lhd \&^{\alpha}_{\mathbf{mut}}\, \mathtt{T} \rightsquigarrow \lambda\, \Psi\ [(a, a')],\ a = a' \to \Psi\ [a]}$$

S-DEREF-MUT-BOR-BOR
$$\frac{\mathbf{E}; \mathbf{L} \vdash \alpha\ \mathsf{alive} \qquad \mathbf{E}; \mathbf{L} \vdash \alpha \sqsubseteq \alpha'}{\mathbf{E}; \mathbf{L} \mid p \lhd \&^{\alpha}_{\mathbf{mut}}\, \&^{\alpha'}_{\mathbf{mut}}\, \mathtt{T} \vdash {}^*p \dashv \mathtt{a}.\, \mathtt{a} \lhd \&^{\alpha}_{\mathbf{mut}}\, \mathtt{T} \rightsquigarrow \lambda\, \Psi\ [((v, w), (v', w'))],\ w' = w \to \Psi\ [(v, v')]}$$

Figure 5.5: Type-spec rules for well-typed instructions

$$\text{S-ASSGN}\quad \frac{\mathbf{E};\mathbf{L} \vdash \mathtt{T}_1 \multimap^{\mathtt{T}} \mathtt{T}_1' \rightsquigarrow \nu, u}{\mathbf{E};\mathbf{L} \mid p_1 \lhd \mathtt{T}_1, p_2 \lhd \mathtt{T} \vdash p_1 := p_2 \dashv p_1 \lhd \mathtt{T}_1' \rightsquigarrow \lambda\Psi\ [a,b].\ u\ a\ b}$$

$$\text{S-MEMCPY}\quad \frac{\mathsf{size}(\mathtt{T}) = n \qquad \mathbf{E};\mathbf{L} \vdash \mathtt{T}_1 \multimap^{\mathtt{T}} \mathtt{T}_1' \rightsquigarrow\ \nu_a, u_a \qquad \mathbf{E};\mathbf{L} \vdash \mathtt{T}_2 \multimap^{\mathtt{T}} \mathtt{T}_2' \rightsquigarrow\ \nu_b, u_b}{\mathbf{E};\mathbf{L} \mid p_1 \lhd \mathtt{T}_1, p_2 \lhd \mathtt{T}_2 \vdash p_1 :=_n {}^*p_2 \dashv p_1 \lhd \mathtt{T}_1', p_2 \lhd \mathtt{T}_2' \rightsquigarrow\ \lambda\Psi\, a\, b,\ \Psi(u_a\ a\ (\nu_b\ b))\ (u_b\ b)}$$

Figure 5.5: Type-spec rules for well-typed instructions (cont)

Instead, this resolution corresponds to where x' falls out of scope. In surface Rust this could also be called the 'precise drop' of the borrow.

Because resolve consumes the borrow being resolved, it cannot be placed too early: if the borrow was needed later in the program this would lead to a typing error. However, if a resolve is forgotten the program will still type, however certain specifications may no longer be provable.

Finally, the atypical `jump` instruction calls a continuation, the return continuation, and exits the function.

The `main` function As we will see in Theorem 5.2.2, RustHornBelt doesn't provide a proof of soundness for standalone functions like `decr_positive`, instead the key theorem only applies to closed, well-typed programs. For this reason, we also provide the typing of a `main` function, which calls `decr_positive`. The λ_{Rust} code is provided in Figure 5.7, and the judgment we wish to establish for this code is:

$$\emptyset;\emptyset \mid \emptyset;\emptyset \vdash \texttt{main} \dashv f.\ f \lhd \forall\alpha.\ \mathbf{fn}(\mathcal{F} \sqsubseteq [];\mathbf{own}\ ()) \to () \rightsquigarrow \lambda\Psi\,(),\ \Psi\ ()$$

Like `decr_positive`, this translation involves a certain amount of ceremony and boilerplate, we are obligated to box variables before borrowing or writing to them. Similarly, to call a function we must provide a return continuation, which requires us to allocate a continuation inside of `main`. We focus on a few salient details of this program.

On line A, we use C-BORROW to create a borrow from b. Unlike Rust, λ_{Rust} performs borrowing as part of type-context inclusion. This operation introduces a new prophecy for the mutable borrow. Note that the code also is free of any `resolve` statements; the unique borrow will be consumed by `decr_positive`, and thus cannot be resolved within the scope of `main`. On the other hand, the associated lifetime α will be resolved here. On line 9, we create the return continuation for the call; this continuation requires the value of x to be 42 as a precondition to running. We perform the actual call on line 16. At this point, we can instantiate the postcondition in the transformer of `decr_positive` with the one provided to r, allowing us to establish the postcondition we sought.

Combined with Theorem 5.2.2, the judgment for `main` tells us the program will not get stuck; in particular, the assert statement must succeed, finally establishing the safety property we sought.

5.2 Soundness of Type-Specs

Traditionally, the purpose of a type-system is to avoid 'bad' behaviors; it should have a concomitant notion of safety. At a base level, we should ensure that well-typed programs do not exhibit *undefined behavior*, a property that was ensured by the prior work RustBelt. In RustHornBelt, we extended λ_{Rust} with assertions, allowing us to extend our notion of safety to ensure that programs do not fail their assertions. The traditional approach to proving the safety of a type-system uses *syntactic*

$\texttt{funrec}\,\texttt{decr\_positive}(x)\,\texttt{ret}\,k :=$

$\{\mathbf{E} : \mathcal{F} \sqsubseteq_e \alpha; \mathbf{L} : \mathcal{F} \sqsubseteq [] \mid \mathbf{K} : k \lhd \mathbf{cont}(\mathcal{F} \sqsubseteq []; \mathsf{a.a} \lhd \mathbf{own}\ (); \Psi); \mathbf{T} : x \lhd \mathbf{own}\ \&^{\alpha}_{\mathbf{mut}}\ \mathbf{int}\} \rightsquigarrow$
$\{x_{.0} > 0 \land (x_{.1} = x_{.0} - 1 \to \Psi\ ())\}$

$\texttt{let}\,x' = {}^*x\,\texttt{in}$ $\{\text{F-LET}, \text{S-DEREF}, \text{TREAD-OWN-MOVE}\}$

$\{\mathbf{T} : x' \lhd \&^{\alpha}_{\mathbf{mut}}\ \mathbf{int}, x \lhd \mathbf{own}\ \lightning\} \rightsquigarrow \{x'_{.0} > 0 \land (x'_{.1} = x'_{.0} - 1 \to \Psi\ ())\}$

$\texttt{let}\,x'' = {}^*x'\,\texttt{in}$ $\{\text{F-LET}, \text{S-DEREF}, \text{TREAD-OWN-COPY}\}$

$\{\mathbf{T} : x'' \lhd \mathbf{int}, x' \lhd \&^{\alpha}_{\mathbf{mut}}\ \mathbf{int}\} \rightsquigarrow \{x'' > 0 \land (x'_{.1} = x'' - 1 \to \Psi\ ())\}$

$\texttt{let}\,c = x'' > 0\,\texttt{in}$ $\{\text{F-LET}, \text{S-NAT-LEQ}\}$

$\{\mathbf{T} : c \lhd \mathbf{bool}, x'' \lhd \mathbf{int}, x' \lhd \&^{\alpha}_{\mathbf{mut}}\ \mathbf{int}\} \rightsquigarrow \{c \land (x'_{.1} = x'' - 1 \to \Psi\ ())\}$

$\texttt{assert}\,c;$ $\{\text{F-ASSERT}\}$

$\{\mathbf{T} : x'' \lhd \mathbf{int}, x' \lhd \&^{\alpha}_{\mathbf{mut}}\ \mathbf{int}\} \rightsquigarrow \{x'_{.1} = x'' - 1 \to \Psi\ ()\}$

$\texttt{let}\,v = x'' - 1\,\texttt{in}$ $\{\text{F-LET}, \text{S-NAT-OP}\}$

$\{\mathbf{T} : v \lhd \mathbf{int}, x' \lhd \&^{\alpha}_{\mathbf{mut}}\ \mathbf{int}\} \rightsquigarrow \{x'_{.1} = v \to \Psi\ ()\}$

$x' := v;$ $\{\text{F-LET}, \text{S-ASSGN}, \text{TWRITE-BOR}\}$

$\{\mathbf{T} : x' \lhd \&^{\alpha}_{\mathbf{mut}}\ \mathbf{int}\} \rightsquigarrow \{x'_{.1} = x'_{.0} \to \Psi\ ()\}$

$\texttt{resolve}\,x';$ $\{\text{F-RESOLVE}\}$

$\{\mathbf{T} : \varnothing\} \rightsquigarrow \{\Psi\ ()\}$

$\texttt{jump}\,k()$ $\{\text{F-JUMP}\}$

Figure 5.6: Annotated body of `decr_positive`

$\texttt{funrec}\,\texttt{main}()\,\texttt{ret}\,k :=$

$\{\mathbf{E} = \varnothing; \mathbf{L} = \mathrm{F} \sqsubseteq [] \mid \mathbf{K} : k \lhd \mathbf{cont}(\mathrm{F} \sqsubseteq []; \mathsf{a}.\mathsf{a} \lhd \mathbf{own}\ (); \lambda\Psi\ (),\ \Psi\ ()); \mathbf{T} = \varnothing\} \rightsquigarrow \{\Psi\ ()\}$

$\texttt{let}\,\texttt{decr\_positive} = \texttt{funrec}\,\texttt{decr\_positive}(..)\,\texttt{ret}\,k := \ldots\,\texttt{in}$ $\{\ldots\}$

$- \rightsquigarrow \{\Psi\ ()\}$

$\{\text{F-CONSEQUENCE}\}$

$- \rightsquigarrow \{\forall y',\ 43 > 0 \land (y' = 43 - 1 \to y' = 42 \land \Psi\ ())\}$

$\texttt{let}\,x = 43\,\texttt{in}$ $\{\text{F-LET}, \text{S-NUM}\}$

$\{\mathbf{T} : x \lhd \mathbf{int}\} \rightsquigarrow \{\forall y',\ x > 0 \land (y' = x - 1 \to y' = 42 \land \Psi\ ())\}$

$\texttt{let}\,y = \texttt{new}(1)\,\texttt{in}$ $\{\text{F-LET}, \text{S-NEW}\}$

$\{\mathbf{T} : y \lhd \mathbf{own}\ \lightning, x :\lhd \mathbf{int}\} \rightsquigarrow \{\forall y',\ x > 0 \land (y' = x - 1 \to y' = 42 \land \Psi\ ())\}$

$y := x$ $\{\text{F-LET}, \text{S-ASSGN}, \text{TWRITE-OWN}\}$

$\{\mathbf{T} : y \lhd \mathbf{own}\ \mathbf{int}\} \rightsquigarrow \{\forall y',\ y > 0 \land (y' = y - 1 \to y' = 42 \land \Psi\ ())\}$

$\texttt{let}\,b = \texttt{new}(1)\,\texttt{in}$ $\{\text{F-LET}, \text{S-NEW}\}$

$\{\mathbf{T} : b \lhd \mathbf{own}\ \lightning, y \lhd \mathbf{own}\ \mathbf{int}\} \rightsquigarrow \{\forall y',\ y > 0 \land (y' = y - 1 \to y' = 42 \land \Psi\ ())\}$

(A) $\texttt{newlft}\ \alpha;$ $\{\text{F-NEWLFT}, \text{C-BORROW}\}$

$\{\mathbf{L} = \alpha \sqsubseteq [\mathrm{F}], \mathrm{F} \sqsubseteq []\mathbf{T} : b \lhd \mathbf{own}\ \lightning, y \lhd \&^{\alpha}_{\mathbf{mut}}\,\mathbf{int}, y \lhd^{\dagger\alpha}\,\mathbf{own}\ \mathbf{int}\} \rightsquigarrow$

$\{y_0 > 0 \land (y_1 = y_0 - 1 \to y_1 = 42 \land \Psi\ ())\}$

$b := y;$ $\{\text{F-LET}, \text{S-ASSGN}, \text{TWRITE-OWN}\}$

$\{\mathbf{T} : b \lhd \mathbf{own}\ \&^{\alpha}_{\mathbf{mut}}\,\mathbf{int}, y \lhd^{\dagger\alpha}\,\mathbf{own}\ \mathbf{int}\} \rightsquigarrow \{b_0 > 0 \land (b_1 = b_0 - 1 \to b_1 = 42 \land \Psi\ ())\}$

$\texttt{letcont}\,r(()) =$

$\quad\texttt{endlft}\ \alpha;$ $\{\text{F-ENDLFT}\}$

$\quad\{y \lhd \mathbf{own}\ \mathbf{int}\} \rightsquigarrow \{y = 42 \land \Psi\ ()\}$

$\quad\texttt{let}\,x = {}^{*}y\,\texttt{in}$ $\{\text{F-LET}, \text{S-DEREF}, \text{TREAD-OWN-MOVE}\}$

$\quad\{x \lhd \mathbf{int}\} \rightsquigarrow \{x = 42 \land \Psi\ ()\}$

$\quad\texttt{let}\,c = x = 42\,\texttt{in}$ $\{\text{F-LET}, \text{S-NAT-LEQ}\}$

$\quad\{\mathbf{T} : c \lhd \mathbf{bool}\} \rightsquigarrow \{c \land \Psi\ ()\}$

$\quad\texttt{assert}\,c;$ $\{\text{F-ASSERT}\}$

$\quad\{\mathbf{T} = \varnothing\} \rightsquigarrow \{\Psi\ ()\}$

$\quad\texttt{jump}\,k()$ $\{\text{F-JUMP}\}$

$\texttt{in}$ $\{\text{F-LETCONT}\}$

$$\left\{\begin{array}{l}\mathbf{K} : r \lhd \mathbf{cont}(\alpha \sqsubseteq [\mathrm{F}], \mathrm{F} \sqsubseteq []; \mathsf{a}.\mathsf{a} \lhd \mathbf{own}\ (), y \lhd^{\dagger\alpha}\,\mathbf{own}\ \mathbf{int}; \lambda\Psi\ [(), y],\ y = 42 \land \Psi\ ()),\\ \quad k \lhd \mathbf{cont}(\mathrm{F} \sqsubseteq []; \mathsf{a}.\mathsf{a} \lhd \mathbf{own}\ (); \lambda\Psi\ (),\ \Psi\ ())\end{array}\right\}$$

$\texttt{call}\,\texttt{decr\_positive}(b)\,\texttt{ret}\,r$ $\{\text{F-CALL}\}$

Figure 5.7: Annotated body for `main`

approach. This technique has enjoyed tremendous popularity due to the mechanistic nature of the proofs involved, which are often not much more than 'symbol pushing.' However, syntactic proofs are fundamentally non-modular: each new construct added to a language requires redoing all the proofs and may introduce complex interactions with other proof cases.

This non-modularity is a problem when working with a language like Rust, which features a notion of *unsafe* code, whose safety cannot be determined solely from the syntax of the code. Consider the case of `Vec`, which can be represented as a triplet (len, cap, ptr), where ptr is a raw machine pointer to a memory allocation. Indexing our vector requires taking an offset from ptr, which is an *unsafe* operation; if the offset we asked for is greater than len we would potentially be reading uninitialized memory! Because we cannot write judgments that would suffice to ensure the safety of an implementation of `Vec`, to add it to our language, we would have to axiomatize it, extending our type with new hardcoded judgments for various vector operations. This approach is unsatisfying, especially because `Vec` *behaves* safely: using its public (safe) APIs, we cannot cause undefined behavior.

Accounting for and reasoning about unsafe code was one of the primary motivations of RustBelt and the *semantic typing* approach taken to prove soundness. Rather than proving the soundness of Definition 5 directly, we will define a new *semantic judgement*:

$$\mathbf{E}; \mathbf{L} \mid \mathbf{T} \vDash I \dashv \texttt{r}.\ \mathbf{T}' \rightsquigarrow \Phi$$

as a logical relation in the separation logic Iris. We then show two properties: first, this relation subsumes the syntactic one previously used, and second, that this relation is adequate (*i.e.,* that well-typed functions do not get stuck). In this section, we will start by introducing a new model of prophecies in Iris (§ 5.2.1), then we will define our new relation (§ 5.2) and show how we can prove the existence of core rules (§ 5.2.4). Finally, in § 5.3, we will show how semantic typing gives us greater flexibility when dealing with unsafe code.

5.2.1 Parametric Prophecies

The support for mutable borrows is the critical challenge of RustHornBelt, particularly in providing a semantic accounting of the RustHorn-style prophecies that appear in our specifications. We developed a novel prophecy framework in Iris called *parametric prophecies* to solve this. Its key idea is to *consider all possible futures simultaneously.* This is achieved through the *clairvoyant monad* $Clair\ A \triangleq ProphAsn \to A$, a reader monad over a *prophecy assignment* $\pi \in ProphAsn$ modeling one possible future (*i.e.,* mapping of prophecy variables to values). By embedding our reasoning about prophecies (especially the spec Φ) within this monad—*i.e.,* parameterizing over *every future* π—we can *refer to* prophesied values while staying parametric w.r.t. what they are until we are ready to resolve them. In particular, when proving C-BORROW, parametric prophecies will enable us to instantiate a' with a freshly chosen prophecy variable in the domain of π, without having to commit to how it is resolved until the borrow is dropped.

Basics Formally, let a *prophecy (variable)* $x \in ProphVar\ A$ be simply a wrapper around a natural number $n \in \mathbb{N}$. As $ProphVar\ A$ is infinite, we can at any point create a *prophecy token* $[x]_1$ for a *fresh* prophecy x. This token signifies that x has not yet been resolved. Ownership of prophecy tokens can be fractionally split and merged:

PROPH-INTRO
$$\mathsf{True} \Rrightarrow \exists x.\, [x]_1$$

PROPH-FRAC
$$[x]_{q+q'} \dashv\vdash [x]_q * [x]_{q'}$$

A *prophecy assignment* $\pi \in ProphAsn$, modeling one possible future, is a dependent map that assigns a value $\pi\, x \in A$ to every prophecy $x \in ProphVar\ A$ for any type A. Now we have the

clairvoyant monad $\mathit{Clair}\ A \triangleq \mathit{ProphAsn} \to A$, parameterized over *every future* π.

Syntactic Conventions We use the syntax $\uparrow x \triangleq \lambda\pi.\,\pi\,x$ to lift a prophecy $x \in \mathit{ProphVar}\ A$ into a clairvoyant value (*i.e.,* values of sort $\mathit{Clair}\ A$). We mark clairvoyant values with a hat $\hat{}$ (*e.g.,* $\hat{a}$). Also, we use the following *functorial* notations with a star *: $\hat{\phi}\ {}^\star\!\wedge\ \hat{\psi} \triangleq \lambda\pi.\hat{\phi}\,\pi \wedge \hat{\psi}\,\pi$, $\hat{a}\ {}^\star\!= \hat{b} \triangleq \lambda\pi.\hat{a}\,\pi = \hat{b}\,\pi$, $\hat{p}{}^\star\!1 \triangleq \lambda\pi.(\hat{p}\,\pi).1$ (similarly for $^\star\!2$), ${}^\star(\hat{a}, \hat{b}) \triangleq \lambda\pi.(\hat{a}\,\pi, \hat{b}\,\pi)$, and ${}^\star[\hat{a}_1, \ldots, \hat{a}_n] \triangleq \lambda\pi.[\hat{a}_1\,\pi, \ldots, \hat{a}_n\,\pi]$.

A *prophecy observation* $\langle\hat{\phi}\rangle$ (where $\hat{\phi} \in \mathit{Clair}\ \mathsf{Prop}$), asserts that a pure proposition $\hat{\phi}\,\pi$ holds for *every valid future* π (*i.e.,* for every π that respects the prophecy resolutions that have occurred so far). The rules for reasoning about observations are fairly straightforward:

$$\textsc{Proph-Impl}\ \frac{\forall\pi.\ \hat{\phi}\,\pi \to \hat{\psi}\,\pi}{\langle\hat{\phi}\rangle \vdash \langle\hat{\psi}\rangle} \qquad \textsc{Proph-Merge}\ \langle\hat{\phi}\rangle * \langle\hat{\psi}\rangle \vdash \langle\hat{\phi}\ {}^\star\!\wedge\ \hat{\psi}\rangle \qquad \textsc{Proph-True}\ \frac{\forall\pi.\ \hat{\phi}\,\pi}{\langle\hat{\phi}\rangle}$$

Definition 6 (Dependencies)**.** *The dependencies* Y *of a clairvoyant value* $\hat{a}$ *are defined as the set of prophecy variables that occur in* $\hat{a}$*. Equivalently,*

$$\mathit{dep}(\hat{a}, Y) \triangleq \forall\pi, \pi'.\,(\forall z \in Y.\,\pi\,z = \pi'\,z) \to \hat{a}\,\pi = \hat{a}\,\pi'.$$

We can define the key *resolution* rule of prophecy observations, allowing us to resolve each x exactly once.

$$\textsc{Proph-Resolve}\ \frac{\mathsf{dep}(\hat{a}, Y)}{[x]_1 * [Y]_q \ \Rrightarrow\ \langle\uparrow x\ {}^\star\!= \hat{a}\rangle * [Y]_q}$$

Consuming the full token $[x]_1$, we can finally fix the value of the prophecy x to an arbitrary clairvoyant value $\hat{a}$, getting an *observational* equality: $\langle\uparrow x\ {}^\star\!= \hat{a}\rangle$. Internally, we *prune away* all the futures in which x is not equal to $\hat{a}$.

Notably, the rule Proph-Resolve allows the clairvoyant value $\hat{a}$ to depend on *other* prophecies (the ones in the set Y). This is essential in RustHornBelt to model *borrow splitting*. For example, in `Vec`'s `index_mut` (§5.3), the input `v`: `&`α `mut Vec<T>`'s prophecy x should be *partially resolved* to a value depending on the output `&mut T`'s prophecy y, observing $\langle\uparrow x\ {}^\star\!= {}^\star[\ldots, \hat{a}_{i-1}, \uparrow y, \hat{a}_{i+1}, \ldots]\rangle$.

Crucially, however, Proph-Resolve also imposes the condition that the prophecies in Y (*i.e.,* the ones $\hat{a}$ depends on) must all be *unresolved*. This is ensured by consuming (and then immediately returning) fractional tokens for the prophecies in Y—*i.e.,* $[Y]_q \triangleq \mathop{\ast}_{y \in Y} [y]_q$. We need this condition to prevent prophecy resolution from causing a paradox where there are no valid futures. To see how this might happen, suppose we have $[x]_1$ and $[y]_1$; if Proph-Resolve did not impose the "$[Y]_q$" condition, we could use it to first resolve x to $\uparrow y$, and then resolve y to $\lambda\pi.\uparrow x\,\pi + 1$, which put together would yield the impossible observation $\langle\uparrow x\ {}^\star\!= \lambda\pi.\uparrow x\,\pi + 1\rangle$. Thanks to the "$[Y]_q$" condition, however, such a paradox is ruled out. As a result, we can additionally prove the following rule, which establishes that reasoning within the clairvoyant monad remains consistent (*i.e.,* there always exists *some* valid π under which our observations hold):

$$\textsc{Proph-Sat}\ \langle\hat{\phi}\rangle \ \Rrightarrow\ \exists\pi.\,\hat{\phi}\,\pi$$

When dealing with prophecies, we may need to resolve a prophecy where we cannot provide all the dependency tokens at the moment of resolution. To work around this, we can use a *prophecy equalizer* to delay the full resolution of a prophecy.

Definition 7 (Prophecy Equalizer)**.** *For $\hat{a}, \hat{b} \in$ Clair A, the prophecy equalizer $\hat{a} :\approx \hat{b}$ is defined as the proposition* $\mathsf{dep}(\hat{a}, Y) \to [Y]_q \Rrightarrow \langle \lambda\pi.\, \hat{a}\ \pi = \hat{b}\ \pi \rangle$

The rules for prophecy equalizers slightly generalize those for observations:

PROPH-EQZ-TOK
$[x]_1 \vdash \uparrow x :\approx \hat{a}$

PROPH-EQZ-MOD
$\langle \hat{a} \stackrel{\star}{=} \hat{b} \rangle * \hat{a} :\approx \hat{c} \vdash \hat{b} :\approx \hat{c}$

PROPH-EQZ-APP
$$\frac{f \text{ injective}}{\hat{a} :\approx \hat{b} \vdash f \circ \hat{a} :\approx f \circ \hat{b}}$$

5.2.2 Semantic interpretation of Rust types

Like the prior work RustBelt, we model Rust types using a pair of Iris predicates, one describing the *ownership* of that type and one which captures its behavior when *shared*.

$$\begin{aligned} [\![\mathtt{T}]\!].\mathrm{own} &: \mathit{Clair}\ \lfloor \mathtt{T} \rfloor \times \mathit{ThrId} \times \mathit{ListVal} \to \mathit{iProp} \\ [\![\mathtt{T}]\!].\mathrm{shr} &: \mathit{Clair}\ \lfloor \mathtt{T} \rfloor \times \mathit{Lft} \times \mathit{ThrId} \times \mathit{Loc} \to \mathit{iProp} \end{aligned}$$

The *ownership predicate* $[\![\mathtt{T}]\!].\mathrm{own}$ models what it means to *own* an object of type `T`, including the ownership of *resources* (*e.g.,* memory) granted to the object. Like RustBelt the predicate is parametrized over the list of *low-level values* $\overline{v} \in \mathit{ListVal}$, describing the sequence of values stored in the memory for the object, and the thread identifier $t \in \mathit{ThrId}$, used when modeling concurrency. We extend it with a clairvoyant representation value $\hat{a} \in \mathit{Clair}\ \lfloor \mathtt{T} \rfloor$, relating the logical values of the type to their memory representations.

The second *sharing predicate* allows Rust types to behave differently when shared. For example, the cell type `Cell<T>` is equivalent to `T` when fully owned, but it becomes a *shared mutable state* when put under a shared reference. To model such behaviors, RustBelt associates with each type a *sharing predicate* $[\![\mathtt{T}]\!].\mathrm{shr}$. The predicate $[\![\mathtt{T}]\!].\mathrm{shr}(\alpha, t, \ell)$ is meant to model $[\![\&^{\alpha}_{\mathsf{shr}}\, \mathtt{T}]\!].\mathrm{own}(t, [\ell])$, the ownership predicate of a shared reference to `T`. Thus, the predicate also has a *location* (aka address) $\ell \in \mathit{Loc}$ to represent where the object is stored.

Like RustBelt, each type must also ensure these predicates satisfy several properties, ensuring they are well-behaved, like the ability to convert an owning predicate into a sharing one:

TY-SHARE
$$\&^{\alpha}\big(\exists \overline{v}.\ \ell \mapsto \overline{v} * [\![\mathtt{T}]\!].\mathrm{own}(\hat{a}, t, \overline{v})\big) * [\alpha]_q \quad \Rrightarrow \quad [\![\mathtt{T}]\!].\mathrm{shr}(\hat{a}, \alpha, t, \ell) * [\alpha]_q$$

RustHornBelt adds several additional properties that each type must satisfy, like the ability to retrieve the prophecies contained within a value of a given type:

TY-OWN-PROPH
$$\begin{aligned} &[\![\mathtt{T}]\!].\mathrm{own}(\hat{a}, t, \overline{v}) * [\mathtt{T}.\mathsf{lft}]_{q'} \Rrightarrow \\ &\quad \exists\, Y\ \overline{x}\ q,\ \mathsf{dep}(\hat{a}, Y) * [Y]_q * ([Y]_q \Rrightarrow [\![\mathtt{T}]\!].\mathrm{own}(\hat{a}, t, \overline{v}) * [\mathtt{T}.\mathsf{lft}]_{q'}) \end{aligned}$$

Given the ownership predicate for a type `T`, and a token witnessing that the lifetimes appearing in `T` are alive, we can temporarily extract fragments of the prophecies appearing in the value $\hat{a}$[3].

[3]The full version of this rule, present in the Coq development must also account for *step-indexing*, a complication required for soundness in Iris.

Booleans The Boolean type `bool`, written **bool** in λ_{Rust}, is one of the simplest types in Rust. Its ownership predicate is defined as follows:

$$[\![\textbf{bool}]\!].\text{own}(\hat{b}, \_, [v]) \triangleq (\lambda\, x.v) = \hat{b}$$

This is just the equality $v = b$, but we must also account for the prophecy assignment in $\hat{b}$. The sharing predicate $[\![\textbf{bool}]\!].\text{shr}$ is derived from $[\![\textbf{bool}]\!].\text{own}$, just as in RustBelt. The integer type **int** is also interpreted similarly.

Products In Rust, we can make new types out of existing types in many ways. A simple example of that is the product type $\tau_0 \times \tau_1$ (written `(T0, T1)` in Rust). The ownership predicate of the product type can be derived from τ_0 and τ_1's ownership predicates as follows:

$$[\![\tau_0 \times \tau_1]\!].\text{own}(\hat{p}, t, \overline{v}) \triangleq \exists\, \overline{v_0}, \overline{v_1} \text{ s.t. } \overline{v} = \overline{v_0} +\!\!+ \overline{v_1}.\ [\![\tau_0]\!].\text{own}(\hat{p}\text{:}\star 0, t, \overline{v_0}) * [\![\tau_1]\!].\text{own}(\hat{p}\text{:}\star 1, t, \overline{v_1})$$

The representation value is set to the pair of those of the two components, Moreover, the list of low-level values is set to concatenation. The sharing predicate of $\tau_0 \times \tau_1$ is similarly derived from τ_0 and τ_1's sharing predicates.

Box The Rust type `Box<T>` is an *owned pointer* to a type `T`. This pointer behaves in a manner equivalent to values of type `T` but with a level of indirection in memory. In particular, because the box owns its contents, there is no aliasing: it has exclusive control over its contents. This means that in RustHornBelt, we can represent a box solely as the value of its pointed object. Note, in RustHornBelt, we call `Box` **own**. The ownership predicate of a box thus is a wrapper around the ownership predicate for the pointed type:

$$[\![\textbf{own}\ \texttt{T}]\!].\text{own}(\hat{a}, t, \overline{v}) \triangleq \exists\, \ell \text{ s.t. } \overline{v} = [\ell].\ \exists \overline{w}.\ \ell \mapsto \overline{w} * \triangleright [\![\texttt{T}]\!].\text{own}(\hat{a}, t, \overline{w}) * \mathsf{Dealloc}(\ell, \mathsf{size}(\texttt{T}))$$

As a pointer, the low-level value should be a location ℓ. We own the points-to token $\ell \mapsto \overline{w}$ and the pointed object $[\![\texttt{T}]\!].\text{own}(\cdots)$ as well as the right to deallocate $\mathsf{Dealloc}(\cdots)$. The representation value a of an owned pointer is the same as that of the pointed object. The sharing predicate for owned pointers is tricky in this model (due to *delayed sharing* [50, §12.2]), so we omit it.

Shared Borrows We conclude our overview of simple Rust types by considering another form of pointer. As we described earlier, each Rust type is given a sharing predicate, representing its behavior when put inside a shared borrow; this allows us to capture interior mutability. We define the ownership predicate like boxes:

$$[\![\&^{\alpha}_{\textbf{shr}}\ \texttt{T}]\!].\text{own}(\hat{a}, t, \overline{v}) \triangleq \exists\, \ell \text{ s.t. } \overline{v} = [\ell].\ [\![\texttt{T}]\!].\text{shr}(\hat{a}, \alpha, t, \ell)$$

The borrow's value is a location ℓ, where a shared instance of `T` can be found.

Mutable Borrows

Now for the *pièce de résistance*, we model `&`α `mut T`, the type of *mutable references*, as follows:

$$\begin{aligned}[\![\texttt{\&}\alpha\ \texttt{mut T}]\!].\text{own}(\hat{p}, t, [\ell]) \triangleq\ & \exists\, x \text{ s.t. } \hat{p}\text{:}\star 2 = \uparrow x.\\ & \mathsf{VO}_x(\hat{p}\text{:}\star 1) * \&^{\alpha}\big(\exists \hat{a}, \overline{v}.\ \ell \mapsto \overline{v} * [\![\texttt{T}]\!].\text{own}(\hat{a}, t, \overline{v}) * \mathsf{PC}_x(\hat{a})\big)\end{aligned}$$

A lot is going on here. First of all, as expected, the RustHorn-style representation $\hat{p}$ of a mutable reference is a clairvoyant pair of the current and final states of the borrow, where the latter is some prophecy x (hence, $\hat{p}\text{:}\star 2 = \uparrow x$).

The other key difference from the RustBelt model of mutable references is the presence of two ghost state assertions: the *value observer* $\mathsf{VO}_x(\hat{p}.1)$ and the *prophecy controller* $\mathsf{PC}_x(\hat{a})$. These assertions aim to make it possible to refer to the *current* state of the borrow *both inside and outside of the borrow proposition*. In particular, note that, on the one hand, we need to *existentially* quantify over that current state inside the borrow proposition because otherwise, the borrower would not be able to change it when they mutate ℓ; but on the other hand, we also need to be able to connect the current state to the first component of the representation value $\hat{p}.1$. The VO and PC assertions make this possible using a fairly typical Iris-style "linked ghost state" construction, whereby two separately ownable propositions can independently assert the identity of some shared state, with the assurance that (a) their assertions must agree and (b) they can be updated, but only jointly. Formally, we have:

MUT-AGREE
$$\mathsf{VO}_x(\hat{a}) * \mathsf{PC}_x(\hat{a}') \vdash \hat{a} = \hat{a}'$$

MUT-UPDATE
$$\mathsf{VO}_x(\hat{a}) * \mathsf{PC}_x(\hat{a}) \Rrightarrow \mathsf{VO}_x(\hat{a}') * \mathsf{PC}_x(\hat{a}')$$

Interpreting RustHornBelt Judgments

Using the model of types, we can build up interpretations for the more complex constructs of RustHornBelt, starting with *typing contexts*:

$$\llbracket p \lhd \mathtt{T} \rrbracket \triangleq \exists \hat{a}, \llbracket \mathtt{T} \rrbracket.\mathrm{own}(\hat{a}, t, [\![p]\!])$$
$$\llbracket p \lhd^{\dagger\alpha} \mathtt{T} \rrbracket \triangleq \exists \hat{a}, \ [\dagger\alpha] \Rrightarrow\!\!\!\!\ast \ \exists \hat{b}, \ \triangleright(\hat{a} :\approx \hat{b}) * \llbracket \mathtt{T} \rrbracket.\mathrm{own}(\hat{b}, t, [\![p]\!])$$

This interpretation is a natural extension of the original RustBelt one: we existentially quantify the representation of values in the context. For frozen values after the end of α, we get back ownership of the object of type $\mathtt{T}$, whose *actual* value is $\hat{b}$, together with the knowledge that the *prophesied* value $\hat{a}$ (typically of the form $\uparrow x$) is equivalent to $\hat{b}$, via a prophecy equalizer $\hat{a} :\approx \hat{b}$.

The interpretations of the external and local lifetime contexts are unchanged from RustBelt. Using these contexts, we can define the interpretation of our judgments.

$$\begin{aligned}
&\mathbf{E}; \mathbf{L} \mid \mathbf{T} \vDash I \dashv \mathtt{r}.\ \mathbf{T}' \rightsquigarrow \Phi \ \triangleq \\
&\quad \forall \hat{\Psi}, t.\ \left\{ \exists \overline{\hat{a}}.\ [\mathrm{Na} : t] * \langle \lambda\pi.\Phi\,(\hat{\Psi}\,\pi)\,(\overline{\hat{a}\,\pi}) \rangle * \llbracket \mathbf{L} \rrbracket * \llbracket \mathbf{T} \rrbracket(\overline{\hat{a}}, t) \right\} \\
&\qquad I \ \left\{ \mathtt{r}.\ \exists \overline{\hat{b}}.\ [\mathrm{Na} : t] * \langle \lambda\pi.(\hat{\Psi}\,\pi)\,(\overline{\hat{b}\,\pi}) \rangle * \llbracket \mathbf{L} \rrbracket * \llbracket \mathbf{T}' \rrbracket(\overline{\hat{b}}, t) \right\}
\end{aligned}$$

Just like in the original RustBelt, we interpret the judgment for an instruction as a Hoare triple over an instruction I. The key addition to our definition is the presence of prophecy observations in the pre- and post-conditions of the triple. We quantify over all possible postconditions $\hat{\Psi}$, capturing the behavior of all following code. Note that the post-condition is clairvoyant, allowing it to capture prophetic information. In the postcondition of I we assert that the the post condition must hold $(\langle \lambda\pi.(\hat{\Psi}\,\pi)\,(\overline{\hat{b}\,\pi}) \rangle)$, and use the predicate transformer Φ to turn that into the appropriate precondition $(\langle \lambda\pi.\Phi\,(\hat{\Psi}\,\pi)\,(\overline{\hat{a}\,\pi}) \rangle)$. The token $[\mathrm{Na} : t]$ is used to govern concurrent behavior, specifically the *non-atomic borrows* of RustBelt.

Below, we also include the interpretation of type-context inclusion, which expresses the property that $\mathbf{T}_2$ can be obtained from $\mathbf{T}_1$:

$$\begin{aligned}
\Gamma \mid \mathbf{E}; \mathbf{L} \vDash \mathbf{T}_1 \overset{\mathrm{ctx}}{\Rrightarrow} \mathbf{T}_2 \rightsquigarrow \Phi \triangleq \\
\forall t\ \overline{\hat{a}}\ \hat{\Psi},\ \llbracket \mathbf{E} \rrbracket * \llbracket \mathbf{L} \rrbracket * \llbracket \mathbf{T}_1 \rrbracket(\overline{\hat{a}}, t) * \langle \lambda\ \pi, \Phi\ (\Psi\ \pi)\ (\overline{\hat{a}\ \pi}) \rangle \\
\Rrightarrow\!\!\!\!\ast\ \exists \overline{\hat{b}},\ \llbracket \mathbf{L} \rrbracket * \llbracket \mathbf{T}_2 \rrbracket(\overline{\hat{b}}, t) * \langle \lambda\ \pi, \hat{\Psi}\ \pi\ (\overline{\hat{b}\ \pi}) \rangle
\end{aligned}$$

This interpretation also extends the RustBelt original, associating a predicate transformer $\hat{\Phi}$ relating a predicate $\hat{\Psi}$ over the output context $\mathbf{T}_2$ to the input context $\mathbf{T}_1$. Both the transformer and predicate are *clairvoyant* allowing them to depend on prophetic information. On the right-hand side of the wand, we require an observation for $\hat{\Psi}$ when applied to the values of $\mathbf{T}_2$, while on the left-hand side, we provide an observation over the *transformed* postcondition $\hat{\Phi} \cdot \hat{\Psi}$. This judgment is of particular importance in RustHornBelt as *borrowing* is done through type-context inclusion (C-BORROW).

5.2.3 Soundness of the RustHornBelt type-spec system

In prior sections, we have shown how to define and interpret the various judgments of RustHornBelt. We now sketch the proof of soundness for this system. Proving the soundness of a semantic typing system is traditionally done using two lemmas. The first establishes the property that the semantic type system subsumes the original syntactic one, that is:

Theorem 5.2.1 (Fundamental Theorem of Logical Relations)**.** *Given a valid syntactic rule of the RustHornBelt type-spec system, the resulting semantic rule holds in Iris if we replace each* $\vdash$ *with* $\models$*.*

This theorem shows that the semantic type-spec system subsumes the syntactic one and that we can thus ignore the syntactic system going forward. In fact, in the actual implementation of RustHornBelt, we never explicitly defined the syntactic type system, and instead directly defined all the syntactic rules as their semantic equivalents.

Then the safety of the semantic type system is shown through an adequacy lemma.

Theorem 5.2.2 (Adequacy)**.** *For any* λ_{Rust} *function* F *such that* $\emptyset;\emptyset \mid \emptyset;\emptyset \models F :\mid f.\, f \lhd \forall \alpha.\, \mathbf{fn}() \to () \rightsquigarrow \lambda \Psi\, [].\Psi\ ()$ *holds, no execution of* F *(with the trivial continuation) ends in a stuck state.*

Our adequacy theorem is quite similar to the original form found in RustBelt. However, because we have extended our language with assertions, proving this theorem amounts to demonstrating that no assertion can be violated during execution, thus proving the panic-safety of our programs.

This theorem is stated using our semantic type system and amounts to showing the validity of an Iris proposition. The proof proceeds by unfolding the definitions and applying the Iris adequacy theorem. Since we phrase the statement in terms of semantic typing, we do not commit to the specific set of typing rules allowed. We can always independently prove the soundness of a new semantic typing rule without affecting the adequacy theorem, giving a more extensible and modular foundation for our type system.

5.2.4 Proving Soundness of Type-Spec Rules

To demonstrate how we can extend our type-spec system with new rules, we will give an overview of how we can prove the core rules for manipulating mutable borrows. Before we start with the proofs, we recall two rules of RustBelt's *lifetime logic*.

LftL-borrow
$$\triangleright P \ \Rrightarrow\ \&^{\alpha} P * \left([\dagger\alpha] \Rrightarrow\!\!\!\!\!\!\ast\ \triangleright P\right)$$

LftL-bor-acc
$$\&^{\alpha} P * [\alpha]_q \ \Rrightarrow\ \triangleright P * \left(\triangleright P \Rrightarrow\!\!\!\!\!\!\ast\ \&^{\alpha} P * [\alpha]_q\right)$$

The rule LftL-borrow allows us to create a borrow (in Iris) from a proposition P, while LftL-bor-acc lets us temporarily open up a borrow, requiring us to replace $\triangleright P$ to close the borrow again.

Borrow creation

First, let us tackle the creation of a mutable borrow (C-BORROW):

$$\mathbf{E}; \mathbf{L} \Vdash p \lhd \mathbf{own}_n\, \mathtt{T} \overset{\mathsf{ctx}}{\Rrightarrow} p \lhd \&^{\alpha}_{\mathbf{mut}}\, \mathtt{T}, p \lhd^{\dagger\alpha} \mathbf{own}_n\, \mathtt{T} \rightsquigarrow \lambda\Psi\,[a], \forall a', \Psi\ [(a, a'), a']$$

Unfolding the semantics of the type-spec judgment, we reach the following implication:

$$\begin{aligned}&\forall\, t\, \hat{a}\, \mathtt{a}\, \hat{\Psi},\ [\![\mathbf{own}_n\, \mathtt{T}]\!].\mathrm{own}(\hat{a}, t, \mathtt{a}) \ast \langle \lambda\,\pi, \forall a', \hat{\Psi}\ \pi\ [(\hat{a}\ \pi, a'), a']\rangle \Rrightarrow \nsim\\ &\quad \exists\, \hat{b}\, \hat{c},\ (\exists\, \mathtt{a},\ [\![\&^{\alpha}_{\mathbf{mut}}\, \mathtt{T}]\!].\mathrm{own}(\hat{b}, t, \mathtt{a})) \ast \langle \lambda\,\pi,\ \hat{\Psi}\ \pi\ [\hat{b}\ \pi, \hat{c}\ \pi]\rangle \ast\\ &\quad (\exists \mathtt{c}\,,\ [\dagger\alpha] \Rrightarrow \nsim\ \exists \hat{d},\ \triangleright(\hat{c} :\approx \hat{d}) \ast [\![\mathbf{own}_n\, \mathtt{T}]\!].\mathrm{own}(\hat{d}, t, \mathtt{c}))\end{aligned}$$

Our proof must use the resources from $[\![\mathbf{own}_n\, \mathtt{T}]\!].\mathrm{own}(...)$ to construct the ownership predicate for the borrow and have enough leftover to create the wand, which will restore ownership when the borrow expires. We proceed as follows.

First, we create a prophecy x and get the value observer $\mathsf{VO}_x(\hat{a})$ and the prophecy controller $\mathsf{PC}_x(\hat{a})$ for x:

$$\begin{array}{l}\textsc{mut-intro}\\ \mathsf{True} \ \Rrightarrow\ \exists x.\ \mathsf{VO}_x(\hat{a}) \ast \mathsf{PC}_x(\hat{a})\end{array}$$

Pick $\hat{c} \triangleq \uparrow x$ and $\hat{b} \triangleq {}^{\star}(\hat{a}, \uparrow x)$. From the input observation, we immediately get the output observation simply by instantiating a' into the prophecy's value $\uparrow x\ \pi$:

$$\langle\, \lambda\pi.\, \forall a'.\, (\hat{\Psi}\,\pi)\, [a', (\hat{a}\,\pi, a')]\,\rangle \ \vdash\ \langle\, \lambda\pi.(\hat{\Psi}\,\pi)\, [\hat{c}\,\pi, \hat{b}\,\pi]\,\rangle$$

We then unfold the model of $[\![\mathbf{own}_n\, \mathtt{T}]\!].\mathrm{own}(\hat{a}, t, \mathtt{a})$ to get:

$$\mathtt{a} \mapsto \overline{v} \ \ast\ \mathsf{Dealloc}(\mathtt{a}, |\mathtt{T}|) \ \ast\ \triangleright [\![\mathtt{T}]\!].\mathrm{own}(\hat{a}, t, \overline{v})$$

And let P be $\exists \hat{a}', \overline{v}.\ \mathtt{a} \mapsto \overline{v} \ast [\![\mathtt{T}]\!].\mathrm{own}(\hat{a}', t, \overline{v}) \ast \mathsf{PC}_x(\hat{a}')$.

By LftL-borrow, constructing and depositing $\triangleright P$, we create a *borrow proposition* $\&^{\alpha} P$ and its *inheritance* $[\dagger\alpha] \Rrightarrow \nsim \triangleright P$. Now we have all we need to construct the required resources for the *mutable reference* `b`:

$$\mathsf{VO}_x(\hat{a}) \ \ast\ \&^{\alpha} P \ \vdash\ [\![\&\alpha\ \mathtt{mut}\ \mathtt{T}]\!].\mathrm{own}\big({}^{\star}(\hat{a}, \uparrow x), t, \mathtt{b}\big)$$

We use the remaining resources for the *frozen* box:

$$([\dagger\alpha] \Rrightarrow \nsim \triangleright P) \ \ast\ \mathsf{Dealloc}(\mathtt{a}, |\mathtt{T}|) \ \vdash\ [\dagger\alpha] \Rrightarrow \nsim\ \exists \hat{d},\ \triangleright(\hat{c} :\approx \hat{d}) \ast [\![\mathbf{own}_n\, \mathtt{T}]\!].\mathrm{own}(\hat{d}, t, \mathtt{c})$$

To prove this, we "execute" the given view-shift wand with $[\dagger\alpha]$ to get $\triangleright P$. Take out the value $\hat{a}'$ out of P. We chose $\hat{d} = \hat{a}'$. The rule PROPH-CTRL-EQZ allows us to transform a prophecy controller $\mathsf{PC}_x(\hat{a}')$ into an equalizer $\uparrow x :\approx \hat{a}'$, effectively denying future mutations to this borrow.

$$\begin{array}{l}\textsc{proph-ctrl-eqz}\\ \mathsf{PC}_x(\hat{a}) \ \Rrightarrow\ \uparrow x :\approx \hat{a}\end{array}$$

Consuming $\mathsf{PC}_x(\hat{a}')$ inside P, we get the desired $\uparrow x :\approx \hat{a}'$. Using the remaining parts of P and $\mathsf{Dealloc}(\mathtt{a}, |\mathtt{T}|)$, we can construct the box.

Write To write to a mutable reference `*b = c` (S-ASSGN, TWRITE-BOR), we get access to the borrow proposition's content by LftL-bor-acc and update it, Moreover, renew the observed current state by MUT-UPDATE.

Borrow dropping Consider dropping off a mutable reference (F-RESOLVE).

First, by LftL-bor-acc, we get temporary access to the borrow proposition's content, which contains the prophecy controller $\mathsf{PC}_x(\hat{a})$. We can use TY-OWN-PROPH to recover the prophecy tokens for the dependencies of the pointed type. These dependencies allow us to use the following ghost update rule to *resolve the prophecy* x, disposing of the value observer in the process (as we should only be able to resolve once!):

$$\text{MUT-RESOLVE}\quad \frac{\mathsf{dep}(\hat{a}, Y)}{\mathsf{VO}_x(\hat{a}) * \mathsf{PC}_x(\hat{a}) * [Y]_q \Rrightarrow \langle \uparrow x \stackrel{\star}{=} \hat{a} \rangle * \mathsf{PC}_x(\hat{a}) * [Y]_q}$$

Now we get an observation $\langle \uparrow x \stackrel{\star}{=} \hat{a} \rangle$, which makes the prophecy's value $\uparrow x$ effectively equal to the current state $\hat{a}$. We can use it for the postcondition to satisfy the rule's spec $\lambda \Psi, [b].\, b.2 = b.1 \to \Psi[]$.

Unfreezing Unfreezing of objects at a lifetime's end (F-ENDLFT) can be proved easily. We first get a dead-lifetime token $[\dagger\alpha]$ by consuming $[\alpha]_1$ in the lifetime context. With this token we "execute" the view-shift wand of each frozen object $[\dagger\alpha] \Rrightarrow\!\!\!\!\ast\ \exists \hat{b}.\, \hat{a} :\approx \hat{b} * [\![\mathtt{T}]\!].\mathrm{own}(\hat{b}, t, \mathtt{a})$, to get an active object $[\![\mathtt{T}]\!].\mathrm{own}(\hat{b}, t, \mathtt{a})$. Using this object, we can apply TY-OWN-PROPH to recover the tokens that we need to turn our equalizer $\hat{a} :\approx \hat{b}$ into an observation $\langle \hat{b} \stackrel{\star}{=} \hat{a} \rangle$ for each object. Combining these observations, we can prove the specification to the rule $\lambda \Psi, \overline{a}.\, \Psi\, \overline{a}$.

5.3 Rust APIs with Unsafe Code

So far, we discussed Rust's *safe* features. Now, we present RustHorn-style specs for various Rust APIs with *unsafe* implementations, which we have verified in RustHornBelt.

`Vec` API One common use of unsafe code in Rust APIs is to provide a more efficient implementation than Rust's safe typing rules allow. A canonical example of this is the ubiquitous *vector* (or growable array) type `Vec<T>`. The `Vec` API manages a dynamically allocated memory block to store and provide access to an unbounded number of objects of the type `T`, which it achieves through the effective use of *raw pointers*. Raw pointers are Rust pointers whose aliasing is untracked by the type system and potentially unsafe to use. The `Vec` API supports a variety of operations; for RustHorn-style verification, we are particularly interested in those that perform destructive *state mutation*.

First, let us consider the following operations:

```
1 fn push<T>(v: &mut Vec<T>, a: T)
2 fn pop<T>(v: &mut Vec<T>) -> Option<T>
```

They both destructively update a vector through a *mutable reference* `v: &mut Vec<T>` to it. The operation `push` adds an element `a: T` to the end of the vector (and returns nothing), and `pop` removes the last element `a` from the vector, returning `Some(a)` (and `None` if the vector is empty).

Before describing the behavior of these operations, we must first choose a representation for the type `Vec<T>`. Naturally, we represent a vector as a list of its contents: $\lfloor \mathtt{Vec<T>} \rfloor \triangleq \mathit{List} \lfloor \mathtt{T} \rfloor$. Correspondingly, the `push` and `pop` operations get the following specs:

$$v.2 = v.1 +\!\!+ [a] \to \Psi[]$$

$$\text{if } v.1 = [] \text{ then } v.2 = [] \to \Psi[\mathsf{None}] \quad \text{else } v.2 = \mathsf{last}\ v.1 \to \Psi[\mathsf{Some}(\mathsf{init}\ v.1)]$$

where $\mathsf{last}\, w$ is the last element of the list w, and $\mathsf{init}\, w$ is w without its last item. In the case of both functions, $v.1$ represents the initial state of the mutable reference `v`; and since `v` is dropped before the function returns, we also learn that the prophesied "final" value of `v` (*i.e.,* $v.2$) is precisely the state of `v` at the end of the function. Thus, $v.1$ and $v.2$ act like just an input and output.

Things get more interesting when an operation not only inputs but also *outputs* a mutable reference. Let us consider the following operation for random access:

```
fn index_mut<α,T>(v: &α mut Vec<T>, i: int) -> &α mut T
```

Physically, it is just an address calculation: get the head address of the buffer of a vector and add the offset of i blocks. In Rust, however, such addresses are linked with *ownership*. In `index_mut`, the *mutable borrow* over a vector is *subdivided* into a smaller borrow over a specific element of the vector, inheriting the lifetime α.

We give to `index_mut` the following RustHorn-style spec:

$$0 \le i < |v.1| \ \ \wedge \ \ \forall a'.\ v.2 = v.1\{i \coloneqq a'\} \to \Psi[(v.1[i], a')]$$

The precondition $0 \le i < |v.1|$ is for the bounds check. In addition, we *prophecy* the final state a' of the new, *subdivided* borrow for the output. Now the old borrow's prophesied final state $v.2$ is *partially* determined with respect to a' (an example of *partial prophecy resolution*). It is set to $v.1\{i \coloneqq a'\}$, which can be read as $v.1$ with the i-th element's determination left to the prophesied value a'.

`IterMut` API Rust's `IterMut` API for *mutable iterators*—though implemented with unsafe code—exemplifies how Rust's type system provides stronger guarantees than those of "safe" languages like Java, leveraging ownership to eliminate common pitfalls like *iterator invalidation*. In Chapter 6, we will show how to generalize this specification to reason about abstract compositions of iterators.

With `iter_mut`, you can create a mutable iterator out of a mutable reference to a vector:

```
fn iter_mut<α,T>(v: &α mut Vec<T>) -> IterMut<α,T>
```

As the lifetime parameter α of `IterMut` indicates, a mutable iterator is an advanced form of mutable borrow, having temporary ownership of some memory sequence. Rust's type system ensures that, while the iterator `IterMut<α,T>` is active, the ownership of the iterated vector is frozen, preventing the vector from being modified while it is being iterated over—a phenomenon known as iterator invalidation.

With `next`, you can perform one step of mutable iteration:

```
fn next<α,T>(it: &mut IterMut<α,T>) -> Option<&α mut T>
```

This yields a mutable reference to the head element `a: &α mut T`, moving the focus to the next element and returning `Some(a)` (or `None` if the iterator has reached the end).

With the iterated application of `next`, it is possible to convert the mutable iterator into a bunch of mutable references to the individual elements of the vector, which can all be used simultaneously—*i.e.,* one need not give up the mutable reference to one element to obtain a mutable reference to the next. Hence, in RustHornBelt, we naturally represent a mutable iterator as a *list of mutable references* to each element of the iterated container, setting $\lfloor \texttt{IterMut<}\alpha\texttt{,T>} \rfloor \triangleq \mathit{List}(\lfloor T \rfloor \times \lfloor T \rfloor)$. This leads to the following straightforward specification for `next`:

$$\begin{array}{l} \mathsf{if}\ \ it.1 = [] \ \ \mathsf{then}\ \ it.2 = [] \to \Psi[\mathsf{None}] \\ \qquad\qquad \mathsf{else}\ \ it.2 = \mathsf{tail}\, it.1 \to \Psi[\mathsf{Some}(\mathsf{head}\, it.1)] \end{array}$$

We can also give the following spec to `iter_mut`, which might look tricky at first:

$$|v.2| = |v.1| \;\to\; \Psi[\mathsf{zip}\ v.1\ v.2]$$

Essentially, what we are doing is an *elementwise split* of the mutable borrow over the vector (one example of *borrow subdivision*, like `Vec`'s `index_mut`). The borrow's final state $v.2$ is split elementwise into a list of prophesied values $v.2[0], v.2[1], \cdots, v.2[|v.1| - 1]$, and the length ($|v.2| = |v.1|$) is guaranteed to stay constant. The output iterator works as if it were a list of mutable references to each vector element. The function zip works like $\mathsf{zip}\,[a, b, c]\,[a', b', c'] = [(a, a'), (b, b'), (c, c')]$.

Combining `iter_mut` and `next`, we can write and functionally verify various programs that iteratively mutate vectors. For example, let us consider the following function:

```
fn inc_vec(v: &mut Vec<int>) { for a in v.iter_mut() { *a += 7; } }
```

This uses a mutable iterator, `v.iter_mut()`, to increment each element of the vector `*v` by 7. The `for` statement is syntactic sugar for repeatedly calling the `next` method and unwrapping the result to get `a: &mut int` until `None` is returned. Using the specs of `iter_mut` and `next`, we can derive the following spec on `inc_vec`: $v.2 = \mathsf{map}\,(+7)\,v.1 \to \Psi[]$.

`Cell` API Though helpful in avoiding memory safety bugs and data races, Rust's prohibition of aliased mutable state is too restrictive in many situations, such as implementing cyclic data structures. Rust also provides several APIs with *interior mutability* to meet such needs, allowing mutation even through a *shared* reference, albeit in carefully controlled ways.

Arguably the most straightforward such API is `Cell`, whose safety is guaranteed by various restrictions (*e.g.,* it can only be used within a single thread). It provides the following operations:

```
fn new<T>(a: T) -> Cell<T>
fn get<T: Copy>(c: &Cell<T>) -> T
fn set<T>(c: &Cell<T>, a: T)
```

You can convert a `T` to a cell `Cell<T>` by calling `new`. Then, using a *shared* reference to a cell `&Cell<T>` with *copyable* content, you can both read from the cell by `get` *and* write a new value to the cell by `set`.

Such interior mutability helps write code but makes functional verification (especially in the RustHorn style) more challenging. RustHornBelt proposes a straightforward approach to solve this problem: *invariants*.

Concretely, we represent `Cell<T>` as an invariant predicate, with $\lfloor$`Cell<T>`$\rfloor \triangleq \lfloor$`T`$\rfloor \to \mathsf{Prop}$. For `get`, we know that the real value a satisfies the invariant, which amounts to the following spec: $\forall a.\ c(a) \to \Psi[a]$, where c is the invariant representing the cell, of sort $\lfloor$`T`$\rfloor \to \mathsf{Prop}$. For `set`, we promise that writing to the cell will preserve the invariant, hence the following spec: $c(a) \land \Psi[]$. For `new`, we choose the cell's invariant Φ, which should be satisfied by the initial value. Thus, we give `new` the following spec, for any Φ: $\Phi(a) \land \Psi[\Phi]$.

Using these specs, we can do some functional verification. For example, let us consider the following function:

```
fn inc_cell(c: &Cell<int>, i: int) { c.set(c.get() + i); }
```

We should ensure that the update by `set` does not invalidate the cell's invariant. That is achieved with the following spec for `inc_cell`: $\big(\forall n.\ c(n) \to c(n+i)\big) \land \Psi[]$. Before $\land$ is the main precondition, which is satisfied if, for example, $c = \lambda n.(n \text{ is odd})$ and $i = 4$.

RustHornBelt allows the invariant Φ for a cell to depend on runtime values. For example, we can call `inc_cell` with the invariant $\lambda n.\ n \bmod k = 1$, where k represents another program variable `k: int`. We restrict this dependency to *non-prophesied* values: we cannot choose an invariant that

depends on the prophecy of a mutable borrow. To this day, we do not know if extending our approach to allow such invariants is possible. However, this does not prevent common use cases `Cell` like *memoization*, which does not generally require storing mutable borrows in the memoized cache.

Finally, we have also proven sound similar invariant-based specs for the `Mutex` API, a thread-safe variant of `Cell`, which uses a lock to control mutable access to the shared cell.

5.3.1 Proving specifications for APIs with unsafe code

We can also *semantically* verify all our RustHorn-style specs for *safe Rust APIs with unsafe implementations*.

For an interesting example, let's consider the `iter_mut` method for converting a mutable reference `&`α `mut Vec<T>` into a *mutable iterator* `IterMut<`α`,T>` (§ 5.3). To verify the method, it suffices to prove the following Hoare triple:

$$\begin{array}{ll} & \{ \langle \lambda\pi.\, |{\uparrow x}\,\pi| = |\hat{v}\,\pi| \to (\hat{\Psi}\,\pi)\,[\mathsf{zip}\,(\hat{v}\,\pi)\,({\uparrow x}\,\pi)] \rangle \ * \\ & \qquad [\alpha]_q \ * \ [\![\texttt{\&}\alpha\ \texttt{mut Vec<T>}]\!].\mathrm{own}\big(^\star(\hat{v}, {\uparrow x}), t, \texttt{v}\big) \ \} \\ \texttt{iter\_mut(v)} & \{ \texttt{it}.\ \exists\, \hat{b}.\ \langle \lambda\pi.(\hat{\Psi}\,\pi)\,[\hat{b}\,\pi] \rangle \ * \\ & \qquad [\alpha]_q \ * \ [\![\texttt{IterMut<}\alpha\texttt{,T>}]\!].\mathrm{own}(\hat{b}, t, \texttt{it}) \ \} \end{array}$$

Here, we sketch the proof. By `Vec<T>`'s semantics, the vector's value $\hat{v}$ decomposes into $^\star[\hat{a}_1, \ldots, \hat{a}_n]$. Now we create new prophecies $y_1, \ldots, y_n$ along with a value observer $\mathsf{VO}_{y_i}(\hat{a}_i)$ and a prophecy controller $\mathsf{PC}_{y_i}(\hat{a}_i)$ for each i. We then pick $\hat{b} \triangleq {}^\star[{}^\star(\hat{a}_1, {\uparrow y_1}), \ldots, {}^\star(\hat{a}_n, {\uparrow y_n})]$, and construct the mutable iterator $[\![\texttt{IterMut<}\alpha\texttt{,T>}]\!].\mathrm{own}(\hat{b}, t, \texttt{it})$, which is equivalent to iterated separating conjunction of the (imaginary) mutable reference $[\![\texttt{\&}\alpha\ \texttt{mut T}]\!].\mathrm{own}\big(^\star(\hat{a}_i, {\uparrow y_i}), t, [\ell + i \cdot |\texttt{T}|]\big)$ to the i-th element, over $i \in 1..n$ (where ℓ is the head location). Also, we need the observation $\langle {\uparrow x} \ {}^\star{=}\ {}^\star[{\uparrow y_1}, \ldots, {\uparrow y_n}] \rangle$. To achieve this, we should *split the borrow proposition* of `&`α `mut Vec<T>` to get the borrow propositions for `IterMut<`α`,T>`, *partially resolving* the old prophecy x. Although we omit details, our semantic model can verify even *borrow subdivision* like this.

5.4 Implementation and Evaluation

We evaluated our approach discussed in § 5.2 by fully mechanizing the semantic soundness proof of the type-spec system in the Coq proof assistant, verifying various safe Rust APIs that encapsulate unsafe code.

We built RustHornBelt's Coq development by extending that of RustBelt [53]. It has ~19kLOC of Coq code in total. We were able to reuse the key sub-components, the *lifetime logic* (~2kLOC) and the untyped core calculus (λ_{Rust}) (~3kLOC), as well as the overarching proof structure for verifying the type system. The development took two implementors ~six months to complete, adding ~7kLOC to the final proofs.

We first modeled basic Rust types and verified type-spec rules for operations on them, extending RustBelt with functional specs. The verified basic types include: box pointer `Box<T>`, shared and mutable references `&`α `(mut) T`, tuple `(T1,...,Tn)`, sum $\texttt{T}_1 + \cdots + \texttt{T}_n$[4], array `[T;n]`, integer `int`, boolean `bool`, function `fn(`$\overline{\texttt{T}}$`) -> T'`, and recursive types[5].

Then we also modeled advanced Rust types and verified type-spec rules for key API functions encapsulating unsafe code, including:

[4] This amounts to Rust's `enum` type.

[5] This supports *non-covariant* recursion, *e.g.,* recursion with a self reference under the mutable reference `&`α `mut`.

API	#Funs	LOC		
		Type	Code	Proof
`Vec`	9	147	59	459
`SmallVec`	9	209	75	619
`&`α (`mut`) `[T]` / `Iter(Mut)`	9	253	38	428
`Cell`	8	102	20	188
`Mutex` / `MutexGuard`	7	258	30	222
`JoinHandle`	2	73	12	52
`MaybeUninit`	5	140	8	108
Misc	3	0	14	85

Figure 5.8: Coq mechanization of Rust APIs. #Funs: Number of the functions verified. Type: LOC of the semantic model and proof for the type(s). Code: LOC of the λ_{Rust} implementation of the functions. Proof: LOC of the verification proof of the type-spec rules.

- Vector `Vec<T>` — `new`, `drop`, `len`, `push`, `pop`, `index(_mut)`, `as_(mut_)slice/iter(_mut)`[6]
- Small-vector `SmallVec<T,`n`>` — `new`, `drop`, `len`, `push`, `pop`, `index(_mut)`, `as_(mut_)slice/iter(_mut)`[6]
- Shared/mutable slice `&`α (`mut`) `[T]` — `len`, `split_at(_mut)`, `[T;`n`]::as_(mut_)slice`
- Shared/mutable iterator `Iter(Mut)<`α`,T>`[7] — `Iter(Mut)::next`, `Iter(Mut)::next_back`
- Cell `Cell<T>` — `new`, `into_inner`, `from_mut`, `get_mut`, `get`, `set`, `replace`
- Mutex `Mutex<T>` — `new`, `into_inner`, `get_mut`, `lock`
- Mutex guard `MutexGuard<`α`,T>` — `deref(_mut)`, `drop`
- Thread / `JoinHandle<T>` — `spawn`, `join`
- Maybe-uninitialized `MaybeUninit<T>` — `new`, `uninit`, `assume_uninit(_ref, _mut)`
- Misc — `swap`, `panic!`,[8] `assert!`[8]

We implemented each function in the core calculus λ_{Rust}. As in RustBelt, our λ_{Rust} implementation of each function is meant to extract the essence of the real-world Rust implementation, simplifying away uninteresting details. For example, our λ_{Rust} version of `Vec::push` uses a simpler reallocation strategy than the original Rust version.

In Fig. 5.8, we report the code size of the implementation and proof of a selection of Rust APIs. A function with a large implementation involving mutable borrows tends to require a larger code

[6] We equate the two methods because we used the same model for the shared/mutable slice and iterator.
[7] For simplicity, for the shared/mutable iterator `Iter(Mut)<`α`,T>`, we used the same model as the shared/-mutable slice `&`α (`mut`) `[T]`.
[8] Abortion is implemented just as a stuck term.

size and more significant proof effort. Roughly speaking, modeling a Rust type took ~1 hour, and verifying each function took about 10 minutes–2 hours for us. We still need a large amount of boilerplate code for the proof. Further automation of this part is left to future work.

We also validated our type-spec system by (somewhat manually) verifying small Rust programs with ~800 LOC of Coq code. The verified programs include what correspond to `inc_vec` and `inc_cell` shown in §5.3, demonstrating the Rust APIs `Vec`, `IterMut` and `Cell`.

5.5 Correspondence with Creusot

We claim that RustHornBelt (this chapter) provides strong evidence for the soundness of the approach used in Creusot (Chapter 4), but there are essential differences between the two. At first glance, the two are different: in RustHornBelt, we relate λ_{Rust} programs to predicate transformers; in Creusot, we translate MIR programs to functional ones. However, those functional programs are then turned into first-order logical formulas through a verification condition generator similar to what RustHornBelt is doing. Thus, discussing the similarities and differences between the two approaches still makes sense.

Of course, as we discussed in Chapter 5, λ_{Rust} makes many simplifications to MIR in the interest of more straightforward modeling, and these carry over to Creusot, which works with MIR. However, more subtle differences also exist, leaving a formalization gap between the two approaches. Perhaps the most significant difference is related to the use of *places* and resolution. RustHornBelt distinguishes between *reborrowing*, which shortens the lifetime of a borrow, and *unnesting*, which squashes a level of indirection in nested pointers:

C-REBORROW-MUT

$$\frac{\mathbf{E};\mathbf{L} \vdash \alpha' \sqsubseteq \alpha}{\mathbf{E};\mathbf{L} \vdash p \lhd \&^{\alpha}_{\mathsf{mut}}\, \mathtt{T} \overset{\mathsf{ctx}}{\Rrightarrow} p \lhd \&^{\alpha'}_{\mathsf{mut}}\, \mathtt{T}, p \lhd {}^{\dagger\alpha'}\, \&^{\alpha}_{\mathsf{mut}}\, \mathtt{T} \rightsquigarrow \Psi\, [(a, a'), \forall a'', \Psi\, [(a, a''), (a'', a')]}$$

S-DEREF-BOR-BOR

$$\frac{\mathbf{E};\mathbf{L} \vdash \alpha\ \mathsf{alive} \qquad \mathbf{E};\mathbf{L} \vdash \alpha \sqsubseteq \alpha'}{\mathbf{E};\mathbf{L} \mid p \lhd \&^{\alpha}_{\mu} \&^{\alpha'}_{\mathsf{mut}}\, \mathtt{T} \vdash {}^{*}p \dashv \mathtt{a}.\, \mathtt{a} \lhd \&^{\alpha}_{\mu}\, \mathtt{T} \rightsquigarrow \Psi\, [((v, w), (v', w'))], w' = w \to \Psi\, [(v, v')]}$$

The formalization goes so far as to place these rules in entirely different categories, with the first being purely structural while the second is an actual statement.

In MIR (and Creusot), both of these are subsumed by *place expressions* and the `Ref` rvalue. Place expressions allow us to arbitrarily dereference and destruct a type behind a `Ref`. A reborrow is equivalent to the place expression `&mut * x` while an unnest is `&mut ** x`. Creusot unifies and generalizes the RustHornBelt rules for unnesting and resolution. Instead of using rules for borrowing, reborrowing, and unnesting, Creusot uses a single R-REF-MUT, below:

R-REF-MUT

$$\frac{\mathbf{T} \vdash \mathtt{p}_2 \overset{\text{pl}}{\hookrightarrow} e \qquad \mathbf{T} \vdash b \hookleftarrow \mathtt{p}_1/(e, \mathtt{any}) \qquad \mathbf{T} \vdash \mathtt{p}_1 \overset{\text{pl}}{\hookrightarrow} p \qquad \mathbf{T} \vdash S \hookleftarrow \mathtt{p}_2/p.1}{\mathbf{T} \vdash \mathtt{p}_1\ \mathtt{=}\ \mathtt{\&mut}\ \mathtt{p}_2 \overset{\text{stmt}}{\hookrightarrow} b/S}$$

which performs all three by allowing borrows of place expressions. An unnesting is then expressed by a literal interpretation of its Rust syntax `&mut ** x` as:

```
b = { cur = x.cur.cur; fin = any };
x = { cur = { cur = b.fin ; fin = x.cur.fin } ; fin = x.fin };
```

An observant reader will observe that this translation is not equivalent to the RustHornBelt specification; we do not have `x.cur.fin` $= $ `x.fin.fin` $(w = w')$ or `b.fin` $= $ `x.fin.cur` $(v = v')$.

The missing parts come from Creusot's resolution mechanism (§ 4.3.2). Since we are done with `x`, we will resolve its value after the unnesting, adding: `x.cur` = `x.fin`. By substitution and simplification, we can derive the missing equalities.

5.6 Related Work

Formalized proofs of verifiers Outside of Rust, there is a long lineage of foundational tools for program verification. Cao et al. [19] is a toolchain for verifying C programs built on top of the certified CompCert [62] C compiler. It provides its own separation logic as an alternative to Iris. More recently, Sammler et al. [83] provides a semi-automated tool for verifying C code in Coq, based on Iris. As the name suggests, it is a predecessor to RefinedRust. RefinedC uses ownership typing to verify C programs but does not include a borrowing mechanism like Rust or RefinedRust.

Prophecies First introduced to prove refinement between state machines [2], *prophecies* have been studied for decades, although they remain a somewhat exotic technique.

Jung et al. [54] modeled prophecies in Iris (influenced by existing literature [91, 96]), mainly to prove *logical atomicity* of tricky concurrent data structures (though there have been other applications [92]). In their approach, prophecy creation and resolution take the form of *ghost* program instructions to ensure consistency of prophecy reasoning, which provide a sort of "ground truth" for the prophecy but also require cumbersome user annotations. Moreover, their prophecies distinguish between the *name* of a prophecy and the value it resolves to (mediated by a kind of "prophecy heap" mapping prophecy names to values). As such, they do not provide a way to resolve a prophecy to a value that mentions the values of other (as yet unresolved) prophecy variables—a feature we require in RustHornBelt to model nested borrows and subdivisions.

For separation logic verification of fine-grained concurrency like Jung et al. aimed at, Turon et al. [90] and Liang and Feng [66] employed a technique of *speculation*. Their approach allows the proof to speculate about multiple possible logical states, combine them through the "speculative choice" connective $P \oplus Q$, and then cull the set of possible states once more information becomes available later in the proof. However, it does not provide an analog of prophecy variables. In contrast, our prophecy framework provides *persistent* observations $\langle \hat{\phi} \rangle$, which can express knowledge of prophecy variables' values that hold under *all* possible futures.

Chapter 6

Iterators

> This chapter is based on the article ‘“Specifying and Verifying Higher-order Rust Iterators”’ published in TACAS’23 [26]. The presentation has been significantly extended from the original publication.

Rust empowers systems software programmers by offering them safe and powerful linguistic abstractions to solve their problems. Rust’s *borrowing* mechanism is the most notorious of these abstractions, enabling safe usage of pointers without a garbage collector or performance penalty. A close second is perhaps Rust’s *iterator* system, through which Rust provides composable mechanisms to express the traversal and modification of collections. Iterators also underlie Rust’s `for` loop syntax and are thus the primary manner Rust developers write loops or interact with data structures. It is, therefore, essential for a verification tool for Rust to provide good support for iterators.

Rust iterators generate sequences of values. More concretely, they are objects providing a method `fn next(&mut self) -> Option<Self::Item>`. This method takes a *mutable reference* (`&mut self`) to the iterator, allowing it to change its internal state, and optionally returns a value of type `Self::Item`, the type of the values generated by the iterator. If the iterator returns `None` instead of returning such a value, it means iteration has finished for now, though it may resume later. Rust’s `for` loops are just syntactic sugar for repeatedly calling `next` at the beginning of each iteration until such a call returns `None`. For example, the following two pieces of code present a Rust loop for iterating over integers between 0 (included) and `n` (excluded), using a *range* iterator:

```
for i in 0..n { <body> }
```

```
let mut iter = Range { start: 0, end:
    n };
loop { match iter.next() {
  None => break,
  Some(i) => <body>
} }
```

The code on the left-hand side uses an idiomatic `for` loop, while the other shows its desugared version.

Iterators present unique challenges for verification tools: indeed, because the use of iterators is pervasive in Rust, it is necessary to allow code verification using iterators with as little interaction as possible. In particular, the most common patterns, such as iterating over integers in a given range or reading the elements of a vector should not need any annotation other than the loop invariants the user would write if not using iterators. On the other hand, Rust’s iterator library is complex, with many features representing as many challenges for verification: iterators can be built from various data structures and modified through *iterator combinators*, which makes it possible

to create iterators from simpler ones, by, e.g., skipping the first few elements or applying a given function to each of the elements. These challenges are exemplified by Example 1, presented below:

Example 1 (Our Target)**.**

```
1    let mut cnt = 0;
2    let w = vec![1,2,3].iter().map(|x|{cnt += 1; x + 1}).collect();
3    assert_eq!(w, vec![2,3,4]);
4    assert_eq!(cnt, 3);
```

On line 2, quite a lot happens simultaneously. First, we produce an iterator over the elements of the vector `vec![1,2,3]` with the syntax `.iter()`, which we transform through a call to `map`. The method `map` is an *iterator combinator*: it returns a new iterator that calls the given closure on each of the elements generated by the underlying iterator and forwards the value returned by the closure. Interestingly, the closure we pass to `map` *captures mutable state*: it modifies the variable `cnt`. Finally, the method `collect` gathers the elements generated into a new vector `w`.

We aim at requiring only lightweight annotations for verifying this kind of code: the appeal of *iterator chains* like on line 2 are the ergonomics; they are compact and highly readable. For verification of iterator-based code to be successful, it must preserve this ergonomics. However, despite its apparent simplicity, this code is challenging to verify: it combines higher-order functions and mutable state, uses potentially overflowing integers, and assertions on line 4 check full functional behavior.

More generally, to support iterators, a verification tool for Rust needs to provide a specification scheme that both provides good ergonomics and overcomes the following technical challenges:

- *Strong Automation*: for verification to be used, it must require little to no user interaction and lead to good verification performance.
- *Interruptibility*: iterators can produce infinite sequences of values and be interrupted before completion. Thus, specification and verification must happen as the iterator is used and not at completion.
- *Non-Determinism*: iterators can feature both specification or implementation non-determinism, so the sequence of known values might not be known in advance to the verifier. For example, the order of elements generated by an iterator over a hash table may be left unspecified for a client.
- *Compositionality*: iterators can be consumed by *combinators*, so their specifications need to follow a general pattern which makes them *composable*. For example, the specification of a combinator such as `skip(n)`, which skips the first `n` elements of a given iterator, should accept the specification of any iterator and provide a sound and useful specification for the combined iterator.
- *Higher-Order & Effects*: some iterator combinators, such as `map`, are *higher-order*, they take a closure as parameter. To verify programs using these combinators, a verification tool should overcome the challenges of higher-order functions, which potentially capture mutable states.

Contributions

In order to reach this goal, we propose a new specification scheme for iterators in Rust. Our contributions can be summarized as follows:

- In § 6.1, we provide a general specification scheme for Rust iterators in first-order logic. It supports possibly non-deterministic, infinite, and interruptible iterators. It is inspired by

Filliâtre and Pereira's specification of iterators in Why3 [32], but it is adapted to the prophetic semantics of Creusot.

- In § 6.2.1, we show that this scheme can be trivially instantiated for basic iterators such as a range of integers.
- In § 6.2.2, we show how this scheme can be instantiated to give full functional specification to *mutating iterators*. These iterators allow mutating the content of a data structure by iterating over mutable references *pointing to the content of the data structure*.
- In § 6.2.3, we show that our specification scheme is *composable* so that it can be used to specify iterator combinators transforming arbitrary iterators into more complex ones. We give two examples: `take`, which truncates an iterator to, at most, a given number of elements, and `skip`, which skips a given number of elements at the beginning of iteration.
- In § 6.3, we explain how we can combine the techniques presented in previous sections to specify higher-order iterator combinators by taking `map` as an example. This provides a way to verify the functional correctness of programs using higher-order iterators while requiring lightweight annotations.
- We implement our specification scheme in Creusot. This implementation extends Creusot's handling of `for` loops to benefit from *structural invariants* provided by the specification of iterators. We evaluate it on several benchmarks in § 6.4.

6.1 Reasoning about Iteration

In this section, we present the general mechanism we use to specify iterators (§ 6.1.1) and how these kinds of specifications are used in a `for` loop (§ 6.1.2). Before going in-depth into these definitions, we introduce the style of specification we use in this chapter.

6.1.1 Specifying Iterators

In Rust, the mechanism of iterators is captured by a *trait* named `Iterator`, whose simplified definition can be given as:

```
1 trait Iterator {
2   type Item;
3   fn next(&mut self) -> Option<Self::Item>;
4 }
```

This trait describes the *interface* an iterator should implement: an iterator should give a type `Item` of generated elements and should implement a method `next` which optionally returns the next generated element, and possibly mutates in place the internal state of the iterator through the mutable reference `&mut self`.

As can be seen in Figure 6.1, we extend[1] the iterator trait with the purely logical *predicates* `produces` and `completed`. We require that any implementation of this trait satisfies the *laws* `produces_refl` and `produces_trans`. Thanks to the two predicates, the `next` method is specified. Any implementation of the `Iterator` trait needs to give a logical definition of `produces` and `completed` predicates, prove the laws, give a program definition for `next`, and finally prove that it satisfies its specification.

[1] In our implementation, to keep better compatibility with existing Rust code, we choose to define the iterator specification as a sub-trait of the `Iterator` trait from Rust's standard library and to give the specification of `next` using Creusot's `extern_spec!` mechanism. For simplicity, we present it here as a unique trait- the main idea of the specification is the same.

```
trait Iterator {
  type Item;
  #[predicate] fn completed(&mut self) -> bool;
  #[predicate] fn produces(self, visited: Seq<Self::Item>, _: Self)
                    -> bool;
  #[law]    // I.e., ∀ a, a ~>^ε a
  #[ensures(a.produces(Seq::EMPTY, a))]
  fn produces_refl(a: Self);

  #[law]    // I.e., ∀ a b c, a ~>^v b ∧ b ~>^w c ⇒ a ~>^(v·w) c
  #[requires(a.produces(ab, b) && b.produces(bc, c))]
  #[ensures(a.produces(ab.concat(bc), c))]
  fn produces_trans(a: Self, ab: Seq<Self::Item>,
                    b: Self, bc: Seq<Self::Item>, c: Self);

  #[ensures(match result {
      None => self.completed(),
      Some(v) => (*self).produces(Seq::singleton(v), ^self)})]
  fn next(&mut self) -> Option<Self::Item>;
}
```

Figure 6.1: Iterator trait extended with the specification.

Iterators are specified as *state machines*: a value of an iterator type is seen as a state, and the predicate `produces` defines the transition relation (noted $a \overset{s}{\rightsquigarrow} b$). The predicate `completed` (noted $completed(r)$) gives the set of final states. The `completed` predicate takes a mutable reference `&mut self`, which allows us to specify mutations that happen when an iterator returns `None`. This added expressivity in the specification allows us to express properties of *unfused* iterators, which may intermittently produce `None` during iteration. The `produces` transition relation is annotated with *sequences* of generated values rather than with unique values so that a user can reason about interesting properties of *sequences* as a whole rather than directly reasoning about the notion of transitive closure, which the automated solvers don't handle well. The price to pay is the laws of reflexivity and transitivity, which the implementors of iterators have to prove.

6.1.2 Structural Invariant of `for` Loops

Part of the appeal of `for` loops is the *structure* they provide over the looping process. When a programmer sees a `for`, they can conclude that the body will be executed at most once for each element in the iterator. Unlike with `while` loops, it is impossible to decrease the loop index or otherwise perform unpredictable looping patterns. This informal reasoning can be formalized as a loop invariant, provided *structurally* by the `for` loop itself. The iterator at the i-th iteration is the result of calling `next` exactly i times on some initial state. In our formalism, given an initial iterator state `initial` and a current iterator state `iter`, we can state this invariant as $\exists\, p$, `initial` $\overset{p}{\rightsquigarrow}$ `iter`. This invariant holds for *any* `for` loop over *any* iterator: it can be derived from the laws `produces_refl` and `produces_trans`.

When using Creusot, every `for` loop benefits from this structural invariant: we change the way these loops are desugared into the more primitive `loop` construct by adding ghost variables

```
1    let mut count = 0;
2    #[invariant(count_is_n, @count == produced.len())]
3    for i in 0..n { count += 1; assert!(0 <= i && i < n); }
4    assert!(n < 0 || count == n);
5
```

Figure 6.2: A simple for loop using ranges.

`init_iter` and `produced` and the new structural invariant `init_iter.produces(produced, iter)`. More precisely, a simple for loop `for x in iter {<body>}` is desugared into:

```
let init_iter = ghost! { iter };
let mut produced = ghost! { Seq::EMPTY };
#[invariant(structural, init_iter.produces(produced, iter))]
loop { match iter.next() {
  None => break,
  Some(x) => {
    produced = ghost! { produced.concat(Seq::singleton(x)) };
    <body> },
} }
```

Interestingly, the ghost variable `produced` can be referred to in a user invariant to relate the state of the loop with the iteration state.

In the piece of code in Figure 6.2, we use a variable `count` to count the number of elements generated by an iterator and use such an invariant to verify its intended meaning.

6.2 Examples of Specifications of Simple Iterators

In §6.1, we have presented a general framework to specify iterators and use them in `for` loops. This section presents several simple examples of iterators defined in this framework.

6.2.1 The `Range` Iterator

We start with a simple `Range` iterator, whose purpose is to iterate over the integers in a given range. The notation `a..b` used idiomatically in Rust is syntactic sugar for this kind of iterator. The original definition from the Rust standard library is generic over the type of integers used, but for the sake of simplicity, we use a monomorphic version here:

```
struct Range { start: usize, end: usize }
```

If `self.start` $\geq$ `self.end`, the `next` method returns `None`. Otherwise, it increments `self.start` and returns the initial value of `Some(self.start)`. Note that the upper bound of the range, `end`, is *excluded* in the iteration.

In order to instantiate our iterator specification scheme with `Range`, we use the `produces` and `completed` predicates defined by:

$$
\begin{aligned}
r \overset{v}{\rightsquigarrow} r' \quad &\triangleq \quad |v| = r'.\texttt{start} - r.\texttt{start} \wedge r.\texttt{end} = r'.\texttt{end} \\
&\quad \wedge\ |v| > 0 \Rightarrow r'.\texttt{start} \leq r'.\texttt{end} \\
&\quad \wedge\ \forall i \in [0, |v| - 1],\ v[i] = r.\texttt{start} + i \\
\mathit{completed}(r) \quad &\triangleq \quad *r = \hat{}r \wedge (*r).\texttt{end} \geq (*r).\texttt{start}
\end{aligned}
$$

Transitivity and reflexivity are easily verified.

Rust's standard library also contains ranges whose upper bound is included rather than excluded and ranges without an upper bound. They can all be specified using similar techniques.

Note that with these definitions, the structural invariant of `for` loops directly implies that the loop index (the last produced value) is in the range. In addition, if the range is non-empty, one can deduce that the last iterated value is `end` -1. These two properties usually require an additional invariant if the loop is encoded using the `while` construct. For an illustration, consider Figure 6.2.

6.2.2 `IterMut`: Mutating Iteration Over a Vector

Our approach to iterators can be used to iterate over vector elements. But instead of presenting the simple case of a *read-only* vector iterator, we study a more general iterator, `IterMut`, permitting both *read and write* vector elements while iterating; the simpler case of the read-only iterator uses the same ideas.

This iterator produces mutable references for each element of a vector in turn. The state of this iterator is a mutable reference to the *slice* (i.e., a fragment of a vector) of elements that remain to be iterated:

```
1 struct IterMut<'a, T> { inner: &'a mut [T] }
```

To define the production relation of `IterMut`, we use a helper function:

$$tr : \texttt{\&mut Seq<T>} \to \texttt{Seq<\&mut T>}$$

which transposes a mutable reference to a slice into a *sequence* of mutable references to its elements. Its defining property is:

$$|tr(s)| = |*s| \ \wedge \ \forall i \in [0, |*s| - 1],\ tr(*s)[i] = *s[i] \ \wedge \ tr(\hat{}s)[i] = \hat{}s[i]$$

With the help of tr, the `produces` and `completed` relations of `IterMut` are simple to express:

$$\begin{aligned} it \overset{v}{\rightsquigarrow} it' &\triangleq tr(it.\texttt{inner}) = v \cdot tr(it'.\texttt{inner}) \\ completed(it) &\triangleq *r = \hat{}r \wedge |*r| = 0 \end{aligned}$$

It means that the iterator it produces a sequence of mutable references, which must be the initial segment of $tr(it.\texttt{inner})$, into a final state it' such that $tr(it.\texttt{inner})$ is the sequence of mutable references that are left to be generated. Such an iterator is completed when the inner slice is empty.

This compact specification is enough to reason about mutating through the returned pointers, as in the following example:

```
1 #[invariant(all_zero, forall<i: Int> 0 <= i && i < produced.len()
2                                   ==> @^produced[i] == 0)]
3 for x in v.iter_mut() { *x = 0; }
4 proof_assert!{
5   forall<i: Int> 0 <= i && i < (@v).len() ==> @(@^v)[i] == 0
6 }
```

That is, we can prove with a simple loop invariant that this loop sets to 0 all the elements of the vector.

The reasoning that occurs to prove this program is as follows. First, at the end of a loop iteration, we know that the final value of the borrow `x` is equal to 0 since we have just written 0, and this value will not change since `x` goes out of scope. Together with the invariant of the preceding iteration, this is enough to prove that the invariant is maintained. Second, after the loop has been executed, the final iterator state is empty, so we know `produced` contains the complete sequence of borrows to elements of v. However, thanks to the loop invariant, the prophetic value of each of these borrows is 0. So, we can deduce that the final content of v is a sequence of zeros.

6.2.3 Iterator Transformers

Because all iterators implement the same trait `Iterator`, which gives them a specification, we can quickly build combinators that wrap and transform the behavior of an iterator.

It is important to note that, following Rust's standard library, these transformers are generic over the *type* of the underlying iterator; individual values of a type cannot have different predicates. While the verification tool cannot know the concrete definitions of `produces` or `completed` for the wrapped iterator, it knows it must satisfy the `Iterator` trait interface.

The simplest example is `Take<I>` (where `I` is another iterator), which truncates an iterator to produce at *most* n elements. The state of `Take<I>` is a record with two fields: a counter `n` for the remaining elements to take and an iterator `iter` to take from. The specification predicates of `Take<I>` are defined as follows:

$$\begin{aligned} it \overset{v}{\rightsquigarrow} it' &\triangleq it.\texttt{iter} \overset{v}{\rightsquigarrow} it'.\texttt{iter} \wedge it.\texttt{n} = it'.\texttt{n} + |v| \\ \mathit{completed}(it) &\triangleq (*it).\texttt{n} = 0 \wedge *it = \hat{}it \\ &\quad \vee (*it).\texttt{n} > 0 \wedge (*it).\texttt{n} = (\hat{}it).\texttt{n} + 1 \wedge \mathit{completed}(it.\texttt{iter}) \end{aligned}$$

The subtle definition here is $\mathit{completed}(it)$: if the counter is 0, then `next` does nothing. However, following Rust's implementation, if the counter is not 0, it is first decremented even if the call to the underlying iterator returns `None`.

Again, when instantiated to a specific underlying iterator *type*, we can substitute the definitions of $(\rightsquigarrow)$ and $\mathit{completed}(-)$ for the underlying iterator to get a concrete definition of these predicates for `Take<I>`, which are easier to handle by automated solvers.

Another common transformer is `Skip<I>`, whose goal is to *skip* the first n elements of an iterator. Similarly to `Take<I>`, the state is a record with two fields: a number `n` of elements to skip and an underlying iterator `iter`.

The $\rightsquigarrow$ relation of `Skip<I>` is defined as follows:

$$\begin{aligned} it \overset{v}{\rightsquigarrow} it' &\triangleq v = \varepsilon \wedge it = it' \\ &\quad \vee it'.\texttt{n} = 0 \wedge |v| > 0 \wedge \exists\, w,\ |w| = it.\texttt{n} \wedge it.\texttt{iter} \overset{w \cdot v}{\rightsquigarrow} it'.\texttt{iter} \end{aligned}$$

The first disjunct is needed to ensure reflexivity of $(\rightsquigarrow)$. The second disjunct describes what happens after a non-empty sequence of calls. If we produced some sequence of elements v, we must have been able to skip n elements first, which we existentially quantify over.

If the `Skip<I>` iterator is completed, the underlying iterator has also completed, but potentially after having generated some skipped elements that we existentially quantify over:

$$\begin{aligned} \mathit{completed}(it) &\triangleq \exists\, w\ i,\ (\hat{}it).\texttt{n} = 0 \wedge |w| \leq (*it).\texttt{n} \\ &\quad \wedge (*it).\texttt{iter} \overset{w}{\rightsquigarrow} *i \wedge \mathit{completed}(i) \wedge \hat{}i = (\hat{}it).\texttt{iter} \end{aligned}$$

Using `Skip<I>` and `Take<I>`, we can prove an algebraic property of iterators: if we take n elements and skip n elements from that iterator, we must necessarily get the empty iterator.

```
1    proof_assert!(iter.take(n).skip(n).next().is_none())
```

This property is easy to prove from the composition of both production relations.

6.2.4 Fuse

Iterators in Rust have an unusual wrinkle: an iterator that has completed (by returning `None`) can *resume* producing values at a later point. Because this behavior is unintuitive and often undesired,

Rust provides a key iterator, ensuring that an iterator will never return a value after the first `None`. This combinator, called `fuse` by analogy to electric circuits, is an oft-overlooked component to the correctness of iterator code. The type of `Fuse` is defined as `Option<I>`, either there is an active iterator `I` or `None` to indicate iteration has finished. Using this, we can specify the production relation as follows:

$$\begin{aligned} iter \overset{v}{\rightsquigarrow} iter' \quad \triangleq \quad & (\forall a,\, iter = \texttt{Ok}(a) \Rightarrow i \overset{v}{\rightsquigarrow} iter') \\ & \wedge\, (iter = \texttt{None} \Rightarrow v = \varepsilon \wedge iter = iter') \end{aligned}$$

It can produce normal values if the iterator is still in the `Ok` state. When it is in the `Err` state, it can only produce the empty sequence and must remain in the same state, thus fusing.

The $completed(-)$ relation is defined as follows:

$$completed(it) \quad \triangleq \quad it = \texttt{None}$$

The iterator is completed if it is in the `None` state. While unfused iterators are rare in practice and application, this combinator demonstrates the generality of our specification.

6.3 A Higher-order Iterator Combinator: `Map`

The `Map` iterator applies an arbitrary closure of type `F` with *mutable state* to the successive elements of an arbitrary iterator `I`.

```
1   struct Map<I, F> {
2     iter: I,
3     func: F,
4   }
```

The challenge with `Map` is handling the preconditions of the closure being called. The function provided could have arbitrary preconditions; even worse, those preconditions could depend on the closure's captured (mutable) state. As shown in § 3.6, mutable closures can use `precondition` and `postcondition_mut` to refer to the precondition and postconditions of the closure. At all times, `Map` must retain knowledge that the precondition is satisfied and accumulate the output of the postconditions to provide that information to the client of the iterator.

To help work through this, we use a thought experiment where we see `Map` implemented as a loop with a yield instruction to generate elements in the style of e.g., Python generators:

```
1 fn map<I : Iterator, B, F: FnMut(I::Item) -> B>(iter: I, f: F) {
2   for a in iter { yield (f)(a) }
3 }
```

To verify it, we need `f.precondition(a)` to be `true` at each iteration, so we need an invariant that implies it. This exposes the fundamental property that must be true of our closure: the postcondition at iteration n must be able to establish the precondition for iteration $n+1$. In the vocabulary of iterators:

$$it \overset{s \cdot e_1 \cdot e_2}{\rightsquigarrow} i' \to pre(*f, e_1) \to post(f, e_1, r) \to pre(\hat{}f, e_2)$$

expresses that if we eventually produce an element e_1 which satisfies the precondition of the initial closure $*f$, then combined with the postcondition of f, we must be able to establish the precondition for the final closure $\hat{}f$ with the following element e_2. Quantifying over a prefix s in the iteration from a known initial state i ensures this property holds for all subsequent iterations.

To encode this property in `Map`, we use a *type invariant*[2], which allows specifying a property that values of a type must uphold. Here, the invariant states that (1) the precondition for the next call will be verified; (2) the preservation property above holds for the current state it; (3) these two invariants are reestablished if the underlying iterator returns `None` (this is usually trivial since the underlying iterator is often fused: it cannot generate new elements once it returns `None`); and (4) the type invariant of the underlying iterator holds.

These invariants are initially required as a precondition of the `map` method used to create the `Map` iterator. To be tackled by automated solvers, this verification condition needs to be unfolded; therefore, closures and their pre and post-conditions must be statically resolved thanks to Rust's unique anonymous closure types.

The specification predicates for `Map` can now be stated:

$$
\begin{aligned}
it \overset{v}{\rightsquigarrow} it' \quad \triangleq \quad & \exists v'\, fs,\ |v'| = |fs| = |v| \wedge it.\texttt{iter} \overset{v'}{\rightsquigarrow} it'.\texttt{iter} \\
& \wedge (it.\texttt{func} = *fs[0] \wedge \hat{}fs[0] = *fs[1] \wedge .. \wedge \hat{}fs[n] = it'.\texttt{func}) \\
& \wedge \forall i \in [0, |v| - 1],\ pre(*fs[i], v'[i]) \wedge post(fs[i], v'[i], v[i]) \\
& \wedge unnest(it.\texttt{func}, it'.\texttt{func}) \\
completed(it) \quad \triangleq \quad & completed(it.\texttt{iter}) \wedge (*it).\texttt{func} = (\hat{}it).\texttt{func}
\end{aligned}
$$

In $\rightsquigarrow$, we quantify existentially over two pieces of information: the sequence of values v' produced by the underlying iterator and the sequence of *mutable references* of states fs that the closure traverses. We require that fs forms a chain, the final state of each element being the same as the current value of the following one. Finally, we require the closure pre- and post-conditions for every iteration and that the unnesting relation relates to the first and last states. On the other hand, the definition of $completed(-)$ straightforwardly states that the underlying iterator is completed.

Interestingly, the user of this specification can use the precondition of the closure to encode closure invariants that she wishes to maintain along the iteration (as with loop invariants). This specification for `Map` allows us to specify many use cases, so long as the supplied closure is "history-free": its specification does not depend on the sequence of previously generated values, like in `x.map(|a : u32| a + 5)`. While this is undoubtedly the most common usage of `map`, we sometimes need a more robust specification.

Extending `Map` With Ghost Information. If we attempt to use the previous specification of `Map` to verify Example 1, we would rapidly struggle to prove that `cnt` matches the length of the vector. This happens because with `Map` the closure can only specify the behavior of a single call in isolation: all we can state is that `cnt` was incremented once. Deducing that we called the closure the appropriate number of times would require an induction over the iteration. Compared to a `for` loop, `Map` has lost some expressivity, and we have no mechanism like `produced` to specify the behavior of iteration.

To simplify the verification of this kind of code, we extend the signature of `Map` to provide the closure of the sequence of elements generated by the underlying iterator since the creation of the mapping iterator object. This information does not change the behavior of the program: we make it *ghost*, so it can only be used in specifications:

```
1  struct MapExt<I : Iterator, F> {
2    iter: I,
3    produced: Ghost<Seq<I::Item>>,
4    func: F,
```

[2]At the time this work was performed, we simulated type invariants by manually adding the required pre and post-conditions. Today, type invariants are a first-class feature of Creusot.

```
}

fn map_with_past<I, B, F>(i: I, f: F) -> MapExt<I, F>
  where I: Iterator,
        F : FnMut(Ghost<Seq<I::Item>>, I::Item) -> B
  {
    MapExt { iter: i, produced: ghost! { Seq::EMPTY }, func: f }
  }
```

The extended map, `MapExt`, is thus given an additional ghost field containing this sequence, `produced`. The relation $(\rightsquigarrow)$ is extended to account for this ghost information by adding a conjunct stating that $it'.\texttt{produced} = it.\texttt{produced} \cdot v'$ and passing the additional ghost parameter $it.\texttt{produced} \cdot v'[0..i-1]$ to the pre- and post- conditions. The $completed(-)$ relation is extended by adding the conjunct $(\texttt{\^{}}it).\texttt{produced} = \varepsilon$ (the produced field is reset when the iterator returns `None`):

$$
\begin{aligned}
it \overset{v}{\rightsquigarrow} it' \quad \triangleq \quad & \exists v'\, fs,\ |v'| = |fs| = |v| \wedge it.\texttt{iter} \overset{v'}{\rightsquigarrow} it'.\texttt{iter} \\
& \wedge\ it.\texttt{produced} \cdot v' = it'.\texttt{produced} \\
& \wedge\ (it.\texttt{func} = *fs[0] \wedge \texttt{\^{}}fs[0] = *fs[1] \wedge .. \wedge \texttt{\^{}}fs[n] = it'.\texttt{func}) \\
& \wedge\ \forall\, i \in [0, |v| - 1],\ pre(*fs[i], v'[i], it.\texttt{produced} \cdot v'[0..i]) \wedge \\
& \quad post(fs[i], v'[i], it.\texttt{produced} \cdot v'[0..i], v[i]) \\
& \wedge\ unnest(it.\texttt{func}, it'.\texttt{func}) \\
completed(it) \quad \triangleq \quad & completed(it.\texttt{iter}) \wedge (*it).\texttt{func} = (\texttt{\^{}}it).\texttt{func} \wedge (\texttt{\^{}}it).\texttt{produced} = \varepsilon
\end{aligned}
$$

The type invariants are adapted accordingly.

This extra information avoids the need for an explicit induction to establish that we have correctly counted the number of iterations. It suffices to look at the postcondition of the last call to `next`. While this example only uses the length of the sequence, this ghost information is helpful in various situations. Using this extended version of map we can prove Example 1 with the following code:

```
let w: Vec<u32> = vec![1,2,3]
.iter()
.map(
    #[requires(cnt@ == _prod.len() && cnt < usize::MAX)]
    #[ensures(cnt == old(cnt) + 1u32 && cnt@ == prod.len() + 1 && result ==
    *x+1u32)]
    |x, prod| {
        cnt += 1;
        *x+1
    },
)
.collect();
```

This specification explcitly tracks the fact that `cnt` is the same as the length of `prod`, the sequence of elements already produced. After the call to collect, we can thus conclude that `cnt` is the same as the length of the iterated vector.

Iterator	LOC	Spec	Time	Fully auto.
`Range`	13	39	0.40	✓
`IterMut`	12	34	0.61	✓
`Map`	23	46	0.89	✗
`MapExt`	42	115	1.06	✗
`Skip<I>`	20	53	0.51	✗
`Take<I>`	17	43	0.40	✓
`Fuse`	29	51	0.52	✗

Figure 6.3: Evaluation results for the implementations of iterators. "LOC" counts the lines of program code, while "Spec" counts specification code and assertions. "Time" measures in seconds the time taken to solve the proofs. "Fully auto." determines whether manual tactics were used.

6.4 Evaluation

In this section we measure the performance of both the proofs of iterators and their clients, using Creusot [27]. The results in Figure 6.3 and Figure 6.4, were gathered using a Macbook Pro with an M1 Pro CPU and 32 GB of RAM, running macOS 12.2. Why3 was limited to running four prover instances simultaneously, using Z3 4.11.2, CVC5 1.0.2, and Alt-Ergo 2.4.1.

Why3 supports *proof transformations*: manual tactics that can be combined with automated solvers. We minimize their use because we wish to obtain ergonomic specifications that work well with automation. Nevertheless, certain complex proofs required minor manual work, which we indicate.

The table in Figure 6.3 contains a selection of the iterators and combinators we have verified. The `Range`, `IterMut`, `Skip` and `Take` iterators are implementations of the iterators described in §§ 6.2.1 to 6.2.3. The `Fuse` combinator is responsible for transforming any iterator into a *fused* one, which will always return `None` after the first, never resuming iteration. Two versions of `Map` are provided; the first is the standard library `Map`, which is restricted to closures whose preconditions are 'history-free', and the version in `MapExt` is provided with ghost information about previous calls as explained in § 6.3.

Some manual proof steps were required to prove several iterators. For `Skip<I>` and `Fuse`, the manual tactics only tell Why3 to access lemmas about sequences. For `Map`, and `MapExt`, tactics were used to instantiate the existential quantifiers within the production relation.

Correspondence of `IterMut` with RustHornBelt In § 5.3, we presented a *different* specification for `IterMut`. The primary difference occurs in the model of the iterator itself. RustHornBelt represents the iterator as a sequence of borrows, baking in the equivalent to our transpose operation *tr*. Outside of this difference, our specifications are equivalent, but we provide a larger framework within which `IterMut` fits in and can interact with other iterators.

We also verified several clients of iterators, particularly featuring combinations of several iterators. The example `decuple_range` maps a `Range`, multiplying elements by 10, collecting the results into a vector, and verifying functional correctness; `counter` is an annotated version of the example in the introduction, verifying we can use the mutable state to count the elements of an iterator; `concat_vec` uses `extend` to append an iterator to the end of a vector; `all_zero` is uses `IterMut` to

Benchmark	LOC	Spec	Time	Fully auto.	Used Iterators
`all_zero`	5	3	0.43	✓	`IterMut`
`skip_take`	3	2	0.40	✓	`Skip`, `Take`
`counter`	12	4	0.55	✓	`Map`, `Iter`
`concat_vec`	3	3	0.41	✓	`IntoIter`
`decuple_range`	9	3	0.64	✓	`Map`, `Range`
`hillel`	89	109	0.86	✓	`Range`, `Iter`, `IterMut`
`knights_tour`	89	55	1.15	✓	`Range`, `IntoIter`

Figure 6.4: Selected results of iterator *clients*. "LOC" counts the lines of program code, while "Spec" counts specification code and assertions. "Time" measures in seconds the time taken to solve the proofs. "Fully auto." determines whether manual tactics were used.

zero every cell of a vector; `take_skip` checks that if we take n elements and then skip the n next elements of the resulting iterator, we must get `None`. `hillel` is a port of a prior Creusot solution to Hillel Wayne's verification challenges [85]. `knights_tour` is a port of a prior solution to the Knight's Tour problem. In both cases, updating the code to use `for`-loops and iterators actually *reduced* the number of specification lines.

Because our lines of specification include the assertions that test functional properties, we believe the resulting overhead is reasonable, especially in our client examples. Additionally, our specifications for iterators seem to have a low impact on verification times. We compared `hillel` and `knights_tour` with alternative versions that only differ by using traditional `while` loops instead of iterators; verification times are 0.91 and 1.14, respectively. This provides evidence that integrating our iterators does not cause prohibitive increases in verification time.

6.5 Related Works

The formalization of iterators is a well-studied subject with implementations in a variety of imperative and functional languages: WhyML [32], Eiffel [79], Java [72], and OCaml [80]. Of particular relevance is the approach developed by Filliâtre and Pereira [32], which specifies iterators in WhyML using a ghost field `visited : seq 'a` and two predicates `permitted : cursor 'a -> bool` and `completed : cursor 'a -> bool` where `cursor 'a` is an iterator for values of type `'a`. This work leverages Why3's region system to distinguish individual cursors over time. In contrast, in our context, we lose *object identity*: there is no way to identify that two iterator values are two successive states of the same iterator. We thus generalize this approach to our setting by explicitly providing pre- and post-states in `produces`. Our work is also more expressive: we specify and verify higher-order iterators using potentially mutable closures ruled out by Why3's region system. The framework of iteration described by Polikarpova, Tschannen, and Furia [79] is limited to finite, deterministic iteration: the user must provide upfront the sequence of abstract values the iterator will produce. Pottier [80] presents an implementation of iterators for a hash map written in OCaml. They do this by working in the separation logic CFML [21], utilizing Coq's powerful but manual reasoning mechanisms for theorem proving. While Pottier does not provide a general specification of iterators (*cascades*) with mutable state, CFML should permit it, though usage may require a challenging proof.

Chapter 7

Verifying an SMT solver

In parallel to Creusot's development, we have verified programs of increasing complexity as a stress test and objective setting method. In 2022, Høverstad Skotåm used Creusot to verify a competitive Boolean Satisfiability (SAT) solver [46] with performance approaching the fastest unverified solvers. This proof effort motivated the development of many features of Creusot and provided insight into problematic aspects of the verification workflow. The evolution of the CreuSAT work and the last chapter of this thesis is on the verification of Sprout, a solver based on the CDSAT [15] algorithm for Satisfiability Modulo Theory (SMT) solving.

An SMT solver tries to answer the question: Does a model exist for a first-order formula Φ? The ability of modern solvers to answer this question rapidly for a large variety of formulas has led to them becoming essential parts of many key algorithms – including deductive verifiers – and often constitute part of their Trusted Code Base (TCB). Their importance and simple specifications make them prime candidates for verification. However, SMT algorithms have also resisted verification because they lie at the awkward intersection between deep semantic reasoning and highly mutable imperative data structures.

In the rest of this chapter, we will present the implementation and proof of Sprout[1], the first verified SMT solver written in Rust. We begin by introducing the CDSAT algorithm presented by Bonacina, Graham-Lengrand, and Shankar, along with its formalization in Pearlite (§7.1). Then, we present the architecture of Sprout and its verification by refinement (§7.2). Finally, we present an evaluation of Sprout and the verification results in §7.3.

7.1 A mechanized theory of CDSAT

Ever since the introduction of SMT [9], a key challenge has been finding a way to replicate the success that conflict-driven reasoning has had in SAT solving. When a Conflict Driven Clause Learning (CDCL) [68] algorithm gets stuck, it launches an analysis examining the decisions that led to the current state and *explains* them with a *learned clause*, which avoids repeating the same mistakes. Learned clauses prune the search space, and their introduction has led to order-of-magnitude improvements in SAT solving.

In SMT, problems no longer consist solely of propositional formulas but can integrate reasoning about theories like linear rational arithmetic (LRA). Though a conflict-driven procedure for LRA exists, the most common SMT algorithm, DPLL($\mathcal{T}$) [39], treats the theory as a black box and only uses it to check the validity of assignments. Any information derived by a conflict-driven procedure

[1] The codebase can be found at https://github.com/xldenis/cdsat

for LRA remains *internal* and cannot constrain the overall search for a solution. Furthermore, the problems considered often are *combining* multiple theories using techniques like Nelson-Oppen [89] equality sharing.

The problem of combining CDCL with a single conflict-driven theory was solved by MCSAT [48]. In MCSAT, conflict reasoning is lifted to the level of the theory $\mathcal{T}$, no longer occurring purely in boolean terms. However, while further work was conducted on instantiations of MCSAT for specific combinations of theories [13, 49, 41], it remained insufficiently general. CDSAT [15] is the successor to this work, generalizing MCSAT and extending the conflict-driven satisfiability to generic combinations of disjoint theories.

To verify Sprout, we developed a mechanized theory of CDSAT in Pearlite. This mechanization elaborates the key concepts of CDSAT and proves the *soundness* of the abstract CDSAT algorithm presented by Bonacina, Graham-Lengrand, and Shankar. Defining everything in terms of mathematical structures allows the mechanization to focus on the semantic aspects of the algorithm; it does not specify data structures or even a deterministic process for execution.

7.1.1 First-order Theories & Modules

> In this section we summarize the concepts and background of CDSAT, for more details we encourage readers to refer to the original article: Bonacina, Graham-Lengrand, and Shankar [15].

The *signature* Σ is a pair (S, F) of *sorts* S and symbols F. The elements of F include an equality $\cong_s$ for every $s \in S$ and consist of n-ary S sorted functions. We call $\mathcal{V} = (\mathcal{V}^s)_{s\in S}$ as a collection of *variables* where $\mathcal{V}^s$ is the set of variables of sort s. Elements of $\mathcal{V}^s$ (for all $s \in S$) are called $\Sigma[\mathcal{V}]$-*terms* of sort s, and forall $f \in F : (s_1 \times .. \times s_n) \to s$ and $t_1 : s_1, ..t_n : s_n$ we have $f(t_1, ..t_n)$ is a $\Sigma[\mathcal{V}]$-term of sort s. We can obtain the formulas of multi-sorted first-logic by taking the closure of the set of $\Sigma[\mathcal{V}]$-terms under Boolean connectives and quantifiers. We call Σ-*sentences* the set of boolean $\Sigma[\mathcal{V}]$-terms with no free variables.

Definition 8 (Theory)**.** *A theory $\mathcal{T}$ is a pair $(\Sigma, \mathcal{A})$ of a signature and a set of Σ-sentences $\mathcal{A}$ called* axioms*. Given a set of disjoint theories $\mathcal{T}_1, ..\mathcal{T}_n$, the theory $\mathcal{T}_\infty$ is the union of the component theories.*

Example 2 (Theory of Linear Rational Arithmetic)**.** *The theory $(\Sigma, \mathcal{A})$ of LRA has the following signature $\Sigma = (\mathbb{Q}, \{0, 1, +, <\})$. As our axioms $\mathcal{A}$, we take the standard model of linear arithmetic.*

CDSAT solves $\mathcal{T}_\infty$-satisfiability problems seen as sets of assignments to terms. The assignments in a problem may be of any given sort and the assignable values may not be in the theory's signature. We thus *extend* the signature of theories to distinguish the set of assignable values for any given theory.

Definition 9 (Conservative theory extensions)**.** *Let $\mathcal{T} = (\Sigma, \mathcal{A})$ be a theory with $\Sigma = (S, F)$ then $\mathcal{T}^+ = (\Sigma^+, \mathcal{A}^+)$ is an* extension *if $\Sigma^+ = (S, F \uplus F^+)$ and $\mathcal{A}^+ = \mathcal{A} \uplus \mathcal{C}$, where F^+ is a set of constant symbols and $\mathcal{C}$ is a set of Σ^+-sentences. We call F^+ the set of $\mathcal{T}$-values. An extension $\mathcal{T}^+$ is conservative if every $\mathcal{T}^+$-unsatisfiable set is a $\mathcal{T}$-unsatisfiable.*

A sort $s \in S$ is called $\mathcal{T}$-*public* if there are $\mathcal{T}$-values of sort s.

The semantics of a theory $\mathcal{T}$ are characterized by their *models*, which interpret the sentences of $\mathcal{T}$ in some mathematical domain. A $\Sigma[\mathcal{V}]$-*interpretation* $\mathcal{M}$ interprets each sort s in S as a non-empty domain $s^{\mathcal{M}}$, with the Boolean sort interpreted in the usual manner, each variable in $\mathcal{V}$ as an element of the appropriate sort, each symbol f in F as a function in the appropriate domains

and $\cong_s$ as the equality relation in the appropriate domain. A $\mathcal{T}[\mathcal{V}]$-*model* is a $\Sigma[\mathcal{V}]$-interpretation where ignoring the interpretations of variables yields a $\mathcal{T}$-model.

A $\mathcal{T}$-*assignment* $u \leftarrow \mathfrak{v}$ is a pair of a $\mathcal{T}_\infty$-term u and a $\mathcal{T}$-value $\mathfrak{v}$. We use $\top$ and $\bot$ as symbols for true and false respectively. We abbreviate an assignment $u \leftarrow \top$ as u for convenience. Similarly, we abbreviate $u \leftarrow \bot$ as $\overline{u}$. An assignment of non-Boolean sort is called *first-order*. By convention, we use L to refer to Boolean-sorted assignments and A to refer to either Boolean or first-order assignments.

Theory Modules

Theory modules are abstractions of decision procedures for individual theories. A theory module $\mathcal{I}$ is an inference systems composed of $\mathcal{I}$-inferences $J \vdash L$ where J is a set of $\mathcal{T}$-assignments and L is a Boolean assignment. Theory modules are not required to understand arbitrary $\mathcal{T}_\infty$-assignments. Instead, they must only understand the portions relevant to their theory. A *theory view* is the lens through which a module views an arbitrary $\mathcal{T}_\infty$-assignment.

Definition 10 (Theory view)**.** *A* $\mathcal{T}$-view *of a set of* $\mathcal{T}_\infty$*-assignments* A *is the set of* $\mathcal{T}$*-assignments* $A_\mathcal{T}$ *given by the union of the following:*

1. $\{u \leftarrow \mathfrak{v} \mid u \leftarrow \mathfrak{v}$ *is a* $\mathcal{T}$*-assignment in* $A\}$
2. $\bigcup_{k=1}^{n}\{u_1 \cong_s u_2 \mid u_1 \leftarrow \mathfrak{v}, u_2 \leftarrow \mathfrak{v}$ *are* $\mathcal{T}_k$*-assignments in* A *of the same sort*$\}$
3. $\bigcup_{k=1}^{n}\{u_1 \ncong_s u_2 \mid u_1 \leftarrow \mathfrak{v}, u_2 \leftarrow \mathfrak{w}, \mathfrak{v} \neq \mathfrak{w}$ *are* $\mathcal{T}_k$*-assignments in* A *of the same sort*$\}$

An inference $J \vdash L$ is *sound* if all $\mathcal{T}^+[\mathcal{V}]$-models which endorse J also endorse L, and by extension a theory module is sound if all its inferences are.

The Bool **theory module** Propositional reasoning is not built-in to CDSAT, though the Boolean sort is. The Bool theory module provides the rules of propositional logic we expect.

It introduces the usual boolean connectives $\wedge, \vee, \neg$ and, at a minimum, must provide an evaluation rule for formulas composed of boolean sub-formulas. In practice, it also includes additional rules for negation, conjunction, and unit propagation

$$\neg u \vdash \overline{u} \qquad \overline{u_1 \vee \ldots \vee u_n} \vdash \overline{u_i} \qquad u_1 \vee \ldots \vee u_n, \{\, \overline{u_j} \mid j \neq i \,\} \vdash u_i$$
$$\overline{\neg u} \vdash u \qquad u_1 \wedge \ldots \wedge u_n \vdash u_i \qquad \overline{u_1 \wedge \ldots \wedge u_n}, \{\, u_j \mid j \neq i \,\} \vdash \overline{u_i}$$

The LRA **theory module** Linear rational arithmetic (LRA) considers formulas composed of linear (in)equalities. It introduces the sort $\mathbb{Q}$ of rational numbers, along with the symbols $+, -, <$; we also generally admit *scalar multiplication* and $\leq$ as convenient shorthands.

Like Bool, it provides an evaluation rule that determines the value of a boolean LRA formula u from the values of its sub-formulas. However, the completeness of LRA requires more than just evaluation and includes several bookkeeping rules:

$$\overline{u_1 < u_2} \vdash u_2 \leq u_1 \qquad \overline{u_1 \leq u_2} \vdash u_2 < u_1 \qquad u_1 \cong_\mathbb{Q} u_2 \vdash u_1 \leq u_2, u_2 \leq u_1$$

$$u_1 \leq x, x \leq u_2, u_1 \cong_\mathbb{Q} u_0, u_2 \cong_\mathbb{Q} u_0, x \ncong_\mathbb{Q} u_0 \vdash \bot$$

The most important rule of LRA is the *Fourier-Motzkin Elimination* rule, which removes eliminates a variable from a pair of inequalities:

$$u_1 \lessdot_1 x, x \lessdot_2 u_2 \vdash u_1 \lessdot_3 u_2$$

where $\lessdot_1, \lessdot_2, \lessdot_3 \in \{<, \leq\}$ and $\lessdot_3$ is $<$ if and only if either of $\lessdot_1$ or $\lessdot_2$ is $<$.

7.1.2 The CDSAT Trail

The input to CDSAT is a set of $\mathcal{T}_\infty$-assignments like the example below:

$$\{x < 10, a \wedge b, b \Rightarrow x > 15\} \tag{7.1}$$

It then applies a series of deductions to produce a model or, like the example above, the non-existence of one. Like other SAT and SMT solvers, the CDSAT algorithm maintains a *trail* of assignments, which it uses to reason about the current state of the search.

Definition 11 (Trail)**.** *A* trail Γ *is a sequence of $\mathcal{T}_\infty$-assignments and a* reason *for addition, written either as $A_?$ for a* decision*, $A_{\vdash J}$ for a* justified *assignment with justification J, or an* input assignment $A_\emptyset$.

In the trail, decisions are arbitrary choices made when the search got stuck, while *justified* assignments are logical deductions made by a dedicated *theory solver*.

We formalize the notion of the trail using a *list* type in Creusot.

```
1   pub enum Trail {
2       Empty,
3       Assign(Assign, Int, Box<Trail>),
4   }
5
6   pub enum Assign {
7     Decision(Term, Value),
8     Justified(FSet<(Term, Value)>, Term, Value),
9     Input(Term, Value),
10  }
```

The type `FSet` is a Creusot type for *finite sets*, while `Term` is a type for $\mathcal{T}_\infty$-terms and `Value` is a type for $\mathcal{T}_\infty$-values. Each entry in the trail also stores its *level* as an `Int`, representing how many decisions an assignment depends on.

Definition 12 (Level)**.** *Given an assignment $A_i \in \Gamma$, the* level *of A is:*

1. $\text{level}_\Gamma(A_i) = 0$ *if A is an input assignment*
2. $\text{level}_\Gamma(A_i) = \text{level}_\Gamma(H)$ *if A_i is a justified assignment with justification H where the level of H is the maximum of the levels of its elements.*
3. $\text{level}_\Gamma(A_i) = 1 + \max\{\text{level}_\Gamma(A_j) \mid j < i\}$ *if A_i is a decision*

Unlike the notion of the level found in SAT or DPLL($\mathcal{T}$) solvers, the level is not *monotonic* over a CDSAT trail. Due to *late propagations*, a justified assignment may be added later during the search that does not depend on high-level values.

The *restriction* of a trail $\Gamma^{\leq n} = \{A \mid \text{level}_\Gamma(A) \leq n\}$ is used to perform *backtracking* by forgetting elements of a level greater than n, and is similarly non-monotonic.

The trail's level and restriction are formalized as methods on our `Trail` type.

```
impl Trail {
  #[logic] pub fn level(self) -> Int { ... }
  #[logic] pub fn restrict(self, level: Int) -> Trail { ... }
}
```

As deductions and calculations are made, the trail should remain *plausible*, it should not contain any immediate contradictions. To enforce this, CDSAT introduces the notion of *acceptability*, which constrains theories from making arbitrary deductions.

Theory modules are responsible for the 'reasoning' of CDSAT; given their $\mathcal{T}$-view understanding of Γ, they must decide which $u \leftarrow \mathfrak{v}$ should be added to the trail. Such assignments should be *relevant* to $\mathcal{T}$, either consisting of $\mathcal{T}$ public sorts or equality (shared between all theories). Moreover, proposed assignments should be *acceptable*, they should maintain plausibility, if $u \leftarrow \text{true} \in \Gamma$ then we should not propose $u \leftarrow \text{false}$. Finally, a theory module should not propose assignments that lead to self-contradiction; if a module proposes L, then there should be no inference such that $\Gamma \vdash \overline{L}$.

Definition 13 (Relevance)**.** *A term u is relevant to $\mathcal{T}$ in Γ if (i) u has a $\mathcal{T}$-public sort and occurs in Γ, or (ii) u is an equality $u_1 \cong_s u_2$ where u_1, u_2 occur in Γ and s is not $\mathcal{T}$-public.*

Definition 14 (Acceptability)**.** *An assignment $u \leftarrow \mathfrak{v}$ is* acceptable *for a trail Γ when:*

1. *u is* relevant *to $\mathcal{T} \in \mathcal{T}_\infty$*
2. *Γ does not assign a value to u*
3. *$u \leftarrow \mathfrak{v}$ is first-order, there are no $\mathcal{I}_\mathcal{T}$-inferences $J \cup \{u \leftarrow \mathfrak{v}\} \vdash \overline{L}$ with $J \subseteq \Gamma$ and $\overline{L} \in \Gamma$*

Acceptability is formalized as a predicate on the trail and used as a precondition for adding an assignment to the trail. Additionally, we require that added deductions are *sound*.

```
impl Trail {
  #[predicate] pub fn acceptable(self, assign: Assign) -> bool;

  #[requires(self.acceptable((term, value)))]
  #[logic] pub fn add_decision(self, term: Term, value: Value) -> Trail;

  #[requires(self.acceptable((term, value)))]
  #[requires(Assign::justified(justification, term, value).sound())]
  #[logic] pub fn add_justified(self,
    term: Term, value: Value, justification: FSet<(Term, Value)>) -> Trail;
}
```

Finally, the trail offers membership testing and accessors:

```
impl Trail {
  #[predicate] pub fn contains(self, assign: (Term, Value)) -> bool;

  #[requires(self.contains(assign) && self.is_justified(assign))]
  #[logic] pub fn justification(self, assign: (Term, Value))
     -> FSet<(Term, Value)>;
}
```

7.1.3 The CDSAT Algorithm

As presented by Bonacina, Graham-Lengrand, and Shankar, CDSAT is a non-deterministic transition system over a trail Γ. The eight rules of CDSAT are spread in two groups. The *normal* rules

$$\textsc{Decide}\quad \frac{A \text{ is an acceptable } \mathcal{T}_k\text{-assignment in } \Gamma}{\Gamma \longrightarrow \Gamma, A_?} \qquad \textsc{Deduce}\quad \frac{J \vdash L \quad J \subseteq \Gamma \quad L \notin \Gamma \quad \overline{L} \notin \Gamma}{\Gamma \longrightarrow \Gamma, L_{\vdash J}}$$

$$\textsc{Fail}\quad \frac{J \subseteq \Gamma \quad L \notin \Gamma \quad \overline{L} \in \Gamma \quad J \vdash L \quad \text{level}_\Gamma(J \cup \{\overline{L}\}) = 0}{\Gamma \longrightarrow \text{unsat}} \qquad \textsc{ConflictSolve}\quad \frac{J \vdash L \quad J \subseteq \Gamma \quad L \notin \Gamma \quad \overline{L} \in \Gamma \quad \text{level}_\Gamma(J \cup \{\overline{L}\}) > 0 \quad \langle \Gamma; J \cup \{\overline{L}\}\rangle \Longrightarrow^* \Gamma'}{\Gamma \longrightarrow \Gamma'}$$

$$\textsc{UndoClear}\quad \frac{A \text{ is a first-order decision} \quad m = \text{level}_\Gamma(A) > \text{level}_\Gamma(E)}{\langle \Gamma; E \uplus \{A\}\rangle \Longrightarrow \Gamma^{\leq m-1}} \qquad \textsc{Resolve}\quad \frac{A_{H\vdash} \in \Gamma \quad \nexists\, A' \in H,\, A' \text{ first order} \quad \text{level}_\Gamma(A') = \text{level}_\Gamma(E \uplus \{A\})}{\langle \Gamma; E \uplus \{A\}\rangle \Longrightarrow \langle \Gamma; E \cup H\rangle}$$

$$\textsc{Backjump}\quad \frac{\text{level}_\Gamma(L) > m = \text{level}_\Gamma(E)}{\langle \Gamma; E \uplus \{L\}\rangle \Longrightarrow \Gamma^{\leq m}, \overline{L}_{E\vdash}} \qquad \textsc{UndoDecide}\quad \frac{L_{H\vdash} \in \Gamma \quad A \in H \quad A \text{ is a first-order decision} \quad \text{level}_\Gamma(E) = \text{level}_\Gamma(L) = \text{level}_\Gamma(A) = m}{\langle \Gamma; E \uplus \{L\}\rangle \Longrightarrow \Gamma^{\leq m-1}, \overline{L}_?}$$

Figure 7.1: The CDSAT transition system

$\Gamma \to \Gamma'$, and the *conflict* rules $\langle \Gamma; E\rangle \Rightarrow \Gamma$ where E is a set of assignments called a *conflict-set*. Both are listed in Figure 7.1.

The Decide rule is used when modules can no longer make inferences and no contradiction has been found. It adds an arbitrary acceptable decision A to the trail. When a theory module $\mathcal{I}$ finds an acceptable inference $J \vdash_{\mathcal{I}} L$ for $J \subseteq \Gamma$, one of the three remaining rules applies:

1. If $\overline{L} \notin \Gamma$, then we can perform a Deduce and continue with reasoning. In the original paper [15] this rule also has a side-condition that the deduced term is part of the 'finite basis' of the theory. This serves only to establish the termination of the algorithm, a property we are not interested in proving in this work.
2. If $\overline{L} \in \Gamma$ and $\text{level}_\Gamma(J \cup \{\overline{L}\}) = 0$, then we have found a contradiction in the input clauses and can Fail.
3. If $\overline{L} \in \Gamma$ and $\text{level}_\Gamma(J \cup \{\overline{L}\}) > 0$, then we have found a conflict in the deductions and decisions we have made, which we can explain through ConflictSolve.

When a ConflictSolve is applied, CDSAT enters *conflict resolution*, which will use *resolution* to generalize and explain the conflict. The Resolve rule is the primary *explanatory* mechanism, replacing an element of the conflict with its logical antecedents. This process deconstructs the conflict, generalizing it and thus learning a more robust explanation.

Once the conflict has been simplified, we can *explain* it through one of the remaining three rules. These rules backtrack the trail to a state before the conflict occurs and ensure the same mistake will not be repeated. The simplest of these rules is Backjump, which can be applied when the conflict contains a Boolean decision L of maximal level. Because the decision is Boolean we can negate it, and thus we learn that the remainder of the conflict *must* imply $\overline{L}$.

If the conflict instead contained a maximal *first-order* decision $A = u \leftarrow \mathfrak{v}$, negation is impossible. It could be possible to learn that the rest of the conflict implies $u \neq \mathfrak{v}$, but this is

counterproductive when working with infinite sorts; in the case of the rational numbers, we would eliminate precisely *one* rational number, hardly a helpful deduction. Instead, UndoClear backtracks the trail to a state before the decision A was made, but after the rest of the conflict, E was introduced. Despite not learning a lemma, we avoid falling into a loop because A is no longer *acceptable* in the resulting trail.

The final conflict resolution rule UndoDecide occurs when the maximal decision A is contained in the justification of another assignment. In this circumstance, the trail is backtracked, and the assignment is negated, but the remainder of the conflict is not learned.

An Example To better understand how CDSAT functions, let us consider how it would solve Problem (7.1):

$$\{x < 10, a \wedge b, b \Rightarrow x > 15\}$$

When CDSAT starts, the LRA theory cannot progress: the only relevant assignment is $x < 10$, but there is no way to deduce anything from it. Instead, the Boolean theory eliminates the conjunction $a \wedge b$, deducing (Deduce) b with justification $a \wedge b$.

$$\{x < 10, a \wedge b, b \Rightarrow x > 15, b\}$$

Then the Bool theory makes further a deduction, learning $x > 15$ justified by $b \Rightarrow x > 15$ and b.

$$\{x < 10, a \wedge b, b \Rightarrow x > 15, b, x > 15\}$$

The LRA now notices a problem, signaling a conflict that $x < 10$ and $x > 15$ can't both be true. Because $\text{level}_\Gamma(x > 15) = 0$, we conclude that the whole formula is unsatisfiable.

Formalizing the CDSAT algorithm

Our formalization of CDSAT represents each rule as a function from a state to its successor. The various side-conditions of each rule are represented as preconditions on the function. For example, the Decide rule is formalized as follows:

```
#[requires(inp.acceptable(tv))]
#[logic] fn decide(inp: Trail, tv : (Term, Value)) -> Trail {
  inp.add_decision(tv.0, tv.1)
}
```

The conflict rules operate on a state `struct Conflict(FSet<(Term, Value)>, Trail)`, which represents a conflict set and the trail at the time of the conflict. The Resolve rule is formalized as follows:

```
#[requires(inp.0.contains(tv))]
#[requires(inp.1.is_justified(tv))]
#[logic] fn resolve(inp : Conflict, tv : (Term, Value)) -> Conflict {
  let justification = inp.1.justification(tv);
  let new_conflict = inp.0.remove(tv).union(justification);
  Conflict(new_conflict, trail)
}
```

When we later prove the soundness of CDSAT, we will individually prove that each function maintains soundness by adding the relevant pre and postconditions.

7.1.4 A proof of soundness

The original CDSAT algorithm has been proven sound, complete, and terminating by the original authors [15]. In this work, we only consider soundness; the remaining properties rely on complex arguments about theory modules that we have not mechanized. Soundness establishes that if a CDSAT responds with unsat, then there exists no model for the input trail Γ, that is $\nexists \mathcal{M},\ \mathcal{M} \models \Gamma$.

Definition 15 (Ground Entailment)**.** *A* ground entailment $\Gamma \Rightarrow^{\leq 0} \Gamma'$ *is a semantic entailment between two trails restricted at level 0.*

$$\Gamma \Rightarrow^{\leq 0} \Gamma' \triangleq \forall \mathcal{M},\ \mathcal{M} \models \Gamma^{\leq 0} \rightarrow \mathcal{M} \models \Gamma'^{\leq 0}$$

It is easy to see that if a given trail Γ is unsatisfiable, then any trail Γ' which entails it must also be unsatisfiable, as any model of Γ' would also be a model of Γ. Ground entailment is transitive, and thus, by establishing that each rule of CDSAT maintains this property between its input and output states, we can establish that when CDSAT returns unsat, the input trail is unsatisfiable.

We achieve this by instrumenting each of the CDSAT transitions with `#[ensures(inp.entails(result))]`. The exceptions are the Resolve and Fail rules. Both rules leave the trail unchanged; instead, Resolve demonstrates that the conflict set remains conflictual, and Fail demonstrates that the produced conflict is sufficient to conclude unsatisfiability.

Theorem 7.1.1 (Soundness of CDSAT)**.** *Given* $\Gamma \longrightarrow^{*}$ unsat*, then* Γ *is unsatisfiable.*

Proof. The proof proceeds by induction on the transitions of CDSAT. □

7.2 A verified implementation of CDSAT

What we have shown until now remains a mechanization of the *theory* of CDSAT; it does not provide us with an executable implementation to solve SMT problems. Sprout is a verified implementation of CDSAT using the mechanization developed in § 7.1. It instantiates CDSAT with two theories: Bool and LRA and provides a deterministic algorithm to implement the rules of CDSAT. The architecture of Sprout is intentionally kept unsophisticated to ease verification of the code. For the same reason, no optimizations were made in implementing theory solvers, even though they are critical to the performance of a solver. As explained earlier, the proof of Sprout proceeds by *refinement*; at each step, it maintains a correspondence between the data in the *concrete* trail and the *abstract* definition provided in § 7.1.2.

7.2.1 The concrete trail

The Sprout trail is implemented as a vector of vectors of assignments, with the outer vector representing the trail levels and the inner vector representing the assignments at that level.

```
1   struct Trail { assignments: Vec<Vec<Assign>>, ghost: theory::Trail }
```

The trail also stores an abstract trail as a *ghost field*, which will not exist at runtime but can be used in specifications. Indices into the trail (`TrailIndex`) are represented as pairs of a level and an index into the inner vector.

```
1   struct TrailIndex(usize, usize);
```

Because the only way to create a new level is by making a decision, the lexicographic ordering of `TrailIndex` ensures that a decision is always the smallest element at a given level. We rely on this property during the implementation of conflict resolution. On this concrete data structure, we can

define a membership test $A \in \Gamma$ verifying that an assignment is contained within the nested vector `assignments`.

This ghost field allows the concrete trail to be kept in relation to the abstract trail through a predicate:

$$\begin{aligned}\text{abs-conc}(\texttt{self}) \triangleq\ & (\forall\, A \in \texttt{self.ghost},\ A \in \texttt{self}) \wedge \\ & (\forall\, A \in \texttt{self},\ A \in \texttt{self.ghost} \rightarrow \text{level}_{\texttt{self}}(A) = \text{level}_{\texttt{self.ghost}}(A)) \\ & (\forall\, A \in \texttt{self},\ A \in \texttt{self.ghost} \rightarrow \text{reason}_{\texttt{self}}(A) = \text{reason}_{\texttt{self.ghost}}(A))\end{aligned}$$

This property ensures that the elements of the concrete trail can be found in the abstract trail with the same level and reason, and vice-versa. Maintaining this property allows us to transport properties from the abstract trail to the concrete trail. This kind of predicate is often known as a *type invariant*, a property which all valid instances of a type must maintain. Though Creusot has recently gained support for this functionality, the work on Sprout predates it. Instead, we simulate invariants by adding pre- and post-conditions to all functions that modify the trail.

The complete invariant for the trail includes abs-conc along with several other properties, most notably that the first element of each level is a decision:

$$\begin{aligned}\text{invariant}(\texttt{self}) \triangleq\ & \text{abs-conc}(\texttt{self}) \wedge \\ & (\forall\, V \in \texttt{self.assignments},\ V(0) \text{ is a decision}) \wedge \\ & (\forall\, V \in \texttt{self.assignments},\ V \neq \emptyset)\end{aligned}$$

The concrete trail offers the same API as the abstract trail: membership tests, restriction, and insertion of decisions and justified assignments.

These last two functions correspond to the implementation of the Decide and Deduce transitions, respectively and their specifications are shown in Figure 7.2. The `add_decision` only requires that the provided term and value are acceptable in the current trail, and produces an output trail which is ground entailed by the input one.

The `add_justified` specification is necessarily more complicated. It requires that the justification – provided as a vector of trail indices – is a justification for the proposed assignment. This uses a helper function `to_abstract_assign` which converts a vector of trail indices into a set of abstract assignments.

7.2.2 The concrete algorithm

As defined in Figure 7.1, the transitions of CDSAT are non-deterministic; nothing specifies *when* a decision or deduction must be made. Sprout chooses to make maximal deductions and only perform decisions when stuck. The core loop of Sprout is described in Algorithm 1.

The algorithm keeps track of two pieces of state, which theories are *saturated* (have made maximal deductions) and a potential *next decision*. At each iteration, it chooses an unsaturated theory (line 5). If no such theory exists, all theories are saturated (*i.e.,* have no deductions to make), meaning either we must make a choice (line 10) or are satisfied (line 8).

Otherwise, we extend the trail with the chosen theory (line 14). The Extend function is the only interface with theory solvers in Sprout; it is provided by the `Theory` trait with the following signature and contract:

Algorithm 1 CDSAT core loop

```
1:  function Solve(Γ)
2:      Saturated ← {(T, ⊥) | T ∈ T_∞}
3:      NextDecision ← None
4:      loop
5:          T ← T such that Saturated(T) = ⊥
6:          if T = None then
7:              if NextDecision = None then
8:                  return Sat
9:              else
10:                 Γ ← Decide(Γ, NextDecision)
11:                 NextDecision ← None
12:                 Saturated ← {(T, ⊥) | T ∈ T_∞}
13:                 continue
14:         r ← Extend(T, Γ)
15:         if Γ changed then
16:             NextDecision ← None
17:             Saturated ← {(T, ⊥) | T ∈ T_∞}
18:         if r is conflict then
19:             if Level(r) = 0 then
20:                 return unsat
21:             else
22:                 Γ ← ResolveConflct(Γ, r)
23:                 NextDecision ← None
24:                 Saturated ← {(T, ⊥) | T ∈ T_∞}
25:                 continue
26:         if r is a decision then
27:             NextDecision ← r
28:         Saturated(T) ← ⊤
```

```
impl Trail {
  #[requires(self.invariant())]
  #[ensures((^self).invariant())]
  #[requires(self.ghost.acceptable(term, val))]
  #[ensures(self.ghost.entails((^self).ghost))]
  pub(crate) fn add_decision(&mut self, term: Term, val: Value) { .. }

  #[requires(self.invariant())]
  #[ensures((^self).invariant())]
  #[requires(self.contains(just))]
  #[requires(
    Assign::Justified(self.to_abstract_assign(just), term@, val@)
  )]
  #[requires(self.ghost.acceptable(term, val))]
  #[ensures(self.ghost.entails((^self).ghost))]
    pub(crate) fn add_justified(&mut self,
      just: Vec<TrailIndex>, term: Term, val: Value) { .. }
}
```

Figure 7.2: The signatures and contracts of `add_decision` and `add_justified`.

```
enum ExtendResult {
  Conflict(Vec<TrailIndex>),
  Decision(Term, Value),
  Satisfied,
}
trait Theory {
  #[ensures(match result {
      ExtendResult::Satisfied => true,
      ExtendResult::Decision(t, v) => (^trail).acceptable(t, v),
      ExtendResult::Conflict(c) => {
        !c.is_empty() && (^trail).contains(c)
        && Model::unsatisfiable(tl.to_abstract_assign(c))
      }
  })]
  #[ensures(trail.ghost.entails((^trail).ghost))]
  fn extend(&mut self, tl: &mut Trail) -> ExtendResult;
}
```

This makes use of a helper function `to_abstract_assign` which converts the vector of trail indices into a set of abstract assignments.

The theory solver may add justified assignments by directly manipulating the trail, and must return one of three statuses. A `ExtendResult::Satisfied` indicates the trail is satisfactory for this specific theory; when all theories simultaneously return `Satisfied`, the solver can return `Sat` as an overall answer. The correctness of this result is established by showing *completeness*, and thus, we do not specify anything for it.

A result of `ExtendResult::Decision(t, v)` occurs when the theory has made the maximal amount of deductions possible but still cannot determine the trail to be satisfactory. It thus proposes a potential decision that would unblock further reasoning. For this decision to be used, the result

must be *acceptable*, or it would be impossible for the main loop to be used.

Finally, a `ExtendResult::Conflict(c)` is returned when a theory has identified a set of conflicting clauses. In this case, the theory must demonstrate the conflict's validity: the values in the trail are jointly unsatisfiable.

Suppose the theory solver made deductions during the call to Extend. Other theories may have their interpretations of the trail affected, so we reset the saturated state and the next decision. For example, given the trail $\{x < 15, b \leftarrow \top, b \Rightarrow x > 15\}$, the Bool could decide to deduce $x > 15$, which would make the trail $\{x < 15, b \leftarrow \top, b \Rightarrow x > 15, x > 15\}$, that LRA would conclude is unsatisfiable.

After extending the trail, we check if the result is a conflict (line 18). If so, we check if the conflict is at level 0 (line 20), concluding that we have found a contradiction in the input clauses. Otherwise, we resolve the conflict (line 22) and reset the saturated state and the next decision.

Finally, if the result is a decision, we store it as a candidate decision (line 26), and mark $\mathcal{T}$ as saturated.

Proving the soundness of this algorithm is straightforward; it is a direct consequence of the soundness of the individual theories. The two difficulties in this code lie in the proof of conflict resolution (ResolveConflict) and ensuring that all decisions are *acceptable*. To preserve the acceptability of decisions, we aggressively clear out candidates if the trail is changed in any way, even if the change is not related to the candidate decision.

Conflict Resolution

Most proof and implementation effort went into the design and proof of the conflict resolution procedure, resulting in Algorithm 2. Due to this procedure's implementation, verifying the soundness of conflict resolution first requires establishing the *completeness* of the same process. Each iteration of the loop attempts to apply one of the conflict resolution rules; if none is applied, the loop would enter an undetermined state, which would not be proven sound. Proving conflict resolution required clarifying several ambiguities in the original definitions and multiple auxiliary lemmas to materialize that understanding. The final algorithm is described in Algorithm 2.

Algorithm 2 Conflict Resolution procedure for Sprout

```
1:  function ResolveConflict(Γ, C)
2:      while C ≠ ∅ do
3:          A ← PopMax(C)
4:          if A is boolean sort and Level(A) > Level(C) then
5:              return Γ^{≤Level(C)} ∪ Ā_{⊢C}
6:          if A is a first-order decision then
7:              return Γ^{≤Level(C)−1}
8:          for B ∈ Justification(A) do
9:              if B is a first-order decision and Level(B) = Level(C) then
10:                 return Γ^{≤Level(C)−1} ∪ Ā_?
11:
12:         C ← C ∪ Justification(A)
```

At each iteration of conflict resolution, the algorithm attempts to apply each of the four conflict resolution rules. Ensuring that the assignment being considered at each iteration can always be applied to one of the rules was a primary source of difficulty in the proof. The key lies in the PopMax function, which returns the most significant element according to the lexicographic order

on `TrailIndex`. This order ensures that we always consider deductions before making decisions. Failure to do so could have meant selecting a Boolean decision, which does not have a greater level than the rest of the conflict. This would prevent applying Backjump, and none of the remaining rules could be applied as they only concern first-order and justified assignments.

The second subtlety occurs at line 7. A not immediately apparent property is that the assignment under consideration must be justified at this point. The first test on line 4 eliminates any Boolean decision, as we could only consider one if it was the last element of the maximal level. Similarly, the test on line 6 eliminates any first-order decision. The only possibility is that the assignment is justified; thus, we can safely iterate over its justification. Finally, the side conditions of UndoDecide and Resolve are complementary, and thus we can safely apply Resolve if UndoDecide does not apply.

Concrete theory implementations

The final component to the soundness of Sprout are the implementations of the Bool and LRA theories. As mentioned earlier, these implementations are intentionally kept simple to ease verification. In each case, we must ensure that any proposed deductions are sound and that returned decisions and conflicts are valid.

The Boolean theory As stated in the original CDSAT paper [15], only the evaluation rule is necessary for completeness, and the unit propagation rules are only used to improve performance.

Thus our naive implementation of the Boolean theory is a simple interpreter, provided by the function

```
1   fn eval(&mut self, tl: &Trail, tm: &Term) -> (Vec<TrailIndex>, Result<Value,
      Term>);
```

which evaluates each boolean clause in the trail and either returns a value or a subterm of unknown value. Along with this result, the evaluator returns the set of trail indices that were used to evaluate the term `tm`. This value is compared to the expected value for each assignment, if there is a mismatch this indicates a conflict, justified by the vector of trail indices provided by the evaluator.

The specification of this evaluator is:

```
1   #[requires(tl.invariant())]
2   #[requires(tm@.well_sorted())]
3   #[requires(tm@.is_bool())]
4   #[ensures(forall<ix : _> result.0@.contains(ix) ==> tl.contains(ix))]
5   #[ensures(match result.1 {
6       Ok(v) => { v@.is_bool()  &&
7         Assign::justified(tl.to_abstract_assign(result.0), tm@, v@)
8       }
9       Err(t) => { tl.acceptable(t, Value::Bool(true)) }
10  })]
11  fn eval(&mut self, tl: &Trail, tm: &Term) -> (Vec<TrailIndex>, Result<Value,
      Term>) { .. }
```

Given a valid trail and and boolean term, the returned vector of indices are all contained within the trail. If the second portion of `result` is a value, then the vector justifies the evaluation of `tm` to that value. However, if instead, we returned a stuck term, that term must be *acceptable* for the trail. This specification is sufficient to derive the soundness of the complete boolean theory.

The LRA theory The LRA theory is significantly more complex than the boolean theory; its completion requires the implementation of all the rules listed in § 7.1.1. While this theory is implemented and has been informally demonstrated both sound and complete, the proof is unfinished. However, we will still present the high-level structure of this theory and our approach to its verification.

The approach taken to implementing LRA is quite similar to the Boolean theory: evaluate each term in the trail, attempting to detect contradictions, however, the details are more complex. The pseudocode for this theory is provided in Algorithm 3.

The LRA theory considers as 'variables' all subterms of real sort but consisting of symbols, not from the LRA signature, and for each tracks an interval of possible values with a set of excluded points. We evaluate each LRA-relevant assignment in the trail, but unlike the boolean theory, the return value has three possible values. We distinguish situations where an assignment A evaluates to a value, a *unit clause* with exactly one undecided LRA-variable, or a clause with *more than one* undecided LRA-variable. Like the boolean theory, we return a vector of trail indices used to evaluate the assignment.

If the evaluation produced a value, we check that it matches the expected value for A, and if not, we return a conflict. If the evaluation produced a unit clause, we use it to update the interval for the relevant LRA-variable, and check if this introduces any contradictions. Two possible contradictions are:

1. The interval is empty, indicating no possible value for this value. In this case, we perform Fourier-Motzkin resolution using the interval bounds to produce an assignment witnessing the conflict.

2. The second conflict occurs when all possible values are within our forbidden set. Forbidden values are obtained from dis-equalities in assignments. When they prevent us from choosing a variable we introduce a witness for disequality-elimination and return a conflict.

When evaluation produces a clause with more than one undecided LRA-variable, we ignore it. These clauses would serve to implement a watch-literal data structure like for CDCL solvers.

Finally, if we found no conflicts in the trail, we attempt to find possible decisions by iterating over the LRA-variables in our domain map and proposing a decision if possible. If no decision is possible, the theory is satisfied with the current state of the trail.

To verify the soundness of this algorithm, there are several properties we must establish:

1. The soundness of the evaluator. This is similar to the boolean case, but involves proofs about arithmetic which despite concerning *linear arithmetic* will involve non-linear reasoning at the Creusot level. However, apart from this wrinkle, the evaluator is a standard function.

2. The soundness of the domain map. The domain map needs to maintain an invariant that each entry is a sound approximation of the values of the LRA-variables. This is primarily derived from the correctness of the LRA-evaluator, though correctly specifying the case handling forbidden values is a moderate complication.

3. The soundness of the Fourier-Motzkin and Dis-equality resolutions. To trigger a conflict the LRA theory needs to construct a term witnessing the conflict, and this is done by either Fourier-Motzkin or Dis-equality resolution. Proving that the resolvents are sound deductions of their inputs is tricky, and we have not yet completed this proof.

Though the LRA theory is not yet proven sound, the modular structure of CDSAT allows the other components to be proven sound independently of LRA.

Algorithm 3 LRA theory implementation

```
1:  function LRA(Γ)
2:      Domain ← ∅
3:      for A ∈ Γ do
4:          (U, r) ← Eval(A)
5:          if r is a value then
6:              if r ≠ Value(A) then
7:                  return Conflict(U ∪ {A})
8:          else if r is a unit clause then
9:              Domain ← Update(Domain, r)
10:             x ← Domain(r)
11:             if x is an empty interval then
12:                 R ← FourierMotzkin(Domain, r)
13:                 return Conflict(U ∪ R)
14:             else if all valid values are forbidden elements then
15:                 R ← DisEquality(Domain, r)
16:                 return Conflict(U ∪ R)
17:     for A ∈ Domain do
18:         if IsDecision(A) then
19:             return Propose(A)
20:     return Sat
```

7.3 Evaluation

In this chapter, we have presented the design, implementation, and formalization of Sprout, a prototype SMT solver. The work on Sprout served as a test case for Creusot, pushing the complexity and scale of reasoning beyond our previous work with CreuSAT and our test cases. As a result, we uncovered limitations in the current implementation of Creusot, particularly at the level of specifications and UI. In this section, we will first present the empirical results of our verification, then discuss the limitations of Creusot from the author's perspective.

In Figure 7.3, we present the results of verifying Sprout. The times were measured on a 2021 Macbook Pro with an Apple M1 Pro and 32 GB of RAM. We used Alt-Ergo 2.4.2, Z3 4.11.2, CVC5 1.0.5 and CVC4 1.8 as the SMT solvers.

Note that the current numbers do not include results the LRA theory, which can be expected to take a significant amount of time to verify. The proof uses Why3 transformations at several critical stages, as a proof stabilization technique. These transformations are mainly used in the conflict resolution proof and a couple instances in the Trail module. Excluding the unverified LRA module, we obtained a proof overhead of roughly 1.2:1, though the proof of LRA theory modules is expected to increase this overhead. Finally, very little cleanup work was done on the proofs; there are dead lemmas, extraneous assertions, and specifications. From the author's cursory inspection, there appears to be significant room for improvement, especially in the Trail module.

7.3.1 Testing Sprout

Donald Knuth once said: "Beware of bugs in the above code; I have only proved it correct, not tried it." [56]. Despite the additional guarantees provided by mechanization, the quote still stands. To validate that Sprout can solve SMT problems, we ran it against a series of simple benchmarks.

Component	LOC	SpecLOC	Time
Theory	–	1030	21s
Trail	288	409	10s
Term	210	69	2.5s
Core loop	212	115	25s
Bool	139	53	10s
LRA	565	35	–
Total	1414	1701	68.5s

Figure 7.3: Results for the verification of Sprout. Components are disjoint portions of the code. LOC is the number of lines of code, SpecLOC is the number of lines of specification, and Time is the time taken to verify the component. **Note:** THe number for LRA may change by the date of the defense if the proof is finished.

Benchmark	# Solved	# Failed	# Timeout
Boolean	50	0	0
Arithmetic	50	0	0
Mixed	50	0	0

Figure 7.4: Results of running Sprout on a series of randomly generated benchmarks. Each problem had up to 50 constraints, with a depth of up to 10. Mixed problems contained up to 50 variables of each sort, while the theory-specific problems contained up to 100 variables. The timeout was 1s. Failures were due to the solver crashing in theory modules.

Given the project's limited scope, we did not attempt to compare the performance of Sprout against other SMT solvers.

In Figure 7.4 we can see the results of a brief evaluation of a mixed set of SMT benchmarks. The benchmarks were generated using a custom harness, and composed of arbitrary constraints of Boolean and linear arithmetic sorts. The expected answer was obtained by running Z3 and CVC5 on the same problems. The benchmarks were run with a 1s timeout; timeouts exclusively happened due to arithmetic constraints. Investigation of the timing out problems indicates that the LRA theory is incomplete: certain deductions are not made, occasionally preventing concluding unsatisfiability.

7.3.2 Experience Report

Work on Sprout first started during two months of the summer of 2022 during an internship at SRI International in collaboration with Maria Paola Bonacina and Stéphane Graham-Lengrand. After this initial period, work was halted until February 2023, at which point the author spent approximately a day a week working on the project through July 2023. This would lead to a lower bound of 3 person-months of work on the project, providing a 100% margin for error, leading to a six-person-month estimate for this project.

Though most of this time was spent addressing the inherent complexity of the verification, a non-negligible amount of time and effort can be attributed to the incidental complexity of Creusot

and its tooling. These can be divided into two categories: (1) interface issues and (2) implementation issues in Creusot.

Interface issues The interface of Creusot is still highly underdeveloped and causes friction in the verification process by breaking the flow of reasoning. Currently, Creusot does not offer a continuous, automatic mechanism to verify code. Instead, the Creusot tool should be run on Rust code and *separately* loaded into an IDE for verification. This process is cumbersome and counter-productive, forcing the developers out of their verification flow and preventing them from rapidly adding logical assertions or specifications to test their impact. While automation can be cobbled together in an ad-hoc manner, the primary interface through which proofs can be discharged, the Why3 IDE, is not designed for this purpose.

The Why3 IDE suffers from performance issues when working on more extensive proofs like Sprout, which may generate hundreds or thousands of proof obligations and reference many different source Rust files. The UI suffers from stuttering and freezing, contributing to frustration and breaking the iteration cycle. To work around this, it is possible to divide the proof into several independent segments, but this ad-hoc workaround introduces problems of *staleness*; it becomes easy for part of the proof to fall out of sync with the rest as the author no longer has a view of the entire proof at once.

The Why3 transformations present in the IDE were also a significant source of frustration. This comes from two aspects: (1) transformations can only be applied through the graphical UI of Why3 and lack basic editing features like editing a prior transformation, and more importantly, (2) transformations operate over the compiled intermediate representation of Creusot and not the source Rust code. For example, the following transformation is extracted from the proof of `resolve_conflict`:

```
1  assert (exists ix . contains2 (shallow_model8 just1) ix &&
2              index_logic3 ( * trail) ix = t)
```

The symbols `shallow_model8`, `index_logic3`, and `contains2` are all generated by Creusot and are not part of the source code. Understanding and manipulating the proof at this stage requires a deep understanding of Creusot's compilation, which fortuitously is the author's area of expertise.

These interface issues can be addressed through engineering efforts and form perhaps the principal hurdle to the usability of Creusot. Providing a fully automatic verification loop would enable much faster iteration and exploration. Having Creusot implicitly split proofs into smaller chunks while presenting a unified view would keep UI performance acceptable even for large proofs. Finally, allowing transformations and tasks to be presented in Rust syntax would make them more accessible to users and easier to maintain.

Implementation issues Beyond issues in the interface of Creusot and Why3, problems arise from the nature of auto-active verification. This mode of verification heavily relies on automated reasoning to discharge proof obligations, and as these procedures are incomplete, they may fail to discharge valid obligations. A common source of problems is the presence of *quantifiers*, as provers may end in 'matching loops' in which they repeatedly instantiate the same set of quantifiers, failing to make progress while blowing up the size of the proof context. To help avoid leading provers astray, we can control the contents of the *context* by hiding irrelevant and private definitions. During the development of Sprout, Creusot did not yet have an *opacity* functionality that allowed this. By making definitions opaque, we can hide their bodies from provers and thus prevent the proof search from getting distracted by irrelevant details. A second tool is to provide explicit *triggers* for quantifiers, instructing the prover to instantiate quantifiers in specific manners. Currently, Creusot does not support triggers, but they could be added.

Finally, the last source of complexity is more fundamental to the design of Creusot. The most significant issue is the semantics of function calls in Creusot's logic. Logical functions are considered to be *total* though they may be *under-specified*. When a logical function is provided a contract, the preconditions are *not* checked at the call site. Instead, the contract generates a quantified lemma stating that a *valid* call to the function will satisfy the postcondition. This allows for a natural encoding into SMT logic but leads to unintuitive behavior when debugging proofs. Often, proofs fail because a lemma cannot be applied due to a missing precondition, but there is no mechanism to surface this information to the user. Verus [58] showed a possible solution to this challenge through `recommends` clauses. If a proof fails, Verus will attempt to assert the validity of relevant logical function preconditions, and if they fail, it will suggest them as possible sources of proof failure. Solving this challenge completely is complex and may involve making tradeoffs in performance and expressivity.

7.4 Related Works

Verified Rust programs Due to the relative youth of Rust verification, few large-scale projects have been undertaken. CreuSAT [46] was a prior inspiration for this work, demonstrating it was possible to verify an optimized SAT solver written entirely in Rust within a short time frame. Unlike our work, CreuSAT follows a 'bottom-up' approach to specification, which, while allowing for more immediate feedback and iteration, renders the broader architecture of the proof complex and challenging to understand. In "Flux" [59], the authors use liquid typing to verify the memory safety of the Wave [47] Web Assembly sandbox. The more limited expressivity of the Flux liquid types limits this work to basic safety properties. The authors demonstrate the usefulness of Flux by applying it to consequential Rust programs.

Verification of automated solvers Though most industrial SAT and SMT solvers prefer to use *certificates* as their approach to verification, a few projects have nonetheless endeavored to verify their implementations directly. The IsaSAT [11] solver is an optimized and verified implementation of a CDCL solver verified in Isabelle/HOL. Like our work, the IsaSAT proof includes an abstract formalization of the CDCL algorithm, progressively refined into a concrete, optimized implementation. In "A Verified SAT Solver with Watched Literals Using Imperative HOL", the authors advocate for the benefits of a *refinement* based approach to verification, which we also use in our work. Isabelle's support for refinement is much more developed and automated than Creusot's, but this could indicate future directions for the project. The authors also remark on the challenge of working with a longer 'edit-compile-verify' loop like the one we evoked in § 7.3.2, driving home the importance of short iteration cycles. CoqSolverFD [20] is a verified Coq implementation of a *finite domain* constraint solver. They establish the soundness and completeness of their solver and obtain an executable Coq program through extraction.

The work most directly comparable to Sprout is found in "Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq" [63], which introduces a formalization of the algorithms from the SMT solver Alt-Ergo, mechanized in Coq. The author formalizes a DPLL($\mathcal{T}$) theory, including soundness, completeness, and termination. They also introduce a novel algorithm CC(X), permitting the combination of the theory of Equality and Uninterpreted Functions (EUF) with another solvable theory. The solver is instantiated as a *tactic* in the Coq [24] proof assistant, allowing it to be used to discharge proof obligations in Coq. It can also be extracted [64] and run as an OCaml program.

The formalization work undertaken in this project far exceeds that of Sprout, including proofs of *completeness*, which requires reasoning about theory combinations under schemes like Nelson-

Oppen [89]. As Why3 permits using Coq as a verification backend, it may be possible to reuse portions of this formalization in future extensions of Sprout. Because Lescuyer's work is written in Coq, the proof is significantly more manual than ours; the authors estimate the total size at approximately 47kloc, though this includes the development of several stand-alone libraries. In comparison, Sprout comes out at approximately 2kloc, including auxiliary proofs and specifications.

The more common and practical approach to verifying the results of SMT solvers is through *certificates*. When a solver produces unsat, it also emits a proof of unsatisfiability that can be checked independently. SMTCoq [30] can load these certificates into Coq and check their validity. This approach lifts the verification burden from SMT solver s; they can use any technique they wish so long as the produced certificate is valid. Instead of verifying a certificate checker, it is also possible to 'generate' 'certificates' using a verified LCF-style kernel, this approach shrinks the size of the verified code base and enables the usage of untrusted operations in the solver. This approach was used in the Verified Polyhedral Library (VPL) [17] to establish the correctness of a domain of abstract polyhedrons, which poses similar challenges to the LRA domain found in Sprout. In this model [16], the untrusted oracles build their result by invoking the verified LCF-primitive operations. Through polymorphism and its accompanying parametricity, it becomes possible to prevent 'cheating' by the oracles.

Chapter 8

Conclusions

In this dissertation, we have attempted to answer the eternal question of the field 'Is formal verification of software tractable?'. Software verification has haunted the field of programming languages ever since the first program was written [3, 22]. While the question is still not solved, we have shown that we have made concrete steps to answering the affirmative.

As evidence for this claim, we have provided an existential witness by constructing a complete Rust verification platform Creusot and applying it to the verification of novel large-scale Rust programs and libraries. In Chapter 5, we formalize a verification procedure that leverages the Rust-type system's ownership properties to simplify the complexity of program verification fundamentally. While in Chapter 4, we apply and extend this scheme to handle actual Rust code, with all its complexities. The opportunities presented by the notion of ownership serve to belie the notion that programming languages are all equal, demonstrating the importance of language design in the context of verification.

The rest of this work applies Creusot to the verification of complex patterns of iteration, and we present a novel specification schema for *higher-order* iterators with *side-effects*. Our proofs of iterators like `IterMut` served to drive home the importance of language on verification, as our natural and concise specifications emerge directly from the design of Pearlite, the specification language of Creusot.

Finally, we demonstrate that Creusot enables the verification of actual, complex software beyond what has been previously accomplished. We presented the implementation and verification of an SMT solver based on a modern SMT algorithm, a task we accomplished with a very modest workforce investment. An SMT solver is a notoriously challenging piece of software that uses imperative structures to perform semantically complex operations; after all, it solves a fragment of first-order logic. Perhaps more substantial evidence for the improvements to verification made possible by Creusot is found in CreuSAT [46]. This project implemented and verified a SAT solver in Rust using Creusot, competitive with but slower than state-of-the-art unverified solvers. More importantly, the author, Høverstad Skotåm, was unfamiliar with formal verification and Rust when starting the project. Over approximately six months, they taught, learned to use, and apply Creusot, and produced CreuSAT. The state of the art for verified SAT solvers is IsaSAT [11], the product of a Ph.D. thesis and several years of work. While both works are not direct comparisons, we believe that the comparison illustrates the improvements to verification made possible by Creusot.

8.0.1 Collaborations

Additionally, beyond publications, Creusot has been featured in several masters' theses and internships. Each of these cases involved collaboration with the student(s) in question and served as a source of feedback and improvements to Creusot. As a way of thanking these students for their time, commitment and trust, we list their works below:

- "CreuSAT Using Rust and CREUSOT to Create the World's Fastest Deductively Verified SAT Solver" written in 2022 by Sarek Høverstad Skotåm, presents the implementation of a verified SAT solver in Rust. The resulting solver is highly optimized and closely trails IsaSAT [11] – the other major verified SAT solver – in performance. [1]
- "Verifying the Rust Runtime of Lingua Franca" written by Johannes Hayeß in 2022 uses Creusot to verify parts of the Lingua Franca [67] runtime. By verifying a pre-existing Rust program, Johannes highlights the real issues in integrating Creusot into a real-world project. His work included specifying new libraries and developing proof for key data structures of Lingua Franca.
- "Type invariants and ghost code for deductive verification of Rust code" written in 2023 Dominik Stolz is the result of an internship at the Laboatoires Méthodes Formelles in which he implemented *type invariants* in Creusot. This work had many unexpected and subtle consequences on critical aspects of Creusot's translation. The consequences of this work were not presented in this thesis.

8.1 Future Work

Naturally, the work is not over; each proof done with Creusot highlights new areas of difficulty, and each new Rust feature added to Creusot drives demand for two more.

Rust Verification The subfield of Rust verification has exploded in the last few years [59, 43, 58], but there are still many areas of Rust with no satisfactory verification story. We believe that Creusot provides a compelling answer to the verification of *safe* programs abiding by ownership discipline, but many Rust programs (locally) break one or both of these constraints. Verifying programs with *Interior Mutability* is an exciting area of future work. Interior mutability is used for synchronization with `Mutex` but also to implement *memoization* or to construct cyclic data structures. While RustHornBelt includes a model of interior mutability, in practice, this has been insufficient and cumbersome to use. Extending the approach of Creusot to co-exist with a model of interior mutability is a crucial direction for future work, as it will help open the door to verifying *concurrent* programs. Many applications for formal verification of Rust lie in domains (e.g., embedded systems) where concurrency is a fundamental concern. Proofs of concurrent programs are notoriously difficult, but unlike other languages, the Rust-type system provides a strong foundation for simplifying concurrent reasoning by enforcing data-race freedom. Developing a framework for verifying *safe* concurrent programs would help Creusot make another step towards real-world verification. Finally, verifying `unsafe` programs has preoccupied many researchers [70, 58, 38]. Finding a way to bridge the gap between the techniques used when verifying low-level unsafe fragments with the more significant, safe proofs performed by Creusot can be considered a 'holy grail.' The ideal approach enables Pay-As-You-Go verification, where verification's complexity is directly proportional to the code's complexity. Given that Creusot is intrinsically limited to *safe* code, this

[1] When this work started, Creusot did not even support `Vec<T>`! The work Sarek did was invaluable in making Creusot usable at *all* for real verification

would mean marrying it with another technology capable of verifying *unsafe* code while ensuring both tools can interact seamlessly.

Rust Specification The Rust community has a solid cooperative and centralizing streak fostered by the organization around the Rust Project. Rust verifiers are no exception, and the possibility of having verification tools interact and collaborate to verify more significant properties has existed since the start. Accomplishing this requires a common specification language with a **shared semantics**. The existing specification languages of projects like Verus, Prusti, and Creusot are scattered across the semantic spectrum. Where Creusot interprets logical symbols as total functions, Prusti interprets them as partial; where Verus distinguishes between *linear* and *non-linear* specifications, Creusot makes no such distinction. Designing a specification language to replace Pearlite that reconciles these differences without overly constraining the *research* projects that would use it would also help build a shared community around Rust verification, furthering our goal of making verification tractable.

Verification Experience While adequate formalism can vastly simplify the challenge of verification – like the prophetic approach does for mutable borrows in Rust – it is just the first ingredient in the *verification experience*. Experience using Creusot highlights the importance of 'secondary' factors in verification: the design of the specification language, the modalities of user interaction, and the tools for 'debugging' proofs. These are the facets through which users directly interact with the verification process, and yet their development and design are usually ad-hoc and ungrounded in a formal investigation.

Assertions of intuitivity are common between authors of verification tools, but what do users consider intuitive? Few studies have been performed on this subject; a better understanding of how engineers and programmers informally reason about and state the properties of their programs would help us design better specification languages. In the design of Pearlite, this would help settle questions about its semantics: Should functions be partial? Should specifications be executable?

'Intuition' has also been a guide in the design of verifier interfaces. Many verifiers believe proofs should be fully automatic and the only user interaction should be *assertions*. This style of *auto-active* verification is appealing; it avoids forcing the user to learn a tool, but what happens when it fails to prove? Creusot, which is based on Why3, inherits its notion of *transformations* and *task view*, which allow users to run custom tactics and inspect the state of the proof. These features are powerful, allowing expert users to extend their capabilities, but are poorly integrated into the program verification process. They hint at a different verification experience, where, despite high automation, the user can be provided tools to understand failures and debug proofs.

Bibliography

[1] *A Proactive Approach to More Secure Code | MSRC Blog | Microsoft Security Response Center*. URL: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/ (visited on 08/23/2023).

[2] Martín Abadi and Leslie Lamport. "The Existence of Refinement Mappings". In: *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 1988, pp. 165–175. ISBN: 0-8186-0853-6. DOI: 10.1109/LICS.1988.5115. URL: https://doi.org/10.1109/LICS.1988.5115.

[3] William Aspray. "Difference and Analytical Engines". In: *Computing before Computers*.

[4] Vytautas Astrauskas et al. "Leveraging Rust Types for Modular Specification and Verification". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), 147:1–147:30. DOI: 10.1145/3360573. URL: https://doi.org/10.1145/3360573.

[5] Brenda S. Baker. "An Algorithm for Structuring Flowgraphs". en. In: *Journal of the ACM* 24.1 (Jan. 1977), pp. 98–120. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/321992.321999.

[6] Haniel Barbosa et al. "Cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 415–442. ISBN: 978-3-030-99524-9. DOI: 10.1007/978-3-030-99524-9_24.

[7] Mike Barnett and K Rustan M Leino. "Weakest-Precondition of Unstructured Programs". In: *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 2005, pp. 82–87.

[8] Mike Barnett et al. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *Formal Methods for Components and Objects*. Ed. by Frank S. Boer et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 364–387. ISBN: 978-3-540-36750-5. DOI: 10.1007/11804192_17.

[9] Clark Barrett and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 305–343. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_11. URL: https://doi.org/10.1007/978-3-319-10575-8_11 (visited on 08/23/2023).

[10] Clark W. Barrett et al. "CVC4". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177. ISBN: 978-3-642-22109-5. DOI: 10.1007/978-3-642-22110-1_14. URL: https://doi.org/10.1007/978-3-642-22110-1_14.

[11] Jasmin Christian Blanchette et al. "A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality". In: *Journal of Automated Reasoning* 61.1 (June 1, 2018), pp. 333–365. ISSN: 1573-0670. DOI: 10.1007/s10817-018-9455-7. URL: https://doi.org/10.1007/s10817-018-9455-7 (visited on 08/18/2023).

[12] François Bobot et al. "Let's Verify This with Why3". In: *Int. J. on Software Tools for Technology Transfer* 17.6 (2015), pp. 709–727. DOI: 10.1007/s10009-014-0314-5.

[13] François Bobot et al. "Centralizing Equality Reasoning in MCSAT". In: *16th International Workshop on Satisfiability Modulo Theories (SMT 2018)*. Ed. by R. Dimitrova and V. D'Silva. Oxford, United Kingdom, July 2018. URL: https://hal.science/hal-01935591 (visited on 08/23/2023).

[14] Corrado Böhm and Giuseppe Jacopini. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules". en. In: *Communications of the ACM* 9.5 (May 1966), pp. 366–371. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/355592.365646.

[15] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Natarajan Shankar. "Conflict-Driven Satisfiability for Theory Combination: Transition System and Completeness". In: *Journal of Automated Reasoning* 64.3 (Mar. 2020), pp. 579–609. ISSN: 0168-7433, 1573-0670. DOI: 10.1007/s10817-018-09510-y. URL: http://link.springer.com/10.1007/s10817-018-09510-y (visited on 05/30/2022).

[16] Sylvain Boulmé. "Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles)". PhD thesis. Université Grenoble-Alpes, 2021.

[17] Sylvain Boulmé et al. "The verified polyhedron library: an overview". In: *2018 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE. 2018, pp. 9–17.

[18] François Bourdoncle. "Efficient Chaotic Iteration Strategies with Widenings". In: *Formal Methods in Programming and Their Applications*. Ed. by Dines Bjørner, Manfred Broy, and Igor V. Pottosin. Vol. 735. Berlin/Heidelberg: Springer-Verlag, 1993, pp. 128–141. ISBN: 978-3-540-57316-6. DOI: 10.1007/BFb0039704. URL: http://link.springer.com/10.1007/BFb0039704 (visited on 06/14/2023).

[19] Qinxiang Cao et al. "VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs". In: *Journal of Automated Reasoning* 61.1 (June 1, 2018), pp. 367–422. ISSN: 1573-0670. DOI: 10.1007/s10817-018-9457-5. URL: https://doi.org/10.1007/s10817-018-9457-5 (visited on 08/23/2023).

[20] Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. "A Certified Constraint Solver over Finite Domains". In: *FM 2012: Formal Methods*. Ed. by Dimitra Giannakopoulou and Dominique Méry. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 116–131. ISBN: 978-3-642-32759-9. DOI: 10.1007/978-3-642-32759-9_12.

[21] Arthur Charguéraud. "Characteristic Formulae for the Verification of Imperative Programs". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. New York, NY, USA: Association for Computing Machinery, Sept. 19, 2011, pp. 418–430. ISBN: 978-1-4503-0865-6. DOI: 10.1145/2034773.2034828. URL: https://dl.acm.org/doi/10.1145/2034773.2034828 (visited on 08/23/2023).

[22] *Checking a Large Routine | The Early British Computer Conferences*. URL: https://dl.acm.org/doi/abs/10.5555/94938.94952 (visited on 08/23/2023).

[23] Sylvain Conchon et al. "Alt-Ergo 2.2". In: SMT Workshop: International Workshop on Satisfiability Modulo Theories. July 12, 2018. URL: https://inria.hal.science/hal-01960203 (visited on 09/01/2023).

[24] Coq Community. *The Coq Proof Assistant*. 2021. URL: https://coq.inria.fr/.

[25] Sylvain Dailler, Claude Marché, and Yannick Moy. "Lightweight Interactive Proving inside an Automatic Program Verifier". In: *Formal Integrated Development Environment*. 2018. DOI: 10.4204/EPTCS.284.1.

[26] Xavier Denis and Jacques-Henri Jourdan. "Specifying and Verifying Higher-order Rust Iterators". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2023, pp. 93–110.

[27] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. "Creusot: A Foundry for the Deductive Verication of Rust Programs". In: *ICFEM*. Vol. 13478. LNCS. 2022. DOI: 10.1007/978-3-031-17244-1_6.

[28] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. ISBN: 013215871X. URL: https://www.worldcat.org/oclc/01958445.

[29] Edsger W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In: *Communications of the ACM* 18.8 (Aug. 1, 1975), pp. 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975. URL: https://dl.acm.org/doi/10.1145/360933.360975 (visited on 08/23/2023).

[30] Burak Ekici et al. "SMTCoq: A Plug-In for Integrating SMT Solvers into Coq". In: *Computer Aided Verification*. Ed. by Rupak Majumdar and Viktor Kunčak. Vol. 10427. Cham: Springer International Publishing, 2017, pp. 126–133. ISBN: 978-3-319-63389-3 978-3-319-63390-9. DOI: 10.1007/978-3-319-63390-9_7. URL: https://link.springer.com/10.1007/978-3-319-63390-9_7 (visited on 09/12/2023).

[31] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 - Where Programs Meet Provers". In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 125–128. ISBN: 978-3-642-37035-9. DOI: 10.1007/978-3-642-37036-6_8. URL: https://doi.org/10.1007/978-3-642-37036-6_8.

[32] Jean-Christophe Filliâtre and Mário Pereira. "A Modular Way to Reason About Iteration". In: *NASA Formal Methods*. Ed. by Sanjai Rayadurgam and Oksana Tkachuk. Vol. 9690. Cham: Springer International Publishing, 2016, pp. 322–336. ISBN: 978-3-319-40647-3 978-3-319-40648-0. DOI: 10.1007/978-3-319-40648-0_24. (Visited on 10/10/2022).

[33] Jean-Christophe Filliâtre. “Formal Verification of MIX Programs”. In: *Journ 'ees en l'honneur de Donald E. Knuth*. urlhttp://knuth07.labri.fr/exposes.php. Bordeaux, France, 2007. URL: http://www.lri.fr/~filliatr/publis/verifmix.pdf.

[34] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. “The Spirit of Ghost Code”. In: *Formal Methods in System Design* 48.3 (June 2016), pp. 152–174. ISSN: 0925-9856, 1572-8102. DOI: 10.1007/s10703-016-0243-x. URL: http://link.springer.com/10.1007/s10703-016-0243-x (visited on 06/08/2023).

[35] Jean-Christophe Filliâtre and Andrei Paskevich. “Abstraction and Genericity in Why3”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 12476. Cham: Springer International Publishing, 2020, pp. 122–142. ISBN: 978-3-030-61361-7 978-3-030-61362-4. DOI: 10.1007/978-3-030-61362-4_7. URL: http://link.springer.com/10.1007/978-3-030-61362-4_7 (visited on 09/12/2023).

[36] Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. “A Verified SAT Solver with Watched Literals Using Imperative HOL”. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP ’18: Certified Proofs and Programs. Los Angeles CA USA: ACM, Jan. 8, 2018, pp. 158–171. ISBN: 978-1-4503-5586-5. DOI: 10.1145/3167080. URL: https://dl.acm.org/doi/10.1145/3167080 (visited on 08/18/2023).

[37] Robert W. Floyd. “Assigning Meanings to Programs”. In: *Program Verification: Fundamental Issues in Computer Science*. Ed. by Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin. Studies in Cognitive Systems. Dordrecht: Springer Netherlands, 1993, pp. 65–81. ISBN: 978-94-011-1793-7. DOI: 10.1007/978-94-011-1793-7_4. URL: https://doi.org/10.1007/978-94-011-1793-7_4 (visited on 08/23/2023).

[38] Nima Rahimi Foroushaani and Bart Jacobs. *Modular Formal Verification of Rust Programs with Unsafe Blocks*. Dec. 25, 2022. DOI: 10.48550/arXiv.2212.12976. arXiv: 2212.12976 [cs]. URL: http://arxiv.org/abs/2212.12976 (visited on 08/18/2023). preprint.

[39] Harald Ganzinger et al. “DPLL(T): Fast Decision Procedures”. In: *Computer Aided Verification*. Ed. by Rajeev Alur and Doron A. Peled. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 175–188. ISBN: 978-3-540-27813-9. DOI: 10.1007/978-3-540-27813-9_14.

[40] Léon Gondelman. “Un système de types pragmatique pour la vérification déductive des programmes. (A Pragmatic Type System for Deductive Verification)”. PhD thesis. University of Paris-Saclay, France, 2016. URL: https://tel.archives-ouvertes.fr/tel-01533090.

[41] Stéphane Graham-Lengrand, Dejan Jovanović, and Bruno Dutertre. “Solving Bitvectors with MCSAT: Explanations from Bits and Pieces”. In: *Automated Reasoning*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 103–121. ISBN: 978-3-030-51074-9. DOI: 10.1007/978-3-030-51074-9_7.

[42] David Harel. “On Folk Theorems”. In: *Communications of the ACM* 23.7 (July 1, 1980), pp. 379–389. ISSN: 0001-0782. DOI: 10.1145/358886.358892. URL: https://dl.acm.org/doi/10.1145/358886.358892 (visited on 09/05/2023).

[43] Son Ho and Jonathan Protzenko. "Aeneas: Rust Verification by Functional Translation". In: *ICFP*. 2022. DOI: 10.1145/3547647.

[44] Tony Hoare. *An Axiomatic Basis for Computer Programming | Communications of the ACM*. URL: https://dl.acm.org/doi/abs/10.1145/363235.363259 (visited on 08/23/2023).

[45] Thierry Hubert and Claude Marché. "Separation Analysis for Deductive Verification". In: *Heap Analysis and Verification*. 2007, pp. 81–93. URL: https://hal.inria.fr/hal-03630177.

[46] Sarek Høverstad Skotåm. "CreuSAT Using Rust and CREUSOT to Create the World's Fastest Deductively Verified SAT Solver". University of Oslo. URL: https://sarsko.github.io/_pages/SarekSkot%C3%A5m_thesis.pdf.

[47] Evan Johnson et al. "WaVe: A Verifiably Secure WebAssembly Sandboxing Runtime". In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023 IEEE Symposium on Security and Privacy (SP). May 2023, pp. 2940–2955. DOI: 10.1109/SP46215.2023.10179357.

[48] Dejan Jovanovic, Clark Barrett, and Leonardo Moura. "The Design and Implementation of the Model Constructing Satisfiability Calculus". In: *2013 Formal Methods in Computer-Aided Design*. 2013 Formal Methods in Computer-Aided Design. Oct. 2013, pp. 173–180. DOI: 10.1109/FMCAD.2013.7027033.

[49] Dejan Jovanović. "Solving Nonlinear Integer Arithmetic with MCSAT". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ahmed Bouajjani and David Monniaux. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 330–346. ISBN: 978-3-319-52234-0. DOI: 10.1007/978-3-319-52234-0_18.

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

[51] Ralf Jung et al. "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. 2015, pp. 637–650. DOI: 10.1145/2676726.2676980. URL: https://doi.org/10.1145/2676726.2676980.

[52] Ralf Jung et al. *RustBelt: Securing the Foundations of the Rust Programming Language — Technical Appendix*. 2018. URL: https://plv.mpi-sws.org/rustbelt/popl18/appendix.pdf.

[53] Ralf Jung et al. "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. URL: https://doi.org/10.1145/3158154.

[54] Ralf Jung et al. "The Future is Ours: Prophecy Variables in Separation Logic". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2020), 45:1–45:32. DOI: 10.1145/3371113. URL: https://doi.org/10.1145/3371113.

[55] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP09: ACM SIGOPS 22nd Symposium on Operating Systems Principles. Big Sky Montana USA: ACM, Oct. 11, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: https://dl.acm.org/doi/10.1145/1629575.1629596 (visited on 09/01/2023).

[56] *Knuth: Frequently Asked Questions*. URL: https://www-cs-faculty.stanford.edu/~knuth/faq.html (visited on 08/17/2023).

[57] Laura Kovács and Andrei Voronkov. "First-Order Theorem Proving and Vampire". In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 1–35. ISBN: 978-3-642-39799-8. DOI: 10.1007/978-3-642-39799-8_1.

[58] Andrea Lattuada et al. "Verus: Verifying Rust Programs Using Linear Ghost Types". In: *Proceedings of the ACM on Programming Languages* 7 (OOPSLA1 Apr. 6, 2023), 85:286–85:315. DOI: 10.1145/3586037. URL: https://dl.acm.org/doi/10.1145/3586037 (visited on 05/16/2023).

[59] Nico Lehmann et al. "Flux: Liquid Types for Rust". In: *Proceedings of the ACM on Programming Languages* 7 (PLDI June 6, 2023), pp. 1533–1557. ISSN: 2475-1421. DOI: 10.1145/3591283. URL: https://dl.acm.org/doi/10.1145/3591283 (visited on 08/18/2023).

[60] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370. ISBN: 978-3-642-17510-7 978-3-642-17511-4. DOI: 10.1007/978-3-642-17511-4_20. URL: http://link.springer.com/10.1007/978-3-642-17511-4_20 (visited on 09/01/2023).

[61] K. Rustan M. Leino and Michał Moskal. "VACID-0: Verification of Ample Correctness of Invariants of Data-structures, Edition 0". In: *Verified Software, Tools, Techniques and Experiments*. 2010.

[62] Xavier Leroy et al. "CompCert - A Formally Verified Optimizing Compiler". In: ().

[63] Stephane Lescuyer. "Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq". These de doctorat. Paris 11, Jan. 4, 2011. URL: https://www.theses.fr/2011PA112363 (visited on 08/18/2023).

[64] Pierre Letouzey. "Extraction in Coq: An Overview". In: *Logic and Theory of Algorithms*. Ed. by Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 359–369. ISBN: 978-3-540-69407-6. DOI: 10.1007/978-3-540-69407-6_39.

[65] Jialin Li et al. "Linear Types for Large-Scale Systems Verification". In: *Proceedings of the ACM on Programming Languages* 6 (OOPSLA1 Apr. 29, 2022), pp. 1–28. ISSN: 2475-1421. DOI: 10.1145/3527313. URL: https://dl.acm.org/doi/10.1145/3527313 (visited on 05/16/2023).

[66] Hongjin Liang and Xinyu Feng. "Modular Verification of Linearizability with Non-Fixed Linearization Points". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 459–470. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462189. URL: https://doi.org/10.1145/2491956.2462189.

[67] Marten Lohstroh et al. "Toward a Lingua Franca for Deterministic Concurrent Systems". In: *ACM Transactions on Embedded Computing Systems* 20.4 (May 18, 2021), 36:1–36:27. ISSN: 1539-9087. DOI: 10.1145/3448128. URL: https://dl.acm.org/doi/10.1145/3448128 (visited on 08/23/2023).

[68] Joao Marques-Silva, Ines Lynce, and Sharad Malik. "Chapter 4. Conflict-Driven Clause Learning SAT Solvers". In: *Handbook of Satisfiability*. IOS Press, 2021, pp. 133–182. DOI: 10.3233/FAIA200987. URL: https://ebooks.iospress.nl/doi/10.3233/FAIA200987 (visited on 08/23/2023).

[69] 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*. Ed. by Peter Müller. Vol. 12075. Lecture Notes in Computer Science. Springer, 2020, pp. 484–514. ISBN: 978-3-030-44913-1. DOI: 10.1007/978-3-030-44914-8_18. URL: https://doi.org/10.1007/978-3-030-44914-8_18.

[70] Yusuke Matsushita et al. "RustHornBelt: A Semantic Foundation for Functional Verification of Rust Programs with Unsafe Code". In: *PLDI*. 2022. DOI: 10.1145/3519939.3523704.

[71] Michael Mol and other contributors. *The Rosetta Code Chrestomathy of Programs*. URL: http://rosettacode.org (visited on 10/01/2021).

[72] João Mota, Marco Giunti, and António Ravara. *On Using VeriFast, VerCors, Plural, and KeY to Check Object Usage*. Sept. 2022. arXiv: 2209.05136 [cs]. (Visited on 10/10/2022).

[73] Leonardo Mendonça de Moura and Nikolaj 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. ISBN: 978-3-540-78799-0. DOI: 10.1007/978-3-540-78800-3_24. URL: https://doi.org/10.1007/978-3-540-78800-3_24.

[74] 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, pp. 41–62. DOI: 10.1007/978-3-662-49122-5_2. URL: https://doi.org/10.1007/978-3-662-49122-5_2.

[75] Thi Minh Tuyen Nguyen. "Taking Architecture and Compiler into Account in Formal Proofs of Numerical Programs". These de doctorat. Paris 11, June 11, 2012. URL: https://www.theses.fr/2012PA112090 (visited on 06/14/2023).

[76] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures". In: *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL*. Ed. by Laurent Fribourg. Vol. 2142. Lecture Notes in Computer Science. Springer, 2001, pp. 1–19. ISBN: 3-540-42554-3. DOI: 10.1007/3-540-44802-0_1. URL: https://doi.org/10.1007/3-540-44802-0_1.

[77] Lawrence C. Paulson, ed. *Isabelle*. Vol. 828. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer-Verlag, 1994. ISBN: 978-3-540-58244-1. DOI: 10.1007/BFb0030541. URL: http://link.springer.com/10.1007/BFb0030541 (visited on 09/01/2023).

[78] Benjamin C Pierce. "Software Foundations". In: ().

[79] Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. "A Fully Verified Container Library". In: *Formal Aspects of Computing* 30.5 (Sept. 2018), pp. 495–523. ISSN: 0934-5043, 1433-299X. DOI: 10.1007/s00165-017-0435-1. (Visited on 10/10/2022).

[80] François Pottier. "Verifying a Hash Table and Its Iterators in Higher-Order Separation Logic". In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. Paris France: ACM, Jan. 2017, pp. 3–16. ISBN: 978-1-4503-4705-1. DOI: 10.1145/3018610.3018624. (Visited on 10/10/2022).

[81] Ralf Jung and The MIRI developers. *MIRI*. URL: https://github.com/rust-lang/miri.

[82] Rust Community. *Rust Programming Language*. 2021. URL: https://www.rust-lang.org/.

[83] Michael Sammler et al. "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, 20211*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 158–174. ISBN: 978-1-4503-8391-2. DOI: 10.1145/3453483.3454036. URL: https://doi.org/10.1145/3453483.3454036.

[84] Stephan Schulz. "E – A Brainiac Theorem Prover". In: ().

[85] *The Great Theorem Prover Showdown*. Hillel Wayne. URL: https://www.hillelwayne.com/post/theorem-prover-showdown/ (visited on 10/14/2022).

[86] The Rust Community. *The `std::cmp::Ord` trait of Rust*. Version 1.0.0. Oct. 29, 2021. URL: https://doc.rust-lang.org/std/cmp/trait.Ord.html.

[87] The Rust Community. *The `std::vec::Vec::push` method of Rust*. Version 1.0.0. Oct. 29, 2021. URL: https://doc.rust-lang.org/std/vec/struct.Vec.html#method.push.

[88] The Rust UCG Working Group. *Unsafe Code Guidelines*. URL: https://rust-lang.github.io/unsafe-code-guidelines/.

[89] Cesare Tinelli and Mehdi Harandi. "A New Correctness Proof of the Nelson-Oppen Combination Procedure". In: *Frontiers of Combining Systems: First International Workshop, Munich, March 1996*. Ed. by Frans Baader and Klaus U. Schulz. Applied Logic Series. Dordrecht: Springer Netherlands, 1996, pp. 103–119. ISBN: 978-94-009-0349-4. DOI: 10.1007/978-94-009-0349-4_5. URL: https://doi.org/10.1007/978-94-009-0349-4_5 (visited on 08/23/2023).

[90] Aaron Joseph Turon et al. "Logical Relations for Fine-Grained Concurrency". In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 343–356. ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429111. URL: https://doi.org/10.1145/2429069.2429111.

[91] Viktor Vafeiadis. "Modular Fine-Grained Concurrency Verification". PhD thesis. University of Cambridge, UK, 2008. URL: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.612221.

[92] Paulo Emílio de Vilhena, François Pottier, and Jacques-Henri Jourdan. "Spy Game: Verifying a Local Generic Solver in Iris". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2020), 33:1–33:28. DOI: 10.1145/3371101. URL: https://doi.org/10.1145/3371101.

[93] P. Wadler and S. Blott. "How to Make Ad-Hoc Polymorphism Less Ad Hoc". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '89*. The 16th ACM SIGPLAN-SIGACT Symposium. Austin, Texas, United States: ACM Press, 1989, pp. 60–76. ISBN: 978-0-89791-294-5. DOI: 10.1145/75277.75283. URL: http://portal.acm.org/citation.cfm?doid=75277.75283 (visited on 02/14/2023).

[94] Fabian Wolff et al. "Modular specification and verification of closures in Rust". In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (2021), pp. 1–29.

[95] Joshua Yanovski et al. "GhostCell: Separating Permissions from Data in Rust". In: *Proceedings of the ACM on Programming Languages* 5 (ICFP Aug. 22, 2021), pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3473597. URL: https://dl.acm.org/doi/10.1145/3473597 (visited on 05/16/2023).

[96] Zipeng Zhang et al. "A Structural Approach to Prophecy Variables". In: *Theory and Applications of Models of Computation - 9th Annual Conference, TAMC*. Ed. by Manindra Agrawal, S. Barry Cooper, and Angsheng Li. Vol. 7287. Lecture Notes in Computer Science. Springer, 2012, pp. 61–71. ISBN: 978-3-642-29951-3. DOI: 10.1007/978-3-642-29952-0_12. URL: https://doi.org/10.1007/978-3-642-29952-0_12.

université
PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies
de l'information et de
la communication (STIC)