SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Type Invariants and Ghost Code in Rust Verification with Creusot

Typinvarianten und Ghostcode in Rust-Verifikation mit Creusot

Author:	Dominik Stolz
Supervisor:	Prof. Tobias Nipkow
Advisor:	Jacques-Henri Jourdan, Xavier Denis
Submission Date:	15.11.2023

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.11.2023 Dominik Stolz

Acknowledgments

I extend my sincere thanks to my advisors, Jacques-Henri Jourdan and Xavier Denis, for their invaluable guidance and steadfast support. Xavier's pivotal encouragement to apply for a research internship at the Laboratoire Méthodes Formelles significantly enriched my research journey. This opportunity was only made possible thanks to Jacques-Henri's relentless dedication to surmounting bureaucratic challenges. Their expertise and keen insights have been instrumental to the completion of this thesis.

I am also grateful to the entire team at the Laboratoire Méthodes Formelles for their warm welcome and stimulating discussions. On a personal note, I wish to thank my family and friends for supporting me throughout this project.

Abstract

This thesis explores the adoption of two established specification patterns within the Rust verifier Creusot: type invariants and ghost code. Type invariants empower users to attach logical predicates to data types, which Creusot enforces for all values of a type. Ghost code enables auxiliary constructs that bolster verification without affecting the program's runtime behavior. Especially when employed in tandem, these patterns enhance the expressivity of specifications and ensure the scalability of Rust verification. We introduce an innovative framework for type invariants, including its implementation in Creusot. Moreover, we scrutinize Creusot's current implementation of ghost code, identifying soundness deficiencies and presenting refinements to address them. Our evaluation demonstrates that these extensions not only facilitate more concise program specifications but also maintain robust verifiability.

Contents

1 Introduction

Writing correct computer programs is hard. Programmers must grasp not only the intricacies of the problem they aim to solve but also the exact semantics of their chosen programming language. As a result, most programs unsurprisingly contain errors. The tolerance for these errors varies depending on the type of error and the program's intended usage context. Consequently, methods of varying thoroughness have been developed to gain confidence in a program's correctness. For most applications, identifying errors through testing and rectifying them retroactively suffices. However, in critical applications where software flaws could have grave consequences up to the point of endangering lives, far higher assurances are needed.

Enter Rust [MK14], a programming language designed to provide heightened safety guarantees and thus a representative of language-based approaches to increasing confidence in programs. It has gained popularity thanks to its promise to overcome the classical tradeoff between high-level safety and low-level control [Jun+21]. Unlike most safe languages, Rust does not incur overhead due to superfluous memory allocations or costly runtime checks. Unlike most low-level languages, well-typed Rust programs statically guarantee the absence of memory safety errors. Adoption of Rust for safety-critical applications [Gil21] bears witness to the success of this approach.

While memory safety already rules out large classes of errors, ensuring a program's functional correctness demands more potent methods. A program is functionally correct if it not only never performs illegal operations but also always computes the expected result. Establishing this property is the goal of deductive program verification, aiming to mathematically prove that a program behaves as expected in all possible scenarios.

Rust is a prime candidate for writing verified programs, especially among imperative programming languages with pointers. Traditionally, verification of programs in such languages presents a challenge due to the complexities of reasoning about aliasing pointers. Aliasing occurs when multiple pointers reference the same memory location. This complicates verification since changes made via one pointer can affect data accessed by another. However, by leveraging the guarantees of well-typed Rust programs, reasoning about aliasing becomes far more tractable.

The Creusot tool [DJM22] builds upon this insight to verify Rust programs. It distinguishes itself from other Rust verifiers, such as Prusti [Ast+19] or Verus [Lat+23], through its use of prophecy variables to encode pointers. This enables Creusot to translate Rust into a functional representation, avoiding the need to reason about general pointer programs.

1.1 Motivation

Scaling verification to larger programs faces a disproportional increase in the complexity of specifications. As a countermeasure, encapsulation patterns have emerged to confine the complexity behind an abstraction barrier. In this work, we focus on two complementary verification patterns: *type invariants* and *ghost code*. While these patterns are already well understood for other languages [Coh+09; Lei10; FGP16], their adoption for Rust and Creusot presents unique challenges. In the following, we motivate how the two concepts are useful for Rust verification, individually as well as in combination.

Type Invariants. Certain data types impose specific conditions on the validity of their values extending beyond the expressivity of Rust's type system. Operations defined for such types often rely on these conditions in their implementation. Hence, verifying the correctness of such operations necessitates a mechanism to specify and enforce adherence to the validity conditions. This is facilitated through the definition of a *type invariant*: a logical predicate attached to a data type. Verification enforces that a type invariant is preserved by all operations, thereby ensuring it holds for all values of the type. For example, a list type optimized for fast lookups might have a type invariant specifying that the stored elements are sorted. An operation such as inserting an element into the list, must guarantee that the list remains sorted.

This thesis presents a novel design for type invariants that we implemented in Creusot. While similar features have been implemented in other Rust verifiers [Ast+19; Leh+23; Gäh+23], support in Creusot necessitates a unique design to maintain soundness in the interaction with its distinguishing prophecies.

Ghost Code. Verification often benefits from additional context that is irrelevant during execution. However, augmenting a program's implementation with assertions or variables to provide such context would mean imposing a runtime overhead. *Ghost code*, auxiliary code added to a program solely for the purpose of verification, solves this issue. It upholds a characteristic *erasure property* stating that it can be removed from a program without changing its behavior. Because ghost code is not executed, it permits additional operations that only have a logical interpretation. These enable specifying higher-level properties that do not map directly onto the low-level details of the implementation. For example, ghost code is commonly used to create a ghost copy, or *snapshot*, of a value at the start of a function. When the value is subsequently modified, the copy allows relating the original and updated values at the end of the function. The usage of ghost code ensures that the creation of snapshots does not incur performance costs at runtime.

In this thesis, we analyze the soundness of an existing implementation of ghost code in Creusot. The analysis revealed several deficiencies, which we corrected by extending the existing implementation.

Type Invariants and Ghost Code Combined. While type invariants and ghost code are both useful individually, the true potential lies in their combination. A data structure may have an associated abstract state that is not directly constructible from its concrete data but connected through a logical relation. In such cases, the data structure can explicitly store the abstract state in a ghost field and enforce its consistency with the concrete state using a type invariant. For instance, consider a data structure that implements a state machine and is represented logically using a predicate specifying the transition relation. Traditionally, specifying properties about the state machine's history requires existentially quantifying over possible histories using the transition relation, as the runtime representation only stores the current state. Ghost code and type invariants offer a more elegant alternative: Operations store additional historical information in a ghost field and a type invariant enforces that historical and current states adhere to the transition relation. This pattern has shown to be useful, for example, to verify Rust's iterators [DJ23].

In summary, type invariants and ghost code are indispensable tools in the Rust verification toolkit. By effectively leveraging these concepts, users can specify and verify larger, more complex programs.

1.2 Thesis Overview

As a preliminary, Chapter 2 provides background information on Rust, deductive program verification, and Creusot. The thesis covers two main topics: type invariants and ghost code, which are presented in Chapter 3 and Chapter 4, respectively. After having been considered independently, the two concepts are evaluated in combination in Chapter 5. We discuss related work in Chapter 6 and draw comparisons with our work. Finally, in Chapter 7, we summarize our contributions and outline potential extensions.

2 Background

Before further discussing type invariants and ghost code, we give an overview of the Rust programming language and automated verification with Creusot.

2.1 Rust

Rust [MK14] is a compiled, statically typed systems programming language. It draws inspiration from both imperative and functional languages.

Basic Syntax. Rust adopts a C-style syntax, with code blocks enclosed in braces and statements terminated by semicolons. By default, variables in Rust are immutable, preventing their modification unless they are explicitly declared mutable using the `mut` keyword. Rust's control flow mechanisms, such as `if` expressions and `while` loops, look similar to those in other imperative languages. Functions are introduced with the `fn` keyword, and their return types are delineated with `->`. While the `return` keyword can explicitly specify a return value, functions implicitly return the value of their final expression. Rust provides various primitive types, including `u8` for unsigned 8-bit integers, `i32` for signed 32-bit integers, and `bool` for booleans. Beyond these primitives, the language supports custom algebraic data types via the `struct` and `enum` keywords, representing record and sum types, respectively. The `match` expression facilitates pattern matching, allowing values of algebraic types to be deconstructed by handling each variant. To attach behavior to types, methods can be defined inside of `impl`-blocks, making them callable using the typical dot notation. Within method definitions, the first parameter, `self`, refers to the value on which the method is invoked.

Figure 2.1 shows an example defining the enum `List` with two variants `Nil` and `Cons`. The `sum` method, defined only for lists of `i32` integers, computes the sum of their elements. Through iterative pattern matching, the function deconstructs the given list `self`, updating the mutable local variables `list` and `res`. Finally, when `list` matches `Nil`, the aggregated sum `res` is returned.

Ownership Types. Rust is distinguished by its language-based approach to memory safety [Jun+21], central to which is its *ownership type system* [CPN98]. In Rust, types are indicative of ownership. Specifically, possessing a value of a certain type implies an exclusive ownership of the associated data. This concept is influenced by *substructural type systems*, which, unlike traditional type systems, place constraints on how often a variable may be accessed. In Rust's context, while variables can be accessed or read

```rust
enum List<T> {
    Nil,
    // recursive type definitions require indirection (here: Box)
    Cons(T, Box<List<T>>),
}

impl List<i32> {
    fn sum(self) -> i32 {
        let mut list = self;
        let mut res = 0;

        loop { // loop expressions evaluate to break value
            match list {
                List::Cons(x, xs) => {
                    res += x;
                    list = *xs; // access inner value of box
                }
                List::Nil => break res,
            }
        }
    }
}
```

Figure 2.1: Example showcasing basic Rust syntax.

multiple times, the exclusive ownership of a variable can be transferred only once. Thus, function arguments are passed by value (i.e., shallow-copied into the callee), resulting in the transfer, or *move*, of ownership to the callee. After having been moved, the argument is no longer accessible to the caller. This system ensures a unique data owner at any point, mitigating concurrency issues like data races due to shared state between threads. Some types, such as primitive integer types and booleans, are exempt from the ownership rules. These types have *copy semantics* instead of move semantics, allowing them to be duplicated through shallow copies without transferring ownership. Rust also allows marking custom types as having copy semantics. Importantly, this ownership paradigm allows the compiler to statically determine when memory can be safely deallocated, eliminating the need for garbage collection. If a variable has not been moved and goes out of scope, Rust automatically *drops* it, invoking destructors and releasing resources.

Figure 2.2 illustrates these concepts. It defines two functions `print_str` and `print_i32` taking arguments of types `String` and `i32`, respectively. The `String` type represents a heap-allocated string and has move semantics, as a shallow copy would reference the same backing heap memory. In the function `example`, we first allocate a new string `s` using the `to_string` method. Then, we pass `s` to `print_str`, transferring its ownership. Because `s` has now been moved, another call to `print_str` would trigger an error during compilation. This ensures memory safety, as transferring ownership also transfers the capability to deallocate the string. This means, that `s` is deallocated at the end of the first invocation of `print_str`, making any subsequent use illegal. The example also demonstrates copy semantics using the primitive `i32` type. After the first call to `print_i32`, the variable `x` remains usable because its value is duplicable.

```
1  fn print_str(s: String) { println!("{s}"); }
2  fn print_i32(x: i32) { println!("{x}"); }
3
4  fn example() {
5      let s = "Rust rocks".to_string();
6      print_str(s); // moves s
7      // print_str(s); ERROR: s has been moved
8
9      let x = 42;
10     print_i32(x); // copies x
11     print_i32(x);
12 }
```

Figure 2.2: Example demonstrating the concept of ownership in Rust.

Borrowing. In addition to Rust's ownership model, the language introduces the concept of *borrowing*, granting temporary access to a value without transferring its

ownership. Borrowing a variable creates a *reference*, of which Rust distinguishes two types: *shared references*, denoted by `&T`, and *mutable references*, denoted by `&mut T`. Unlike some other languages, Rust treats references as first-class types, allowing arbitrary composition, such as having a reference to another reference. References adhere to the edict "aliasing XOR mutation". This means, that shared references permit duplication, creating multiple aliases, but are read-only, ensuring the underlying value remains unchanged. Conversely, mutable references grant exclusive, mutable access to the value they point to, but are unique and cannot be duplicated. In other words, shared references have copy semantics while mutable references have move semantics. Every reference in Rust has an associated *lifetime*, specifying the duration a reference remains valid. While the compiler can infer lifetimes automatically in most cases, users can explicitly specify them using the syntax `&'a T` or `&'a mut T`. Lifetimes help the compiler reason about the relationships between references and the values they point to. During the lifetime of a reference, the original, borrowed variable is *blocked*, meaning it cannot be used or borrowed further. Only once Rust determines that a reference went out of scope, its lifetime ends and the original variable can be used again. The borrowing mechanism ensures safety by enforcing a pivotal rule: for any given value, either multiple shared references or a single mutable reference may exist simultaneously. This prevents mutable shared state, a common pitfall in concurrent programs. Moreover, Rust protects variables from being dropped while borrowed, safeguarding against the peril of dangling references. Figure 2.3 illustrates Rust's borrowing system.

Traits. Rust's trait system is a mechanism for abstracting over functionality shared by several types. It draws inspiration from type classes in languages like Haskell but has its own unique characteristics. A *trait* encapsulates a set of function signatures that each implementing type must support. Consequently, an *implementation* of a trait for a specific type must provide a definition for each function specified by the trait. The special type `Self` may be used within a trait definition to refer generically to the implementing type. A type is said to *implement* a trait if an appropriate implementation exists. Traits can be used to *bound* generic type parameters in polymorphic functions, restricting instantiations to types implementing specific traits. This enables a form of ad hoc polymorphism, where different implementations can influence the behavior of a function based on the instantiation of the type parameter. So-called *marker traits* are special traits used to signify certain properties of types. While not necessarily defining any methods, they rather act as a kind of capability attached to a type. Prominently, the `Copy` trait signifies to the Rust compiler that a type should have copy semantics. The example shown in Figure 2.4 demonstrates these concepts.

2.2 Deductive Verification

Deductive program verification seeks to formally establish that all possible executions of a program adhere to a given specification.

```rust
fn print_str_ref(s: &String) { println!("{s}"); }
fn reverse_str(s: &mut String) { s.reverse(); }

fn example() {
    let mut s = "Rust rocks".to_string();

    let r1 = &s; // borrow as shared reference
    // print_str(s); ERROR: s is borrowed
    // let r2 = &mut s; ERROR: s is borrowed
    print_str_ref(r1); // copy r1
    print_str_ref(r1);
    // lifetime of r1 expires, s is no longer borrowed

    let r2 = &mut s; // borrow as mutable reference
    // print_str(s); ERROR: s is borrowed
    // let r3 = &s; ERROR: s is borrowed
    reverse_str(r2); // move r2
    // lifetime of r2 expires, s is no longer borrowed

    print_str(s); // ok
}
```

Figure 2.3: Example illustrating borrowing in Rust.

```rust
trait Add {
    // Self refers to the implementing type
    fn add(self, other: Self) -> Self;
}

impl Add for i32 {
    fn add(self, other: i32) -> i32 {
        self + other
    }
}

// T is bounded to types implementing Add and Copy
fn double<T: Add + Copy>(x: T) -> T {
    // we can use x twice because T: Copy
    x.add(x)
}
```

Figure 2.4: Example of defining and using traits in Rust.

Hoare Logic. *Hoare logic* [Hoa69] provides a formal system to prove properties about the behavior of imperative programs. It provides a set of rules to reason about *Hoare triples*, the formulas of Hoare logic combining a program with its specification. The triple $\{P\}\, s\, \{Q\}$ states that if the *precondition* P holds before executing the program s then the *postcondition* Q holds afterward. A triple is *valid* if it can be derived from a set of axioms and inference rules. A valid triple implies *partial correctness*: The execution of s does not get stuck, and starting in a state where P is true, if s terminates, then Q is true at the end. Hoare logic gives rise to the *weakest precondition* (WP) calculus [Dij+76]. Given a program and a postcondition, it computes a precondition that makes the corresponding Hoare triple valid by applying the Hoare rules backward. Hence, for all programs s and postconditions Q, the triple $\{\mathrm{wp}(s, Q)\}\, s\, \{Q\}$ is valid. The computed precondition is weakest, in the sense that it is subsumed by every other precondition making the triple valid.

Automated Verification. Leveraging the framework provided by Hoare logic, proving program correctness can be translated into a computational problem automatable to a certain degree. Given a program and its specification, a set of *verification conditions* (VCs) can be computed based on the structure of the program. These logical predicates capture exactly the conditions for partial correctness of a program. For example, a Hoare-style contract $\{P\}\, s\, \{Q\}$ results in a VC essentially stating $P \implies \mathrm{wp}(s, Q)$. To avoid having to find fixpoints when computing the WP of a loop, users explicitly annotate loops with *loop invariants*. The VC includes additional conditions ensuring that every loop preserves its invariant and the invariant implies the loop's postcondition. Finally, automated proving systems, such as satisfiability module theory (SMT) solvers, can be used to prove or disprove the generated formulas.

Why3. Why3 [FP13] is a platform for program verification, offering a unified interface to several automated and interactive theorem provers. These backends include SMT solvers like CVC5 [Bar+22] or Z3 [DB08], and auto-provers such as Alt-Ergo [Con+18]. At its core is the functional language WhyML which is used both for writing programs as well as specifying them using Hoare-style pre- and postconditions. Based on an annotated WhyML program, Why3 generates a set of verification conditions that imply the program's correctness. The interactive Why3 IDE [DMM18] helps with proving the generated VCs. It shows each VC as a *goal* along with its corresponding *context*. The context contains axioms and hypotheses coming from preconditions, proved assertions, or contracts of other functions. Each goal can be either discharged by one of the backends or further transformed, producing one or more subgoals. Invoking a backend solver may either return a result, signifying a proof or disproof of the goal, or timeout. Prover timeouts are typically caused by missing hypotheses or a too complex formulation of the goal. While the former requires adjusting the specifications, the latter can also be tackled by applying manual transformations. Such transformations include, for example, splitting conjunctions, rewriting terms, or instantiating existential quantifiers.

2.3 Rust Verification with Creusot

Toolchain. Creusot [DJM22] is a deductive verifier for Rust programs built on top of the Why3 platform. Figure 2.5 shows a high-level overview of the verification toolchain. Before running Creusot, the user annotates their Rust program with specifications. Creusot invokes the Rust compiler `rustc` to parse and analyze the source program, extracting the compiler-internal mid-level intermediate representation[1] (MIR). Next, Creusot translates the MIR and the given specifications to the MLCFG dialect of WhyML, the functional language at the core of Why3. Based on the generated WhyML program, Why3 computes verification conditions to be discharged by its backends.

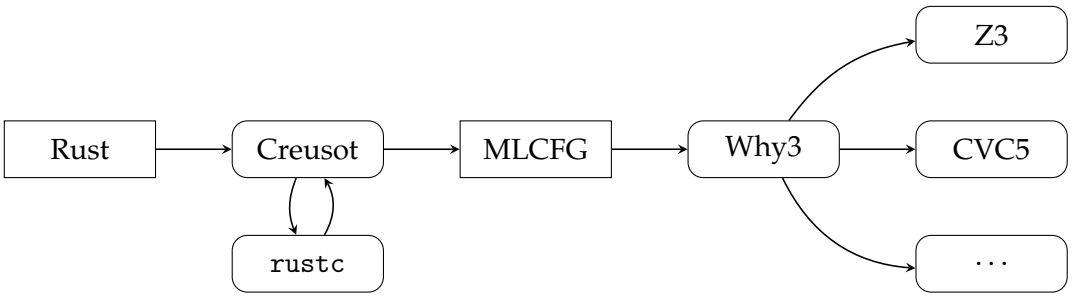


Figure 2.5: Overview of Creusot's verification toolchain.

Specifying Rust Functions. Specifying Rust programs entails annotating functions with Hoare-style contracts. Therefore, Creusot provides the `#[requires(...)]` and `#[ensures(...)]` attributes, specifying a function's pre- and postconditions, respectively. Both have access to the function's *prestate*, i.e., the values of the function arguments when called. The postcondition can additionally refer to the special symbol `result` representing the function's return value. In addition to the properties specified as postconditions, Creusot verifies the absence of panics and overflows for every function, including those without explicit annotations.

Loop Invariants. Loop invariants are properties that hold before entering a loop and are preserved in each loop iteration. Stating loop invariants is a prerequisite to computing VCs. Hence, Creusot requires users to explicitly annotate loops with the `#[invariant(...)]` attribute. It can be attached to all looping constructs, such as while- and for-loops. Why3 generates an initialization and a preservation goal for each loop invariant.

Pearlite. The logical predicates inside of the shown attributes are written in Pearlite, Creusot's specification language. The syntax of Pearlite is a subset of the syntax of

[1] MIR represents Rust code as a control-flow graph. Unlike lower-level representations, such as LLVM IR, MIR fully preserves type information.

Rust expressions extended with additional logical constructs. These include the implication operator `==>`, the universal quantifier `forall<x: T>`, and the existential quantifier `exists<x: T>`. The quantifiers specify the type `T` of the bound variable. Pearlite can use both Rust types, which are translated into a corresponding WhyML type by Creusot, as well as logical types defined in Why3's standard library. Such types are, for example, `Int` for unbounded integers, or `Seq<T>` representing generic, mathematical sequences. Unlike Rust code, Pearlite is not compiled to MIR by `rustc` but instead extracted at the AST level and directly translated to WhyML. The `proof_assert!` macro allows verifying interim properties by embedding Pearlite expressions in Rust code. Unlike Rust's `assert!` macro which checks a boolean expression at runtime, `proof_assert!` checks an expression at verification time.

Auxiliary Functions. Auxiliary logic functions allow factoring out common parts of specifications. While looking similar to regular Rust functions superficially, their bodies are defined using Pearlite instead of Rust. To distinguish them from Rust functions, they are marked with the `#[logic]` or `#[predicate]` attributes. Predicate functions must return a boolean value, whereas logic functions can return values of any type. Using non-Rust syntax in auxiliary functions requires enclosing the Pearlite expression in the `pearlite!` macro which desugars Pearlite constructs to valid Rust syntax.

Verified Sorting Routine. In the following, we demonstrate how to verify a simple sorting routine, using the example of `gnome_sort` shown in Figure 2.6. The function sorts a given slice of integers `v` in place, repeatedly swapping adjacent elements if they are not yet in the right order. The `ensures` attribute specifies the function's postcondition, saying that the elements of `v` are sorted after the function returns. Because both, pre- and postconditions, relate to the prestate of a function, we must use the *final value operator* `^` to refer to the values pointed to by `v` after having been sorted. This operator can only be used in Pearlite, works only on mutable references, and can be understood as a *prophetic* version of the normal dereferencing operator `*`. The example also uses the *model operator* `@`, another special Pearlite construct, which is used to refer to the logical representation of a value. In the example, it lifts the given Rust slice to a logical sequence of type `Seq<i32>`. Proving the postcondition requires a corresponding loop invariant, stating that the first `i` elements in the slice are sorted. The annotations in `gnome_sort` use an auxiliary predicate `sorted`, defining what it means for a given sequence to be sorted. Since its definition contains special Pearlite syntax, namely a universal quantifier and an implication, it is wrapped in the `pearlite!` macro which desugars Pearlite syntax to valid Rust. A more complete specification for the `gnome_sort` function would include an additional postcondition ensuring that no elements are added or removed from the slice.

After having annotated the program with specifications, the user invokes Creusot by running the command `cargo creusot`. This works because Creusot integrates with Rust's build and package management tool Cargo. Running the command produces a

```
#[predicate]
fn sorted(s: Seq<i32>) -> bool {
    pearlite! {
        forall<i: Int, j: Int> 0 <= i && i < j && j < s.len()
            ==> s[i] <= s[j]
    }
}

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

Figure 2.6: Example of verifiying a sorting function with Creusot.

file with the MLCFG representation of the program that can be opened in the Why3 IDE. Figure 2.7 shows the user interface of the Why3 IDE when opened on the MLCFG file generated for the `gnome_sort` function. On the left-hand side, there is a tree of all goals and the right-hand side shows the proof context of the currently selected goal. Taking a closer look at the tree, the top-level goal `gnome_sort'vc` has five children: four failed proof attempts and one transformation. Each of these proof attempts represents the invocation of a different backend. However, none of the backends succeeded in proving the goal and were interrupted by Why3 after a timeout of one second. Consequently, the user applied the `split_vc` transformation to the goal, which splits the top-level goal into multiple subgoals. Rerunning the backends on each of the generated subgoals successfully verified the function, as indicated by the green checkmarks.

Lemma Functions. A common proof strategy is to first prove several simpler lemmas which can subsequently be used as additional hypotheses for more complex properties. To use this strategy in Creusot, one can either use the `proof_assert!` macro or *lemma functions*: auxiliary logic functions with contract annotations. Such a function yields a lemma stating that the function's preconditions imply its postconditions, which, when proven, can be used as a hypothesis in other functions. Because a lemma function typically has no return value, its body is irrelevant to the definition of the lemma. However, the body of a lemma function may include explicit instantiations of other lemmas, guiding provers or even enabling a primitive form of induction. This is exemplified in Figure 2.8 using the lemma `sum_odd_sqr` stating

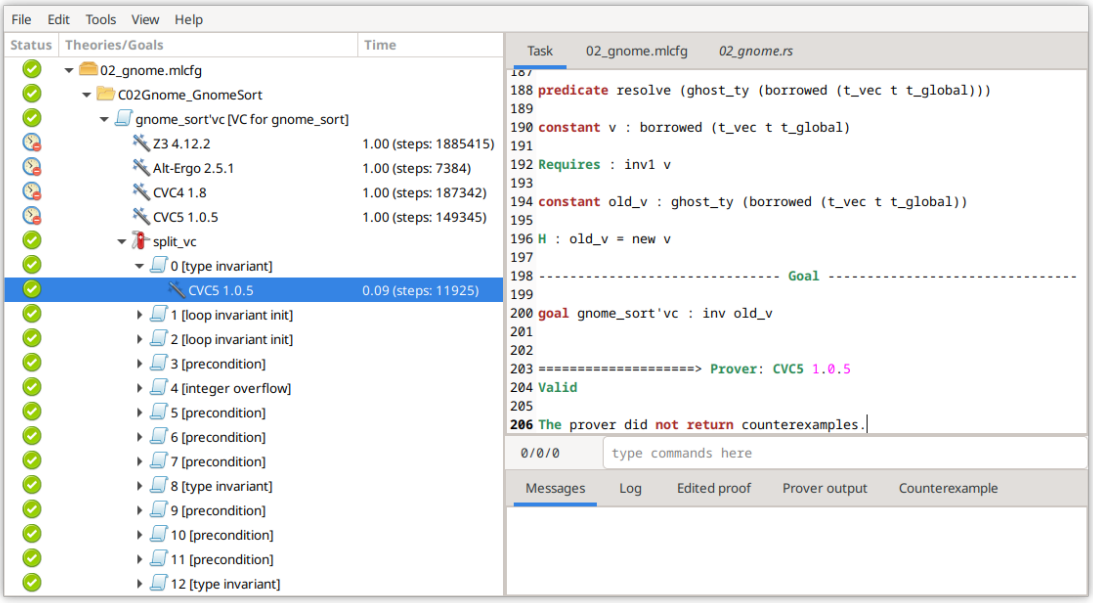


Figure 2.7: The gnome sort function opened in Why3 IDE.

$\forall x.x \geq 0 \implies \texttt{sum\_odd}(x) = x^2$. Its body contains a recursive call corresponding to the induction hypothesis $\forall x.x > 0 \implies \texttt{sum\_odd}(x-1) = (x-1)^2$. Because logic functions are required to terminate, we must provide a `#[variant(...)]` annotation which specifies a strictly decreasing quantity.

```
#[logic]
#[requires(x >= 0)]
#[ensures(sum_odd(x) == x*x)]
#[variant(x)]
fn sum_odd_sqr(x: Int) {
    if x > 0 {
        sum_odd_sqr(x - 1)
    }
}
```

Figure 2.8: Example of a lemma function.

Prophecies. What allows Creusot to functionally reason about imperative Rust programs, is its encoding of mutable references using *prophecies*. Creusot represents mutable references as a pair of their current value and a nondeterministic prophecy equal to the last value pointed to by the reference before it is dropped. We illustrate the basic concept with the example in Figure 2.9. We create a mutable reference `r` by borrow-

ing `x`, and write the value 3 to it. Traditional verification would require knowing how `x` and `r` are connected to also update the value of `x` when writing to `r`. While this would be rather easy in the case of our example, such reasoning generally is hard, especially when taking into account nested references and interprocedural effects. Creusot solves this issue using the prophecy `^r`, linking the borrowed `x` and the reference `r`. In the example, the comments next to the Rust statements indicate their functional translation. Borrowing `x` involves three steps:

1. Create the prophecy `^r`, whose actual value is unknown at this point and is thus assigned the nondeterministic value `any`,
2. Create the reference `r` as the pair of its current value `x` and final value `^r`,
3. Set the value of the borrowed `x` to `^r`.

This works thanks to Rust's borrowing semantics. As the borrowed variable `x` is blocked until `r` is dropped, its concrete value can remain prophetic up to that point. Furthermore, since mutable references cannot alias, the final value for each borrowed variable is uniquely determined.

Writing to the reference means updating its current value and leaving the prophecy untouched. At the end of the lifetime of `r` (i.e., once it is no longer used and thus has been modified for the last time), we learn that its final value `^r` is 3. Creusot automatically determines this point, at which it *resolves* `r`, assuming an axiom stating that the final value `^r` and the current value `*r` are equal. After resolution, the axiom becomes available as a hypothesis and we can prove the assertion using the equalities $\texttt{x} = \texttt{\^{}r} = *\texttt{r} = 3$.

```
1 let mut x = 5; // x <- 5
2 let r = &mut x; // ^r <- any; r <- (x, ^r); x <- ^r
3 *r = 3; // r <- (3, ^r)
4 // Creusot resolves r: assume ^r = *r
5 proof_assert! { x == 3 }; // provable by assumption
```

Figure 2.9: Example of the prophetic encoding of mutable references.

The expressivity of this approach is demonstrated in Figure 2.10. The function `index_mut` takes a mutable slice `s` (i.e., a reference to a contiguous sequence of elements) and an index `i`, returning a reference to the element at that index. Postcondition (2) accordingly states that the current value of the returned reference `result` equals the respective slice element. By writing to the returned reference, the caller can modify the input slice `s`. Consequently, specifying the effect of `index_mut` on `s` requires knowing how the caller uses the returned reference. Thanks to prophecies, this is rather straightforward: The final value at index `i` in `s` is the final value of `result` (cf. postcondition (3)). While the actual value of the prophecy `^result` is never learned in `index_mut`, the postcondition communicates the link between the two prophecies to the caller.

```
#[requires(i@ < (*s)@.len())] // i is a valid index
#[ensures((^s)@.len() == (*s)@.len())] // length of s does not change
#[ensures(*result == (*s)[i@])] // (2)
#[ensures((^s)[i@] == ^result)] // (3)
#[ensures(
    forall<j: Int> 0 <= j && j != i@ && j < (*s)@.len()
        ==> (^s)[j] == (*s)[j]
)] // other elements remain unchanged
fn index_mut<T>(s: &mut [T], i: usize) -> &mut T {
    &mut s[i]
}
```

Figure 2.10: Example of using prophecies to specify `index_mut`.

3 Type Invariants

This chapter focuses on the design and implementation of type invariants in Creusot. First, Section 3.1 explains the main design challenges and the motivation behind our approach. As a solution to these challenges, we present *prophetic invariants* in Section 3.2. For this approach to be sound, several conditions must be met. In Sections 3.3 and 3.4, we focus on these conditions and explain how they are enforced.

3.1 Design Challenges

This section explains the main considerations and design questions that we needed to address to support type invariants in Creusot. Those can be roughly summarized by the following questions:

(D1) How to attach invariants to types in a composable manner?

(D2) When can invariants be temporarily broken and when must they hold?

(D3) How should type invariants interact with mutable references?

Question (D1) entails designing a mechanism to determine the invariant of a type based both on the invariants of its parts as well as user-defined invariants. For type invariants to be practical, there must exist a mechanism for temporarily suspending them. A core challenge thus lies in the design of the rules governing this mechanism, as characterized by question (D2). Furthermore, question (D3) calls for scrutiny in designing a system compatible with Creusot's prophecies. We elaborate on questions (D2) and (D3) in the following two subsections.

3.1.1 Open and Closed Invariants

Concerning question (D2), we distinguish *open* and *closed* invariants. While a type invariant is open, it can be temporarily broken and, conversely, it must hold when closed. Closing an open invariant thus requires proving that it is still satisfied by the potentially modified value. In general, functions should be able to open a value's type invariant internally as long as it is closed before the value can be observed by another function. This leads to function calls as a natural boundary for enforcing type invariants.

Figure 3.1 demonstrates this guiding design principle. The type `SumTo10` is a pair of integers and its invariant requires that the sum of both values equals 10. The function `mingle` takes an argument `s` of type `SumTo10` and returns a value of the same type. Using

```
// Invariant: a + b == 10 (definition omitted in the example)
struct SumTo10 { a: i32, b: i32 }

fn mingle(mut s: SumTo10) -> SumTo10 {
    proof_assert! { s.a + s.b == 10 };
    // OPEN the invariant
    s.a -= 1; // breaks the invariant
    s.b += 1; // restores the invariant
    // CLOSE the invariant
    s
}
```

Figure 3.1: Example of temporarily opening a type invariant.

the type invariant of `s`, the assertion `s.a + s.b == 10` is provable at the start of the function. Next, the invariant is implicitly opened, allowing to temporarily break it by decrementing `s.a`. This is fine because the invariant is closed before finally returning the modified value.

```
// Owned Values
fn zeroize(mut s: SumTo10) {
    s.a = 0; s.b = 0;
    // invariant not closed
}

// Mutable References
fn swap(s: &mut SumTo10) {
    let tmp = s.a;
    *s.a = s.b;
    *s.b = s.a
    // must close invariant
}

// Shared References
fn swapped(s: &SumTo10)
    -> SumTo10 {
    // create a copy of s
    let mut s2 = SumTo10 {
        a: 0, b: 0 };
    s2.a = s.a; s2.b = s.b;

    // swap a and b in place
    let r = &mut s2;
    swap(r);

    s2
}
```

Figure 3.2: Example of type invariants across function boundaries.

To further illustrate when invariants must be restored, we consider the example shown in Figure 3.2. Each of the three functions `zeroize`, `swap`, and `swapped` demonstrates one of the tree principle ways how values can be passed to a function.

Owned Values. The argument of the `zeroize` function is moved; its ownership is transferred from the caller. The value is dropped at the end of the function and cannot

be observed by any other function. Therefore, it is permitted to break the invariant of `s` by setting both fields to zero. If `zeroize` returned `s` or passed `s` to another function, the invariant would have to be before that.

Mutable References. The function `swap` takes its argument `s` as a mutable reference and swaps the values of its two fields in place. Rust's type system guarantees that as long as the original value is borrowed by `swap`, it cannot be observed by other functions. This makes it legal to temporarily break the invariant during the three-step exchange. Nonetheless, the invariant must be restored before the borrow ends and `swap`'s caller regains access.

Shared References. The function `swapped` takes a shared reference to a `SumTo10` value and returns a copy where the values of `a` and `b` are swapped. Because shared references can alias, other functions may observe the value but the read-only characteristic makes it impossible to break the invariant. The simple and obvious implementation for `swapped` would initialize a new `SumTo10` value with the values from `s` in swapped order. However, our implementation deliberately constructs the result in a roundabout way to make the behavior of the involved type invariants explicit. First, a local copy `s2` of `s` is created by first constructing a zero-initialized `SumTo10` value and then setting each field at a time. This demonstrates that it is allowed to construct values not satisfying their type invariant. Only after setting the second field `s2.b`, the invariant of `s2` becomes provable, using the invariant of the argument `s` as a hypothesis. Next, `s2` is mutably borrowed to obtain the mutable reference `r` and pass it to the function `swap` which swaps the two fields. Since the constructed value crosses a function boundary, its invariant must be closed at this point. Finally, the modified value `s2` is returned, which again requires us to show its invariant.

3.1.2 Interaction of Invariants and Borrowing

To answer question (D3) and to better understand how to ensure that invariants are closed at the end of borrows, we take a closer look at the call to `swap` in `swapped`. In Creusot's translation to MLCFG, borrowing `s2` as `r` puts the newly created reference's final value `^r` into `s2` (cf. §2.3). Consequently, to prove the invariant of `s2` after the call to `swap`, we need a hypothesis about `^r`. One way to obtain such a hypothesis would be to have the callee's postcondition guarantee that the final value of the argument `s` satisfies its invariant. This way, the invariant of the final value becomes available as a hypothesis in `swapped` after the call. So, in this example, we can determine a postcondition for `swap` ensuring that the reference's invariant is maintained in `swapped`. However, this approach does not work in general; there is not always an appropriate postcondition for functions that may take a mutable reference as an argument.

To clarify the issue we consider the function `call_generic`, shown in in Figure 3.3. This function mutably borrows `s` of type `SumTo10` and passes it to the generic function

`generic`, which takes an argument of generic type `T`. Since calling `generic` moves `r`, it is not resolved in the caller and the actual value of `^r` remains unknown. Thus, closing the invariant of `s` before it is returned would require the postcondition of `generic` to state that the invariant of its argument's final value holds. However, from the perspective of `generic`, the argument has generic type `T`, which does not have a notion of final values. Consequently, it is not possible to determine the necessary postcondition solely based on the callee as it depends on the instantiation of `T` in the caller.

```
fn generic<T>(x: T) {}

fn call_generic() -> SumTo10 {
    let mut s = SumTo10 { a: 3, b: 7 };
    let r = &mut s; // now s equals ^r
    generic(r); // we learn nothing about ^r
    s // how to prove the invariant of s?
}
```

Figure 3.3: Example demonstrating the issue with deriving postconditions for closing invariants of mutable references.

To summarize, the obvious approach is incompatible with Creusot's prophecies. It is not generally possible to encode the invariant preservation property of mutable references as a postcondition. To overcome this challenge, we developed *prophetic invariants*, a novel approach that is fully composable with Rust's type system.

3.2 Prophetic Invariants

After having outlined the main design challenges of type invariants, we present our approach and demonstrate how it overcomes these challenges. First, Section 3.2.1 shows how a user can define an invariant for a type and how type invariants affect Pearlite and Rust functions. In Section 3.2.2, we present *prophetic invariants*, our solution to the challenge discussed in 3.1.2. Finally, Section 3.2.3 explains the conditions under which this approach is sound. For the remainder of this chapter, we write "Creusot$^+$" to refer to the version of Creusot including our contributions concerning type invariants.

3.2.1 Basic Usage

Defining Invariants. We leverage Rust's trait system as a composable mechanism to attach an invariant to a type, addressing design question (D1). To attach an invariant to a type, a user implements the `Invariant` trait provided by Creusot$^+$, shown in Figure 3.4. It defines the `invariant` predicate that only exists on the logic level and uses Pearlite in its body. It takes a single argument `self` of the type implementing the trait. Figure 3.4

also exemplifies how to define the invariant of the type `SortedInts`, which simply states that the list's logical representation, denoted using the `@` sigil, is sorted.

```
trait Invariant {
    #[predicate]
    fn invariant(self) -> bool;
}

enum SortedInts {
    Nil,
    Cons(i32, Box<SortedInts>),
}

impl Invariant for SortedInts {
    #[predicate]
    fn invariant(self) -> bool {
        pearlite! {
            self@.sorted()
        }
    }
}
```

Figure 3.4: The definition of the `Invariant` trait and an example of defining a type invariant.

User Invariants and Structural Invariants. While we have seen how type invariants are defined, we have not yet explained how the invariant for a particular type is determined. In general, the invariant for a type `T` consists of the conjunction of two parts:

1. A *user invariant*, provided by the user through an implementation of the `Invariant` trait, and

2. A *structural invariant*, derived automatically based on the constituents of `T`.

The user invariant is determined through trait resolution, which, given a type and a trait, finds a suitable implementation. If no implementation exists, the user invariant defaults to the trivial invariant satisfied by all values. The structural invariant combines the invariants of the constituent types. As an example, we consider Rust's `Option<T>` type, which has two constructors, `None` and `Some(T)`. The user invariant is trivial because the `Invariant` trait is not implemented for `Option<T>`. The corresponding structural invariant is `true` for `None` values, otherwise, it equals the invariant of the inner type `T`. This allows us to conclude that the invariant of a value `x` holds when pattern matching on `Some(x)`. Structural invariants ensure composability (D1) and are discussed in detail

in Section 3.3. We henceforth write `inv(x)` to refer to the combined type invariant of a value `x`.

Type Invariants in Pearlite. In Pearlite, Creusot$^+$ enforces type invariants by guarding quantifiers with the bound value's type invariant. For example, the expression `forall<s: SumTo10> P(s)` says that the predicate `P(s)` is true for all values `s` if `s` has type `SumTo10` and the invariant of `s` holds. Consequently, the term `forall<s: SumTo10> s.a@ + s.b@ == 10` is always true and `exists<s: SumTo10> s.a@ + s.b@ != 10` is always false. Generally, for a type `T` and its invariant predicate `inv`, the term `forall<x: T> P(x)` is equivalent to `forall<x: T> inv(x) ==> P(x)`, and `exists<x: T> P(x)` is equivalent to `exists<x: T> inv(x) && P(x)`. This is achieved by synthesizing the guarding invariant predicates during Creusot$^+$'s translation of Pearlite into MLCFG.

Contract Elaboration. As explained in Section 3.1, type invariants are enforced on function boundaries. Therefore, type invariants of a function's arguments are treated as additional preconditions. Analogously, a type invariant of the return value corresponds to an extra postcondition. Figure 3.5 shows an example of how a contract is elaborated based on a function's signature. The predicate `inv` represents the respective type invariant. Creusot$^+$ synthesizes a precondition for each argument and a postcondition for the return value. These simple rules achieve the desired behavior described by question (D2). Inside functions, users are allowed to open the invariants of the function arguments and must close the invariant of the return value. Calling another function requires proving its precondition, meaning that the invariants of the function arguments must be closed. Besides regular Rust functions, Creusot$^+$ also elaborates the contracts of lemma functions. As the axiom generated from such a function universally quantifies over the function's arguments, synthesizing preconditions for those arguments is analogous to how quantifiers are guarded in Pearlite.

```
1 #[requires(inv(a))]      // <-┐
2 #[requires(inv(b))]      // <-┼- synthesized
3 #[ensures(inv(result))]  // <-┘
4 fn concat(a: SortedInts, b: SortedInts) -> SortedInts { todo!() }
```

Figure 3.5: Example demonstrating the effect of contract elaboration.

3.2.2 Prophetic Invariants

In the following, we present *prophetic invariants*, our approach to solving the challenges presented in 3.1.2, and discuss its consequences. To briefly recapitulate, the objective is to maintain the type invariants of mutable references when passed to other functions. We previously explained that it is not generally possible to make the contract of the called function guarantee invariant preservation. Prophetic invariants solve this issue by

assuming the invariants of prophecies in the caller. This eliminates the need to propagate these facts as postconditions. Furthermore, this approach complements Creusot's fundamental prophetic interpretation of borrowing. The prophecy assigned to borrowed variables now includes a guarantee that whatever value it is resolved to satisfies its invariant.

Figure 3.6 shows an example of a mutable reference passed as a function argument, similar to the one in Figure 3.3. In `call_generic`, the value `s` satisfying its invariant is borrowed, obtaining the mutable reference `r` which is passed to the function `generic`. The elaborated contract of `generic` includes a precondition for the invariant of `r`. As Creusot understands a mutable reference as a pair of its current and final value, the invariant of `r` structurally consists of the invariants of its parts `*r` and `^r`. The invariant of the current value follows from the invariant of `s` and the invariant of the final value is prophetically assumed when borrowing.

After the call, Creusot$^+$ must prove that the invariant of `s` still holds. In Section 3.1.2, we concluded that it is not generally possible to automatically infer a contract for `generic` that ensures the invariant of its argument's final value. However, with the encoding proposed in this section, proving the invariant does not require any hypotheses from the call to `generic`. Because `s` and `^r` are logically equal and Creusot$^+$ prophetically assume the invariant for `^r`, the assertion is provable.

```
1  fn generic<T>(x: T) {}
2
3  fn call_generic() -> SumTo10 {
4      let mut s = SumTo10 { a: 3, b: 7 };
5      let r = &mut s; // s <- ^r
6      // ASSUME inv(^r)
7      generic(r);
8      proof_assert! { s.a@ + s.b@ == 10 }; // provable!
9      s
10 }
```

Figure 3.6: Example illuminating how prophetic invariants solve the challenges concerning invariants of mutable references.

3.2.3 Soundness

Soundness of prophetic invariants requires that at no point it is possible to show that the invariant of a prophecy does not hold. If it were possible to produce such a proof, it would contradict the assumption made when creating the prophecy. As contradictory assumptions allow proving arbitrary terms, it would make the verification unsound. Thanks to Rust's guarantees for mutable borrows, there are only two conditions we must consider for every prophecy to uphold soundness:

(S1) The invariant is not provably false when creating the prophecy, and

(S2) The invariant holds when resolving the prophecy.

The first condition (S1) forbids invariants that are trivially false for all values. It is enforced by the *inhabitation law*, to which implementors of the `Invariant` trait must adhere. Laws are special trait methods that, similar to lemma functions, generate a proof obligation for every implementation. As shown in Figure 3.7, the `Invariant` trait defines a law called `is_inhabited` requiring each implementation to prove there is at least one value satisfying the invariant. Consequently, Creusot$^+$ never assumes trivially false invariants and fulfills condition (S1).

```
#[law]
#[ensures(exists<x: Self> x.invariant())]
fn is_inhabited();
```

Figure 3.7: The Inhabitation Law.

Condition (S2) ensures that the prophetically assumed invariant actually holds once a borrow's final value is learned. The final value is determined through resolution, assuming equality of the borrow's current and final value after its last use (cf. §2.3). At this point, Creusot$^+$ generates a proof obligation that the current value fulfills the invariant. Crucially, this must be shown *before* equating current and final value as the assertion would otherwise be a tautology.

To show that these conditions are sufficient, we argue that if condition (S1) is fulfilled, the invariant holds at every point up until the prophecy is resolved. A prophecy is created by mutably borrowing a variable. Rust ensures that the borrowed variable remains inaccessible as long as the reference exists. Since Rust code may only refer to the prophecy through the borrowed variable, the prophecy's value remains unchanged until it is resolved. Therefore, it is sufficient to only check the invariant at prophecy creation (S1) and resolution (S2).

As a consequence of condition (S2), correctly resolving every prophecy is critical for soundness. Before the introduction of type invariants to Creusot, omitting to resolve a prophecy only affected completeness, making some assertions not provable. Now, missing resolutions mean unjustified prophetic invariants. In particular, there are two challenges to consider:

1. If a mutable reference is nested inside a value of another type, the invariant of the prophecy must still be proven, no matter the invariant of the outer type, and

2. Every variable must be resolved at the correct location.

These two challenges are discussed in detail in the following Sections 3.3 and 3.4.

3.3 Structural Invariants

So far, we have seen how type invariants are defined, how they are used, and how they interact with prophecies. In the following, we first argue why structural invariants are necessary for soundness and then explain how exactly they are derived for a specific type.

3.3.1 Preventing Hidden Invariants

In Section 3.2.3, we stated that soundness requires that every mutable reference is resolved, including those nested in other values. Without structural invariants, resolving a value of type `Option<&mut SumTo10>` would not require proving the invariant of the contained reference. Therefore, the assumption that the final value's invariant holds would have been unjustified. This is illustrated in Figure 3.8. The example assumes that the type `Option<T>` does not have a structural invariant. The mutable reference `r` is used to break the invariant of the borrowed `SumTo10` value `s`. Subsequently, `r` is moved into a newly created option `x` that is never used and thus immediately dropped. When `x` is resolved, its invariant must be proven but, since it is trivial, this does not entail proving the invariant of `r`. As a result, the false invariant of the updated value `s` is provable by assumption.

```
1 let mut s = SumTo10 { a: 3, b: 7 };
2 let r = &mut s;
3 *r.a = 0;
4 let x = Some(r);
5 // resolve x: prove inv(x), assume ^x == *x
6 proof_assert! { 0 + 7 == 10 }; // unsound!
```

Figure 3.8: Example demonstrating how hidden invariants can cause unsoundness.

In general, Creusot resolves not just mutable references but values of any type. For this, it utilizes a special trait called `Resolve` which for any type defines what it means to resolve a value of that type. By convention, the resolve operation structurally resolves every component of a value. This ensures that, for example, resolving a pair of mutable references means resolving each individual reference. We want the same behavior for type invariants; resolving a pair of mutable references must require proving the invariants of both contained references. While it may seem obvious to base type invariants for composite values on a similar trait-based mechanism, there are several disadvantages to consider:

1. An incorrectly defined trait implementation, omitting certain fields, would be unsound. Hence, we would have to trust users to correctly define structural invariants for their types.

2. There would be an additional burden to add a trait implementation for every custom type. While this could be partially alleviated with macros, there would still be an ergonomic overhead.

3. The `Resolve` trait uses specialization, an experimental Rust feature, to be able to provide implementations for any type. However, this feature comes with several limitations that would restrict how type invariants can be used.

Due to these factors, we chose to not rely on traits but derive correct structural invariants automatically.

3.3.2 Derivation of Structural Invariants

The following rules specify how Creusot$^+$ derives the invariant of a type τ for a simplified model of Rust's type system. The `uinv`, `sinv`, and `inv` predicates represent the user, structural, and combined invariant, respectively. The injection functions inj_i construct sum types and the projections proj_i destruct product types.

$$\text{Type } \tau := \texttt{bool} \mid \texttt{i32} \mid \tau \times \tau \mid \tau + \tau \mid \&\tau \mid \texttt{\&mut } \tau$$

$$\texttt{inv}_\tau(x) := \texttt{uinv}_\tau(x) \wedge \texttt{sinv}_\tau(x) \tag{3.1}$$

$$\texttt{sinv}_\tau(x) := \texttt{true} \qquad \text{if } \tau \text{ is primitive} \tag{3.2}$$

$$\texttt{sinv}_{\tau_1 \times \tau_2}(x) := \texttt{inv}_{\tau_1}(\text{proj}_1\ x) \wedge \texttt{inv}_{\tau_2}(\text{proj}_2\ x) \tag{3.3}$$

$$\texttt{sinv}_{\tau_1 + \tau_2}(\text{inj}_1\ x) := \texttt{inv}_{\tau_1}(x) \tag{3.4}$$

$$\texttt{sinv}_{\tau_1 + \tau_2}(\text{inj}_2\ x) := \texttt{inv}_{\tau_2}(x) \tag{3.5}$$

$$\texttt{sinv}_{\&\tau}(x) := \texttt{inv}_\tau(*x) \tag{3.6}$$

$$\texttt{sinv}_{\texttt{\&mut } \tau}(x) := \texttt{inv}_\tau(*x) \wedge \texttt{inv}_\tau(\hat{}x) \tag{3.7}$$

Except for rule 3.7, these rules should be unsurprising. In Creusot, mutable references can be understood as a pair of current value and final value. Analogous to rule 3.3, Creusot$^+$ consequently derives the structural invariant of a mutable reference from the invariants of its current and final value. Consequently, calling a function with a mutable reference as an argument requires proving the invariants of both its current and final value as a precondition. However, the prophetic invariant of the final value is usually easy to prove as it has been assumed when creating the reference.

Some types, including Rust's `Vec` type, internally store data with raw pointers and use unsafe code to modify the data. In unsafe code, memory safety is not automatically checked by the Rust compiler but is the responsibility of users. Types like `Vec` provide a safe interface around unsafe code, ensuring memory safety through careful implementation. While Creusot does not support unsafe code, it still works with types safely encapsulating unsafe code. Similar to how users must manually ascertain the safety of such types, users are also required to manually define their type invariants.

For example, the `Vec` type has a user invariant which states that the invariants of all contained elements hold. Raw pointers are considered primitive and thus have a trivial structural invariant according to rule 3.2.

3.3.3 Invariant Encoding

In Section 3.2, we used the generic symbol `inv` to refer to type invariants and explained its meaning for specific types in Section 3.3.2. Now, we take a closer look at how exactly the symbolic predicate and its interpretation are linked.

Creusot translates type declarations from Rust into WhyML. During this translation, Creusot$^+$ additionally generates an *unfolding axiom* that dictates how to unfold the `inv` symbol for that specific type. Figure 3.9 shows the declaration of the `inv` symbol and the generated unfolding axiom for the `SortedInts` type. The `inv` symbol is declared as an abstract predicate in the `Inv` module, which also declares an abstract type `t`. To use the `inv` predicate, the `SortedInts_Inv` module *clones* the `Inv` module, substituting the abstract type `t` with a concrete type. Cloning conceptually copies the definition of one module, applies substitutions, and inserts it into another module. In the example, the `Inv` module is cloned twice to distinguish between the invariant of the `SortedInts` and `i32` types. The generated axiom `inv_sorted_ints` equates `inv` with its definition consisting of the user invariant and a derived structural invariant. In case `self` is a `Cons`, the structural invariant includes the invariants of the head and the tail, which are expressed using the respective `inv` clones. The benefit of this encoding is that it correctly handles recursive and mutually recursive types, as shown in the example. Since each application of the axiom unfolds a single level of invariants, we avoid circular definitions.

For soundness, we must make sure to not generate contradictory unfolding axioms. Contradictory axioms are possible when the left-hand side of the equality occurs on the right-hand side in negated form. Since such negated invariants cannot occur in the structural part of the right-hand side, it suffices to consider user invariants. Figure 3.10 demonstrates an ill-defined user invariant, for which Creusot$^+$ generates the unsound axiom $\forall\, \texttt{self}.\ \texttt{inv}(\texttt{self}) = \neg\texttt{inv}(\texttt{self})$. We safeguard against such invariants by imposing additional rules on defining user invariants. Firstly, user invariants are prohibited from using the `inv` symbol, preventing cases like the one in Figure 3.10. Secondly, we disallow the use of the type for which the invariant is defined in quantifiers, protecting against user invariants like `forall<x: Self> false`. Due to how Creusot$^+$ guards quantifiers, such an invariant results in a contradictory axiom stating $\forall\, \texttt{self}.\ \texttt{inv}(\texttt{self}) = \forall\, x.\ (\texttt{inv}(x) \to \bot)$.

At the time of writing, the enforcement of these rules is not yet implemented in Creusot$^+$. Instead, users are required to manually ascertain that their invariant definitions follow the described rules.

```
module Inv
    type t
    predicate inv (x : t) (* the abstract inv symbol *)
end

module SortedInts_Inv
    clone Inv as Inv0 with type t = sorted_ints
    clone Inv as Inv1 with type t = i32
    (* clone translated Invariant trait implementation *)
    clone Invariant_Impl_SortedInts as UserInv

    (* the unfolding axiom for sorted_ints *)
    axiom inv_sorted_ints : forall self : sorted_ints .
        Inv0.inv self = UserInv.invariant self /\ match self with
        | Nil -> true
        | Cons x xs -> Inv1.inv x /\ Inv0.inv xs
        end
end
```

Figure 3.9: The unfolding axiom generated for `SortedInts` (WhyML syntax).

```
impl Invariant for Unsound {
    #[predicate]
    fn invariant(self) -> bool {
        !inv(self)
    }
}
```

Figure 3.10: Example of a user invariant resulting in an unsound axiom.

3.3.4 Invariant Elision & Parametricity

When a type neither has a user invariant nor structurally contains values of types with a user invariant, its invariant is *trivial*. This means that all values of that type satisfy the invariant (i.e., the invariant is a tautology). Invariant elision is an optimization that removes assertions of trivial invariants. During contract elaboration, Creusot$^+$ determines for each type whether it has a trivial invariant, in which case it skips generating the respective pre- or postcondition. For example, there is no precondition for an argument of type `Option<i32>`. Similarly, Creusot$^+$ does not assume trivial invariants for prophecies nor asserts them at resolution. Additionally, during the generation of structural invariants, fields with trivial invariants are ignored. However, checking invariant triviality generally only works for monomorphic types. For example, whether the invariant of the polymorphic type `Option<T>` is trivial depends on the instantiation of the parameter `T`.

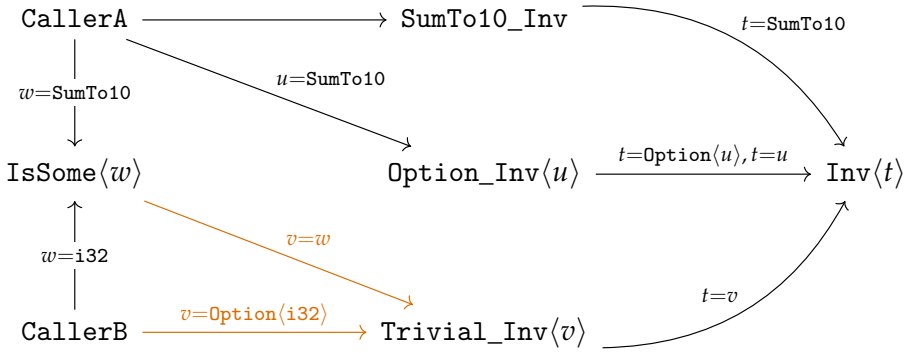
As an additional optimization for polymorphic types, Creusot$^+$ exploits the split between abstract symbol and unfolding axiom in the encoding of invariants (cf. §3.3.3). This enables Creusot$^+$ to give different interpretations to `inv` in different contexts. When instantiating a generic type such that its invariant becomes trivial, Creusot$^+$ provides an axiom that unfolds `inv` to true instead of the structural unfolding axiom. Therefore, the clone of the module generated for the generic type's invariant is replaced with a clone of the special `Trivial_Inv` module, shown in Figure 3.11.

```
1 module Trivial_Inv
2     type t
3     clone Inv as Inv0 with type t = t
4     axiom inv_trivial : forall self : t . Inv0.inv self = true
5 end
```

Figure 3.11: The trivial invariant unfolding axiom (WhyML syntax).

This optimization is illustrated by the example shown in Figure 3.12. It depicts several modules, represented by the nodes, and visualizes clones as arrows between the modules. Modules can have type parameters, indicated in angle brackets, and clones are annotated with the corresponding substitutions. Two substitutions for the same parameter mean that there are two distinct clones of the target module in the origin module. The example considers the modules generated for three Rust functions: `IsSome`, `CallerA`, and `CallerB`. The `IsSome` function is parametric over a type w and, given a value of type `Option`$\langle w \rangle$, returns true if the argument is a `Some` value. The functions `CallerA` and `CallerB` call `IsSome` on `Option`$\langle$`SumTo10`$\rangle$ and `Option`$\langle$`i32`$\rangle$ values, respectively. Since the invariant of `Option`$\langle$`SumTo10`$\rangle$ is not trivial, `CallerA` clones the full unfolding axiom for the `Option` type. However, in `CallerB`, Creusot$^+$ determines that the invariant of `Option`$\langle$`i32`$\rangle$ is trivial and thus clones the trivial unfolding axiom. This is indicated by the orange edge from `CallerB` to `Trivial_Inv`. Without the optimization, `CallerB` would also clone

`Option_Inv` like `CallerA`. As explained in Section 3.3.3, the modules for the unfolding axioms clone the `Inv` module defining the `inv` symbol. For simplicity, the graph hides transitive clones of `Inv` in the function modules.




```
1  fn is_some<T>(opt: Option<T>) -> bool {
2      match opt { Some(_) => true, None => false }
3  }
4
5  fn caller_a(opt: Option<SumTo10>) { is_some(opt); }
6  fn caller_b(opt: Option<i32>) { is_some(opt); }
```

Figure 3.12: Illustration of optimized clones (indicated by orange edges).

The clone of `Trivial_Inv` in `IsSome` (corresponding to the second orange edge) constitutes a special case. Leveraging parametricity [Rey83], invariants can be considered trivial as long as the corresponding types are generic. Consequently, in the body of `IsSome`, Creusot$^+$ can consider the argument's invariant trivial because w is fully parametric. Yet, `CallerA` is still required to prove the full, non-trivial type invariant before calling `IsSome`. The optimization is sound because parametricity means that it is impossible to break the invariant of a value in a generic way. While Rust's type system does not uphold parametricity in general due to ad hoc polymorphism with traits, it works in our more specific case. Thanks to Creusot's modular approach to verification, one function can trust the contracts of other functions. Consequently, when passing a generic value to a function, Creusot$^+$ can assume that the called function correctly enforces type invariants. Besides the usage as a function argument, the only other possible operations on generic values are moving it between variables and dropping it. Both operations cannot break the value's type invariant and, therefore, the parametricity optimization is sound.

3.4 Prophecy Resolution

Resolution is an analysis performed by Creusot that instruments the translated program with axioms revealing the definite values of prophecies. Once the value of a mutable reference is definite, a generated axiom equates the respective prophecy with the reference's current value. In Section 3.2.2, we explained how Creusot$^+$ prophetically assumes the invariant of final values. We argued that this is sound as long as a value's invariant is closed right before it is resolved. Therefore, correctly determining the points of resolution is crucial for soundness. In the course of our work on type invariants, Creusot's existing prophecy resolution algorithm proved to be incompatible with prophetic invariants. In the following, we present a new algorithm implemented in Creusot$^+$ and argue that it is correct.

3.4.1 Early and Late Resolution

Creusot's existing *early resolution* algorithm makes correct programs using type invariants unprovable as it resolves some variables before their type invariant can be proven. Therefore, we devised *late resolution*, a new approach to overcome these issues in Creusot$^+$.

Using the example shown in Figure 3.13, we explain why early resolution is incompatible with prophetic invariants. In the example, the variable `ra` reborrows `r`, projecting the mutable reference to the first field. Like any other borrow operation, it assigns the final value `^ra` to `r`. The assumption of the prophetic invariant `inv(^b2)` is omitted in the example because `^b2` has type `i32` and thus a trivial invariant. Early resolution resolves `r` before writing to `ra`. The created chain of prophecies `s.a == (^r).a == ^ra` guarantees soundness by keeping the still mutable part of `s` prophetic until `ra` is resolved as well. However, at the early resolution point, it is impossible to reason about the invariant of `r`. Because we do not yet know the value of `^ra`, we cannot prove or disprove the invariant `^ra + 7 == 10`. Late resolution delays the obligation to prove the invariant of `r` until after resolving `ra`. At this point, we know the value of `^ra` and can show that the invariant still holds after the write. To summarize, while equivalent in terms of learning the values of prophecies, late resolution ensures that variables are only resolved once we have enough information to prove their invariants.

```
1 let s = SumTo10 { a: 3, b: 7 };
2 let r = &mut s; // s <- ^r, assume inv(^r)
3 let ra = &mut r.a; // (*r).a <- ^ra
4 // (EARLY) resolve r: prove inv(r), assume ^r == *r
5 *ra = 3;
6 // resolve ra: assume ^ra == *ra
7 // (LATE) resolve r: prove inv(r), assume ^r == *r
```

Figure 3.13: Example of how early and late resolution handle reborrows.

In the following, we explain the criteria determining when to resolve a variable in late resolution.

Liveness. Fundamentally, resolution is based on liveness, a classic data-flow analysis determining the set of live variables at each program point. A variable is live if it is still used in any further execution of the program. Once a variable is dead, we know that any contained mutable references will not be written anymore. We write L_p and D_p to denote the sets of live and dead variables at a program point p.

Initialization. Besides liveness, resolution needs to take into account whether a variable is initialized. A variable becomes uninitialized when its value is moved. For example, the assignment `x = y`, where both variables are of type `&mut T`, moves the value of `y` into `x`. Even if `y` is dead after the assignment, it is no longer initialized and thus should not be resolved. This is the expected behavior as the mutable borrow can still be modified through `x`, and so its final value is not yet known. We write I_p for the set of initialized variables at program point p.

Borrowing. Another way a mutable reference may still be modified after being dead is through *reborrows*. Reborrowing describes the derivation of new a reference from an existing one, which becomes blocked as long as the reborrow exists. It is used, for instance, to project a reference of type `&mut (T, U)` to a reference of type `&mut T`. We want to ensure that mutable references are only resolved after their reborrows have been resolved. Therefore, late resolution takes into account the set of currently borrowed variables B_p. To determine this set, Creusot$^+$ relies on information provided by the Rust compiler.

Figure 3.14 exemplifies the data-flow analyses. The comments indicate the sets L_p, I_p, and B_p after each statement.

```
1  let s = SumTo10 { a: 3, b: 7 }; // L={s},  I={s},          B={}
2  let r1 = &mut s;                // L={r1}, I={s, r1},     B={s}
3  let r2 = r1;                    // L={r2}, I={s, r2},     B={s}
4  let ra = &mut r2.a;             // L={ra}, I={s, r2, ra}, B={s, r2}
5  *ra = 3;                        // L={ra}, I={s, r2, ra}, B={s, r2}
6  drop(ra);                       // L={},   I={s, r2},     B={s}
```

Figure 3.14: Example of the data-flow analyses used in resolution.

In summary, late resolution resolves a variable once it is dead, initialized and not borrowed. This yields the set R_p of late-resolved variables at program point p, defined as follows:

$$R_p := I_p \setminus (L_p \cup B_p)$$

The original early resolution algorithm considers only liveness and initializedness. Hence, the set of early-resolved variables is $I_p \setminus L_p \subseteq R_p$. In Figure 3.13, early resolution resolves `r` before writing to `ra` as it is dead and initialized after being reborrowed. Late resolution additionally takes into account whether a variable is borrowed. Since `r` remains blocked until the end of `ra`'s lifetime, the obligation to prove the invariant of `r` is delayed until after resolving `ra`.

3.4.2 Resolution Algorithm

So far, we have seen how the set of resolved variables is determined at a specific program point. Next, we explain how exactly this information is used to insert the resolve statements. As a reminder, a resolve of a variable asserts that its invariant holds and then structurally assumes the current and final value of all contained mutable references are equal. Algorithm 1 shows the late resolution algorithm for a simplified model of Rust's MIR. It takes a function f structured as a control flow graph (CFG) that contains a set of basic blocks $f.blocks$. A basic block b consists of a list of statements $b.stmts$, a set of predecessors $b.preds$, and successors $b.succs$. For simplicity, we only consider assignment statements s which have a left-hand-side variable $s.lhs$. On the right-hand side of these assignments are function applications of the form $g(e)$, where g is the name of a function and e is an opaque expression. We can assume that the left-hand side of an assignment does not occur in its right-hand side, as $v = g(v)$ can be rewritten as $v' = g(v);\ v = v'$ where v' is a fresh variable. Each statement has two associated program points $s.start$ and $s.end$ denoting the points before and after the statement, respectively. Analogously, for a block b we define the points $b.start$ and $b.end$ as the points before the first statement and after the last statement, respectively. We write Resolve(p, V) to insert resolves for all variables in the set V at program point p.

The algorithm first performs a data-flow analysis to compute the liveness set L_p, the initialization set I_p, and the borrowing set B_p for each program point p. From these sets, the set of resolved variables R_p and the set of variables that need eventually be resolved N_p are computed. The set N_p includes initialized variables that are not yet resolved and is defined as follows:

$$N_p := I_p \setminus R_p = I_p \cap (L_p \cup B_p)$$

Fundamentally, at a statement s, the algorithm resolves those variables that are resolved after the statement but have not already been resolved before (line 5). The remainder of the algorithm handles special cases. When the left-hand side of an assignment still needs to be resolved, it must be resolved before the assignment (line 7). Resolving it at a later point would be impossible because the assignment overwrites the old value. When the left-hand side of an assignment is dead after the assignment, it is immediately resolved. This is necessary because such a variable may already be resolved before the assignment and thus does not transition from not resolved to resolved. Lines 10–14 handle merge points in the CFG. For those, the algorithm inserts a new block b^* on each incoming edge to account for differences in resolved variables

between predecessors. In each such block, it resolves those variables resolved at the start of the current block but not at the end of the preceding block (line 12). Additionally, it resolves variables that need to be resolved at the end of the preceding block but not at the start of the current block (line 13). This ensures that Creusot$^+$ correctly resolves variables that are only initialized in some of the predecessors. Finally, in the return block, all variables that still need to be resolved are resolved except the return variable $f.ret$ (line 16).

Algorithm 1 Late Resolution

```
1:  procedure InsertResolves(f)
2:      Compute D_p, R_p, and N_p for all program points p in f
3:      for all b ∈ f.blocks do
4:          for all s ∈ b.stmts do
5:              Resolve(s.end, R_{s.end} \ R_{s.start})
6:              if s.lhs ∈ N_{s.start} then                ▷ Resolve reassigned variables
7:                  Resolve(s.start, {s.lhs})
8:              if s.lhs ∈ D_{s.end} then                  ▷ Resolve never live variables
9:                  Resolve(s.end, {s.lhs})
10:         for all b' ∈ b.preds do                        ▷ Resolve between blocks
11:             Create an empty basic block b*
12:             Resolve(b*.end, R_{b.start} \ R_{b'.end})
13:             Resolve(b*.end, N_{b'.end} \ N_{b.start})
14:             Insert b* between b' and b
15:         if b.succs = ∅ then                            ▷ Resolve before return
16:             Resolve(b.end, N_{b.end} \ {f.ret})
```

3.4.3 Correctness of Resolution

The resolution algorithm is correct if it enforces the soundness of prophecies and prophetic invariants. This requires the following two conditions:

1. No live variable is resolved, and
2. Every value except the one returned from the function is resolved eventually.

We first argue that these conditions are sufficient for correctness and then prove that they are satisfied by the resolution algorithm. Prophecies are sound if the assumption $\hat{}\texttt{r} = *\texttt{r}$, made when resolving a mutable reference r, is correct. This means that the current value of r at the point of resolution must be the actual final value; writing to r after it has been resolved would be unsound. A reference can be written to by dereferencing a variable that refers to the reference. Because dereferencing a variable counts as use in the liveness analysis, it is sufficient to show that the algorithm never

resolves live variables (condition 1). Prophetic invariants are sound if there is a proof obligation for the invariant of every value (cf. §3.2.3). The invariant of the return value is taken care of by the corresponding synthesized postcondition. Invariants of other values are asserted when these values are resolved, making condition 2 sufficient.

We now show that the resolution algorithm fulfills the desired properties.

Theorem 3.4.1. *Let f be a function. For all program points p and sets V, if* InsertResolves(f) *calls* Resolve(p, V), *then $L_p \cap V = \emptyset$.*

Proof. Let f be a function, p a program point and V a set of variables, such that V is resolved at p. We consider each call to Resolve.

- Line 5: $L_{s.end} \cap (R_{s.end} \setminus R_{s.start}) = \emptyset$
  Because $L_{s.end} \cap R_{s.end} = \emptyset$ and $(R_{s.end} \setminus R_{s.start}) \subseteq R_{s.end}$.

- Line 7: $L_{s.start} \cap \{s.lhs\} = \emptyset$
  Because $s.lhs$ is dead before the assignment by the definition of liveness.

- Line 9: $L_{s.end} \cap \{s.lhs\} = \emptyset$
  Because the algorithm checks that $s.lhs$ is dead after the assignment.

- Line 12: $L_{b^*.end} \cap (R_{b.start} \setminus R_{b'.end}) = \emptyset$
  Because $L_{b^*.end} = L_{b.start}$ and $L_{b.start} \cap R_{b.start} = \emptyset$.

- Line 13: $L_{b^*.end} \cap (N_{b'.end} \setminus N_{b.start}) = \emptyset$
  Because $L_{b^*.end} = L_{b.start} = L_{b.start} \cap I_{b.start} \subseteq N_{b.start}$.

- Line 16: $L_{b.end} \cap (N_{b.end} \setminus \{f.ret\}) = \emptyset$
  Because $L_{b.end} = \{f.ret\}$.

□

The second condition ensures the soundness of prophetic invariants. While the first condition considered variables, we must now consider values (i.e., the data named by variables). We consider a single function f and, in the spirit of modular verification, assume that the algorithm works correctly for any function called by f.

Theorem 3.4.2. *Let f be a function. For all values x in f, either*

- *x is moved into another function, or*

- *$f.ret$ refers to x when f returns, or*

- *There is a point p, a set V, and a variable $v \in V$ such that* InsertResolves(f) *calls* Resolve(p, V) *and v refers to x at the point p.*

Proof. Let f be a function and x a value in f. If f returns x, we are done. So, we assume that f does not return x. We consider how x is introduced. If x is the result of an expression used as an argument in a function application, it gets correctly resolved by the called function. Otherwise, x is an argument passed to f or the return value of a called function. Then, x is referred to by the variable corresponding to the argument, or the variable on the left-hand side of the assignment containing the call, respectively. We call that variable v and consider its membership in N_p for every program point p.

1. If $v \in N_p$ holds for all p, then either
   - There is a statement s that reassigns v, and therefore x gets resolved before s in line 7, or
   - The variable v still refers to x at the end of the function and, since $v \neq f.ret$, x is resolved in line 16.
2. If there exists a point p such that $v \notin N_p$ because $v \in R_p$, then either
   - There is a statement s such that $v \notin R_{s.start}$ and $v \in R_{s.end}$. Then, the algorithm resolves x after s in line 5.
   - There is an edge between two blocks b' and b such that $v \notin R_{b'.end}$ and $v \in R_{b.start}$. Then the algorithm creates a new block on that edge in which x is resolved in line 12.
3. If there exists a point p such that $v \notin N_p$ because $v \notin I_p$, then either
   - There is a statement s such that $v \in I_{s.start}$ and $v \notin I_{s.end}$. Then, v must be a call operand and x is moved into the called function, which correctly resolves x.
   - There is an edge between two blocks b' and b such that $v \in I_{b'.end}$ and $v \notin I_{b.start}$. We can assume $v \notin R_{b'.end}$ as we proved the contrary in case 2. Hence, we know $v \in N_{b'.end}$ and $v \notin N_{b.start}$, as $N_{b.start} \subseteq I_{b.start}$. So, the algorithm creates a new block on that edge in which x is resolved in line 13.

□

3.4.4 Incompleteness of Resolution

While late resolution works well for most programs using prophetic invariants, some programs remain unprovable. Those are not fundamental limitations of our approach but rather technical consequences of the resolution algorithm. Three such cases are demonstrated in Figure 3.15.

In the first example `two_refs`, Creusot$^+$ is forced to resolve the variable `x` despite it being borrowed by `xa`. Since the assignment `x = y` overwrites its previous value, it is the last possibility to resolve the original value. Consequently, users must prove its invariant before the assignment, which is impossible as it would require knowing the

value of `^xa`. As a potential solution to this issue, Creusot$^+$ could automatically create a ghost snapshot of `x` before the reassignment and resolve it at the end of `xa`'s lifetime. Future work is required to investigate feasibility.

The second function `project` returns a reference obtained by reborrowing its argument. It encounters a similar issue as the previous example. The argument `x` is resolved at the end of the function, which requires knowing the final value of the return value to prove the invariant. Because the caller may write any value to the returned reference, it is impossible to close the invariant of `x` in `project`. To solve this issue, `project` would need a special kind of precondition restricting how the caller may use the returned reference.

The last example function takes a pair consisting of an owned and a borrowed value. Assigning the first component to `x0` moves it out of the pair, making it partially uninitialized. As resolution still considers such values initialized, `x` is resolved at the end of the function. This is necessary to ensure the second component is correctly resolved. However, this means that the invariant is no longer provable, as the uninitialized field has the unknown value `any`. Solving this issue would require changing the resolution algorithm to work on a per-place basis instead of only considering variables.

```
1  fn two_refs<'a>(mut x: &'a mut SumTo10,
2                  y: &'a mut SumTo10) {
3      let xa = &mut x.a; // (*x).a <- ^xa
4      // resolve x due to reassignment
5      // cannot prove inv(x): ^xa unknown
6      x = y;
7      *xa = 3;
8  }
9
10 fn project(x: &mut SumTo10) -> &mut i32 {
11     &mut x.a // (*x).a <- ^result
12     // resolve x due to return
13     // cannot prove inv(x): ^result unknown
14 }
15
16 fn partial_move(x: (SumTo10, &mut SumTo10)) {
17     let x0 = x.0; // x <- (any, x.1)
18     // resolve x due to return
19     // cannot prove inv(x): x.0 unknown
20 }
```

Figure 3.15: Examples demonstrating the limitations of resolution.

3.5 Limitations

At the time of this writing, type invariants in Creusot$^+$ face certain limitations due to either incomplete implementation aspects or unresolved design considerations. We provide a summary in the following:

- Presently, Creusot$^+$ does not support user invariants for partially instantiated types or adding trait bounds beyond those present on the type definition. For example, the type `Generic<T: Bound>` only permits invariant definitions for exactly that type, keeping the parameter `T` generic. In theory, it should be possible to define user invariants for instantiated types like `Generic<Specialized>` or further constrained types, such as `Generic<Bound + OtherBound>`. This limitation primarily stems from technical challenges in resolving user invariant implementations during the generation of the unfolding axiom for a type. The potential support for constraining user invariants with additional trait bounds requires further consideration to ensure its soundness.

- The generation of structural invariants does not yet cover all Rust types. In particular, Creusot$^+$ lacks support for deriving structural invariants for closures, which should include the invariants of all captured values. This is a technical limitation due to the way Creusot translates closures. Furthermore, Creusot$^+$ does not yet support invariants for empty types, such as enums without variants or the never type `!`. While it would be plausible to assign the invariant `false` to these types, it remains unclear how this aligns with the inhabitation requirement of type invariants.

- Creusot$^+$ does not yet provide user invariant definitions for all types in Rust's standard library that require them. Specifically, container types like `Vec`, which internally use unsafe code, necessitate manual invariant definitions. While Creusot$^+$ defines the invariant for `Vec`, definitions for less commonly used types are still missing.

- Passing values with open invariants between functions is not yet supported in Creusot$^+$. This would be useful for invariant definitions containing calls to auxiliary logic functions with postconditions. Currently, defining an invariant passing `self` to such a function results in an unprovable VC asserting the argument's invariant. The exact mechanics of this feature, especially regarding its soundness, remain an open design question.

- Lastly, the issues discussed in Section 3.4.4 limit which programs can be verified.

4 Ghost Code

This chapter discusses the design and implementation of ghost code in Creusot. While support for ghost code existed in Creusot before our work, it faced several problems leading to unsoundness. Our contribution thus lies in the analysis of the soundness of ghost code and in modifications to the existing implementation to solve these issues. First, Section 4.1 motivates the design by studying several typical use cases. Next, Section 4.2 presents the existing support for ghost code. Finally, Section 4.3 analyzes the soundness of Creusot's ghost code, explains its issues, and proposes countermeasures.

4.1 Design Goals

We distinguish *ghost code* and *program code*. While program code just describes regular, executable Rust code, ghost code is not executed and solely exists to facilitate verification. Both coexist and ghost code usually needs to refer to program values to be meaningful. Yet, ghost code does not share the same semantics as program code. The erasure property of ghost code, requiring that its presence cannot affect the enclosing program code, limits the set of allowed operations. For example, operations such as writing to mutable references or diverging the control flow by returning from a function would be observable from program code and are thus illegal in ghost code. Besides those restrictions, there are other operations that are only meaningful in ghost code. These encompass logic-level constructs such as quantifiers or the final value operator. The central challenge therefore is the design of rules governing the use of ghost code while keeping its interaction with program code sound.

To better understand the requirements of ghost code, we consider several typical use cases.

Assertions, Auxiliary Functions, and Lemmas. A common use case for ghost code is the decomposition of complex properties into simpler ones. Lemma functions (cf. §2.3) and `proof_assert!` assertions guide verification by proving interim properties. While assertions are automatically available as hypotheses for subsequent goals, users must manually add lemma functions to the proof context by invoking them in ghost code. Auxiliary logic functions enable richer assertions by abstracting logical properties. As all of these concepts exist purely for verification, they constitute different forms of ghost code.

Ghost Snapshots. To specify how a variable is updated, it is often useful to relate its original value to the new value. Such specifications are particularly important for loop invariants if a variable is successively modified in the loop. For example, an in-place sorting function typically ensures that the sorted result is a permutation of the input list, i.e., no elements are added or removed. Figure 4.1 illustrates this principle using an augmented version of the `gnome_sort` example (cf. Figure 2.6). In the contract, the current and final value operators are used to express the relationship between input and output. To prove the added postcondition, the sorting loop must preserve the permutation invariant in each iteration. Unlike the contract, loop invariants only have access to the current, partially sorted value of the list. To relate the initial, unsorted value to the current value, we must explicitly bind the initial value to a name at the start of the function. This is achieved using *snapshots*, which assign a name to the value of a variable at a specific program point so that it can be referred to in specifications. In the example, we create a snapshot `old_v` at the start of the function using the `gh!` macro marking its contents as ghost code. If the assignment were normal Rust, it would move `v` into `old_v`, making further uses of `v` illegal. However, since ghost code in Creusot does not have the same ownership semantics as Rust, the assignment does not affect `v`.

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

Figure 4.1: The `gnome_sort` example using ghost snapshots.

Instantiation of Existential Quantifiers. Specifications containing existential quantifiers are challenging to prove automatically since they require provers to construct a witness satisfying the required property. Ghost code can help move this burden from the prover to the user by making the construction of the witness explicit in the specification. This is especially useful if the witness is constructed from intermediate values no longer present at the time the existence should be proven. Ghost code allows keeping those intermediate values without additional overhead for the program code. An example illustrating this concept is shown in Figure 4.2. Given a function `f`, the example counts the values $i \in [0, 100)$ where `f.eval`$(i) = 0$. The program function `f.eval` is specified

using the logical predicate `f.p` where `f.eval`$(i) = x \leftrightarrow$ `f.p`(i, x). To prove the assertion at the end of the example, provers must instantiate the quantified term with a sequence `s`. This is facilitated by the ghost variable `zeros` that directly satisfies the assertion.

```
1  // Count values i s.t. f.eval(i) == 0
2  let mut count = 0;
3  let mut zeros = gh! { Seq::EMPTY };
4  let mut i = 0;
5
6  #[invariant(0 <= i@ && i@ <= 10 && count@ <= i@)]
7  #[invariant(count@ == zeros.len())]
8  #[invariant(forall<j: Int>
9      (0 <= j && j < i@ && f.p(j, 0)) == zeros.contains(j))]
10 while i < 100 {
11     if f.eval(i) == 0 {
12         proof_assert! { f.p(i@, 0) };
13         count += 1;
14         zeros = gh! { zeros.push(i@) };
15     }
16     i += 1;
17 }
18
19 proof_assert! { exists<s: Seq<Int>> count@ == s.len() &&
20     forall<i: Int> (0 <= i && i < 100 && f.p(i, 0)) == s.contains(i) }
21 // Additionally, values in s must be unique (omitted for simplicity)
```

Figure 4.2: Example of ghost code used to instantiate an existential quantifier.

4.2 Ghost Code in Creusot

In this section, we describe how ghost code works in Creusot.

Ghost code in Creusot is based on its specification language Pearlite. It is used in contracts, as the body of auxiliary logic functions, or inside of macros such as `gh!` or `proof_assert!`. Pearlite code is type-checked by the Rust compiler but not borrow-checked. Borrow checking is an analysis performed by the Rust compiler to detect violations of its ownership and borrowing rules. In particular, it rejects code where a variable is used while being borrowed or after it has been moved. Since Pearlite code is not borrow-checked, it can use variables without moving them, enabling the snapshot use case. In other words, it treats every type as copyable, allowing the duplication of any value. However, this is at odds with Rust's non-aliasing guarantees that rely on the fact that mutable references are linear and thus cannot be duplicated. Hence, rules are

needed to ensure that no unsoundness arises from this difference in the semantics of mutable references.

In the following, we first explain how ghost code and program code can interact and then argue how the erasure property of ghost code is guaranteed.

4.2.1 Embedding Ghost Code in Program Code

The grammar of Pearlite is derived from the grammar of Rust expressions. For instance, Pearlite expressions permit literals, variables, binary operators, function calls, type constructors, pattern matching, and field accessors. Notably, Pearlite does not support loops, control flow constructs (e.g. `return`, `break`), or assignments. The lack of assignments means that ghost code cannot mutate variables or write to mutable references. Yet, it is often desirable to store the result of a ghost expression in a variable that persists over several points in the program.

This is enabled by the `gh!` macro and the `Ghost` type, which allow embedding ghost code in program code. The `gh!` macro takes a Pearlite expression of type `T` and produces a Rust value of type `Ghost<T>`. Such values live in program code and can thus be assigned to variables, passed as function arguments, or stored in structs. However, program code cannot extract the inner value of type `T`. Only ghost code can call the ghost function `inner` to access the contained value. Declaring a data type's field to be of type `Ghost` creates a *ghost field*. Such fields associate concrete Rust values with abstract values that can be used in ghost code.

4.2.2 Ghost Code Erasure

Ghost code erasure relies on the dead code elimination optimization performed by the Rust compiler and its backends. In Rust, `Ghost<T>` is defined as a *zero-sized type*, meaning that its runtime representation does not occupy any memory. It only exists in Rust's type system and is erased during runtime. Operations loading or storing values of type `Ghost<T>` are thus no-ops and get removed by the compiler during optimizations.

To ensure that erasing ghost code does not alter a program's behavior, ghost code must not interfere with program code. Noninterference dictates that ghost code does not have any observable side effects. Pearlite expressions uphold noninterference because they terminate, are pure, and cannot update variables or write to mutable references. As Pearlite does not support loops, it suffices to guarantee that recursive logic functions terminate. This is discussed in detail in Section 4.3.1.

4.3 Soundness Analysis

This section analyzes how ghost code in Creusot can lead to unsoundness and proposes several measures to prevent such issues. Importantly, this analysis does not constitute a comprehensive proof of soundness, which is beyond the scope of this thesis. Instead, the analysis aims to identify a set of necessary conditions for soundness in a structured

manner but does not claim that this set is complete. Because support for ghost code, as described so far, existed in Creusot prior to our work, the soundness analysis constitutes the main contribution of this chapter. Based on our analysis, we suggest enhancements to the existing ghost code support and write "Creusot$^+$" to refer to the version of Creusot implementing those.

First, we consider how to ensure that Pearlite is sound as a proof language. In particular, this entails that all auxiliary logic functions terminate and that types do not permit infinite values. These two issues are discussed in Section 4.3.1 and Section 4.3.2, respectively. Second, we regard the interaction of Pearlite with Rust code. Specifically, ghost code must uphold the conditions required for the soundness of prophecies. We discuss the soundness of prophecies in interaction with ghost code in Section 4.3.3.

4.3.1 Termination of Auxiliary Functions

Auxiliary logic functions are required to terminate. While Pearlite does not support loops, nontermination can be the result of recursion, whether directly or indirectly. Indirect forms of recursion encompass mutually recursive functions and recursion using traits. We consider each type of recursion in the following.

Direct Recursion. Creusot permits directly recursive logic functions but necessitates annotating them with the `variant` attribute, specifying a decreasing quantity. The quantity, expressed in terms of the function's parameters, must have a lower bound and must decrease with every recursion. For each recursive call, the user is thus obligated to prove that the quantity, when applied to the callee's arguments, is less than when applied to the caller's arguments. Figure 4.3 illustrates how this prevents unsound recursive functions. Verifying the logic function `direct_rec` would constitute an unsound proof of `false` due to its postcondition. This is prevented by the `variant` attribute dictating that the argument `x` must decrease in every recursive call. Because the argument of the inner call is `x - 1`, the variant requirement is satisfied. However, the lower bound of `x`, specified as a precondition, forces guarding the recursive call in an if-expression. Thus, soundness is maintained as verifying `direct_rec` still requires proving the (unprovable) postcondition in the case where `x` is 0.

Mutual Recursion. Figure 4.3 shows an example of unsound mutual recursion. The postconditions of `mutual_rec1` and `mutual_rec2` can be proven using each other. Creusot rejects such code because it does not support mutually recursive logic functions. During translation, Creusot determines a linear order of definitions based on a topological sort of the call graph. In the case of a cycle in the call graph, Creusot raises an error.

Recursion Through Traits. Traits, which are also supported in Pearlite, can encode nontermination by passing a function to itself as an argument. Figure 4.4 illustrates recursion using traits. The function `call_f` takes an argument `f` of generic type `F` bound

```
#[logic]
#[requires(x >= 0)]
#[ensures(false)]
#[variant(x)]
fn direct_rec(x: Int) {
    if x > 0 {
        direct_rec(x - 1)
    } else {
        // cannot prove false
    }
}
```

```
#[logic]
#[ensures(false)]
fn mutual_rec1() {
    mutual_rec2()
}

#[logic]
#[ensures(false)]
fn mutual_rec2() {
    mutual_rec1()
}
```

Figure 4.3: Example of (mutually) recursive logic functions.

by the trait `Func`, representing a first-class function. We create a new first-class function by implementing `Func` for the singleton type `Rec`. In its implementation of the `call` method, it calls `call_f` on the method receiver argument `self`. Consequently, invoking `Rec.call()` will in turn call `call_f` with the argument `Rec`, which again invokes the `call` method on `Rec`. Similar to mutually recursive functions, such code is rejected by Creusot. The `call` method of `Rec` is not allowed to call `call_f`, because instantiating the generic `call_f` with the implementation of `Func` for `Rec` cyclically depends on the definition of `Rec::call`.

```
trait Func {
    #[logic]
    #[ensures(false)]
    fn call(self);
}

#[logic]
#[ensures(false)]
fn call_f<F: Func>(f: F) {
    f.call();
}
```

```
struct Rec;

impl Func for Rec {
    #[logic]
    #[ensures(false)]
    fn call(self) {
        call_f(self);
    }
}

// Rec.call()
```

Figure 4.4: Example of unsound recursion using traits.

4.3.2 Well-Formedness of Data Types

When Creusot translates Rust types to logic types, it must ensure that the resulting definitions are well-formed. In particular, recursive Rust types get translated to inductive

types, which must guarantee that any value of the type can be constructed by a finite number of applications of the constructors. Normally, Rust's type system prevents infinite values. However, this is no longer true with ghost fields, as is demonstrated in Figure 4.5. The comments indicate which equalities hold after each statement, representing `r` as a pair of its current and final values. Ghost values are created using the function `gh` and unwrapped using `inner`. The recursive type `GhostList` has two constructors `Nil` and `Cons`, which contains a mutable reference to itself as a ghost field. This allows constructing a self-referential value that gives rise to an unsound induction principle. We initialize the variable `x` with `Nil`, borrow `x` to obtain `r`, and take a ghost snapshot `g` of `r`. Next, we construct a `Cons` value from `g`, write it to `r` and resolve `r` learning the updated value of `x`. With that, we have created an infinite value: `x` contains `g` as a subterm which in turn contains `x`. Without ghost code, `r` would be moved into `g` and, thus, the Rust compiler would deny writing to `r` subsequently.

```
1  enum GhostList<'a> {
2      Nil,
3      Cons(Ghost<&'a mut GhostList<'a>>),
4  }
5
6  fn infinite_value() {
7      let mut x = GhostList::Nil;
8      let r = &mut x; // x = ?, r = (Nil, x)
9      let g = gh! { r }; // g = gh (Nil, x)
10     *r = GhostList::Cons(g); // r = (Cons (gh (Nil, x)), x)
11     // resolve r: x = Cons (gh (Nil, x))
12     proof_assert!(^g.inner() == x && x == GhostList::Cons(g));
13 }
```

Figure 4.5: Example of creating infinite values of recursive types with ghost fields.

To prevent unsoundness due to ill-formed type definitions, we propose additional rules for Rust types with ghost fields. Every recursive occurrence of a type in the types of its fields must not be in *ghost position*. A recursive occurrence is in ghost position if it is used directly or indirectly as the parameter `T` of the `Ghost<T>` type. In Figure 4.5, the recursive occurrence of `GhostList` is in ghost position because it is directly used in the ghost type's parameter. An indirect use means that the type recursively occurs as a parameter to another type, which in turn contains a ghost occurrence of the parameter. For example, consider the type `IndirectGhost<T>` which has a field of type `Ghost<T>`. Then, replacing `Ghost` with `IndirectGhost` in the definition of `GhostList` would still be ill-formed.

The implementation of these rules is twofold. On the one hand, Creusot$^+$ rejects recursive type definitions with ghost fields. However, this does not catch cases where a type indirectly includes itself in a ghost field. Therefore, Creusot$^+$, on the other hand,

relies on Why3's validation of inductive types, which dictates that a type may only occur *strictly positively* in its definition. This rejects types that recursively include themselves on the left-hand side of an arrow. In particular, the parameters of opaque, abstract types are not considered strictly positive. By declaring the ghost type as an abstract type, we can leverage this mechanism to reject ill-formed recursive types that are not detected by Creusot$^+$'s check.

4.3.3 Soundness of Prophecies in Ghost Code

The soundness of prophecies in Creusot relies on the guarantees of Rust's borrow checker. In particular, the current value of a mutable reference must not depend on its prophetic final value. Otherwise, causality loops are possible leading to unsound assumptions when resolving the reference. For example, we consider a mutable reference to a boolean `r` that has the current value `not (^r)`. Resolving `r` would assume `^r = *r = not (^r)`—a contradiction.

Direct Use of Prophecies. Using ghost code, it is possible to construct programs where the current value of a mutable reference depends on its prophecy. This is demonstrated in Figure 4.6, showing two variations of an unsound usage of ghost code. In both examples, we first create a ghost boolean `x` and mutably borrow it to obtain `r`. Next, we create a new ghost value by negating the prophecy of `r` and write it to `r`. In the first example, we refer to the prophecy by the borrowed variable `x` and in the second example we use the final value operator. From the logical perspective, both variations are equivalent. After that, `r` is resolved and we assume that `x = ^r = *r = gh (not (inner x))`. Applying `inner` to both sides of the equation directly yields the asserted contradiction.

Without ghost code, constructing such references would not be possible. The final value operator is only available in Pearlite and using `x` while it is borrowed violates Rust's borrowing rules and would thus be rejected by the compiler.

We suggest additional rules that forbid the unsound usage of borrowed variables and the final value operator in ghost code. For this purpose, Creusot$^+$ distinguishes two dialects of Pearlite: *logic* Pearlite and *ghost* Pearlite. While logic Pearlite is used in contracts and assertions, ghost Pearlite is used inside the `gh!` macro. Analogously, Creusot$^+$ distinguishes auxiliary functions containing logic Pearlite, marked with the `#[logic]` or `#[predicate]` attributes, from those containing ghost Pearlite, marked with the `#[ghost]` attribute. Logic functions are allowed to call ghost functions, but not vice versa. Unlike logic Pearlite, ghost Pearlite cannot use borrowed variables or the final value operator, preventing the issues demonstrated in Figure 4.6. Table 4.1 shows a comparison of Rust and the two Pearlite dialects. While theoretically sound, the ability to use borrowed variables in logic Pearlite is parenthesized, as such uses are often misleading, indicating user errors.

Use of Prophecies Through Equality. Another way to let the current value of a mutable reference depend on its prophecy is by using equalities. In particular, Creusot's

```
fn borrowed() {
    let mut x = gh! { true };
    let r = &mut x; // x = ?, r = (gh true, x)
    *r = gh! { !x.inner() }; // r = (gh (not (inner x)), x)
    // resolve r: x = gh (not (inner x))
    proof_assert! { x.inner() == !x.inner() } // UNSOUND!
}

fn final_value() {
    let mut x = gh! { true };
    let r = &mut x; // x = ?, r = (gh true, x)
    *r = gh! { !(^r).inner() }; // r = (gh (not (inner x)), x)
    // resolve r: x = gh (not (inner x))
    proof_assert! { x.inner() == !x.inner() } // UNSOUND!
}
```

Figure 4.6: Two examples demonstrating unsound usage of prophecies in ghost code.

Table 4.1: Comparison of Rust and Pearlite Dialects

	Rust	Ghost Pearlite	Logic Pearlite
Erased	no	yes	yes
Borrow-Checked	yes	no	no
Must Terminate	no	yes	yes
Use Final Value	no	no	yes
Use Borrowed Variables	no	no	(yes)
Call Rust Functions	yes	no	no
Call Ghost Functions	no	yes	yes
Call Logic Functions	no	no	yes

representation of mutable references as a pair of current and final values results in an extensionality axiom stating

$$\forall \texttt{r1}, \texttt{r2} : \texttt{\&mut T} \,.\, \texttt{r1} = \texttt{r2} \iff (*\texttt{r1} = *\texttt{r2}) \wedge (\hat{}\texttt{r1} = \hat{}\texttt{r2})$$

How this can be exploited using ghost code is shown in Figure 4.7. We create a ghost boolean `x`, borrow it to obtain `r1` and take a snapshot `g` of the reference. Now, `r1` is resolved making `inner g` $=$ (`gh true`, `gh true`). Next, we create a second reference `r2` and compare it to `inner g`. The following equalities show that `r2` $=$ `inner g` $\iff$ `inner x`:

$$\begin{aligned}
& \texttt{r2} = \texttt{inner g} && \iff \\
\iff \quad & (\texttt{gh true}, \texttt{x}) = (\texttt{gh true}, \texttt{gh true}) && \iff \\
\iff \quad & \texttt{x} = \texttt{gh true} && \iff \\
\iff \quad & \texttt{inner x} = \texttt{true}
\end{aligned}$$

The extensionality axiom is used in the first step to rewrite the references into pairs. Writing the negated result of the comparison to `r2` yields the same unsound value as in the previous examples. Hence, a contradiction is assumed when `r2` is resolved, allowing to prove the false assertion.

```
fn extensionality() {
    let mut x = gh! { true }; // x = gh true
    let r1 = &mut x; // x = ?, r1 = (gh true, x)
    let g = gh! { r1 }; // g = gh (gh true, x)
    // resolve r1: x = gh true
    let r2 = &mut x; // x = ?, r2 = (gh true, x)
    *r2 = gh! { r2 != g.inner() };
    // r2 = (gh (not (r2 = inner g)), x) = (gh (not (inner x)), x)
    // resolve r2: x = gh (not (inner x))
    proof_assert! { x.inner() == !x.inner() }; // UNSOUND!
}
```

Figure 4.7: Example of unsound prophecies exploiting reference equality.

To guard against the unsound use of the extensionality of mutable references we propose to change the representation of mutable references. In Creusot$^+$, the encoding of mutable references should include a third value that is opaque and incomparable. Consequently, the equality of two references is no longer equivalent to the equality of their current and final values. However, this change has not yet been implemented at the time of writing and future work is required to determine the viability of this measure.

4.3.4 Summary

We discussed the soundness of ghost code in Creusot and proposed several measures to remedy the found issues. Table 4.2 shows an overview of the implementation status for each proposed measure. At the time of writing, all except one measure are implemented. The measures marked with * are functional but could be improved to be less restrictive or give better diagnostics.

Table 4.2: Implementation Status of each Soundness Measure

Issue	Measure	Creusot	Creusot$^+$
Termination	Variant attribute, cycle detection	✓*	✓*
Infinite Values	Check recursive type definitions	×	✓*
Final Value Operator	Ghost Pearlite	×	✓
Borrowed Variables	Ghost Pearlite	×	✓
Reference Equality	Change the encoding of `&mut`	×	×

5 Evaluation

In this chapter, we evaluate Creusot's support for type invariants and ghost code and highlight their connection. In Section 5.1, we explain which criteria determine the success of those features. In Section 5.2, we consider several case studies and describe the experiments conducted to assess the criteria. In Section 5.3, we discuss our results and summarize the limitations of our approach.

5.1 Evaluation Criteria

To evaluate type invariants and ghost code in Creusot we consider the following criteria:

(C1) Specification Expressivity: How concisely can users formulate specifications?

(C2) Verifiability & Prover Performance: How do the presented features impact the ability of provers to verify the VCs?

(C3) Usability & Robustness: How applicable are the presented features to real-world problems?

In the following, we elaborate on these criteria.

C1: Specification Expressivity. Providing sufficient expressivity to enable users to write succinct specifications that map closely to their understanding is crucial for accessibility of verification with Creusot. However, increased expressivity is at odds with avoiding overly complex and verbose specifications. As both type invariants and ghost code provide a means of abstraction, we hypothesize that the presented features help balance this tradeoff. To assess this criterion, we determine the specification overhead by counting the source lines of code (SLOC) making up contracts in proportion to the total SLOC.

C2: Verifiability & Prover Performance. Depending on how specifications and programs are represented logically, the time it takes provers to verify the VCs varies. Provers are usually allotted a maximum runtime, after which they are interrupted. In such cases, Why3 either tries simplifying the subgoal with transformations or aborts verification as a failure. Therefore, it is crucial that our encoding considers the strengths and weaknesses of the provers. The evaluation of this criterion considers whether provers manage to verify a program at all and the consumed time for verification. Verification time is measured by aggregating the times expended to prove each goal.

C3: Usability & Robustness. The applicability of the presented features to real-world problems hinges on minimizing the number of correct yet non-verifiable programs. Our design should embody composability and consistency, which are vital to avoid user confusion. Moreover, a priority should be placed on ergonomics and high automation to promote ease of use and to streamline the verification process. Additionally, the design should robustly guard against accidental misuse, ensuring that users encounter no subtle surprises during utilization. Usability and robustness are assessed qualitatively.

5.2 Case Studies and Experiments

In the following we consider two case studies to gain a better understanding of how well our presented design achieves the evaluation criteria. The first case study comprises several data structures that use type invariants to ensure their internal consistency. The second case study focuses on Rust's iterators, highlighting in particular how type invariants and ghost code complement each other. For each case study, we present the considered test cases, the conducted experiments, and their results. All experiments were performed on a machine running Arch Linux 6.5.5 equipped with an Intel i5-8250U CPU and 16 GB of RAM. We installed Alt-Ergo 2.5.1, Z3 4.12.2, CVC4 1.8, and CVC5 1.0.5 as backends for Why3. The source code of the utilized version of Creusot as well as the case studies is available online [Cre23].

5.2.1 Data Structures

In the first case study, we consider several data structures that use type invariants. These data structures require certain consistency properties such that their implementation is correct. We relied on existing implementations, which we adapted to utilize type invariants. In particular, we analyzed the following data structures:

- `vecmap` This test case is based on an implementation of an associative array originally presented in [Hay23]. It stores key-value pairs in a vector that is required to be sorted with respect to the keys. This permits the mapping type to check membership using binary search. We adapted the implementation to use type invariants to enforce the sortedness property.
- `sparse_array` This test case implements the sparse array data structure from the VACID-0 verification benchmark suite [LM10]. It allows storing a sparsely populated array in a dense, contiguous representation by keeping a mapping from virtual to physical indices. A type invariant ensures the consistency of this mapping. We adapted an existing implementation in Creusot's test suite to use type invariants.
- `bdd` This test case implements binary decision diagrams with hash consing. A context structure stores a mapping from hashes to nodes and a type invariant

ensures the consistency of the mapping and the underlying graph. We adapted an existing implementation in Creusot's test suite to use type invariants.

We conducted two experiments on each test case:

E1: Manual Encoding of Type Invariants. Instead of using the built-in type invariant support presented in this work, users can alternatively encode type invariants manually. This is achieved by defining a predicate stating the invariant property and explicitly adding pre- and postconditions to functions enforcing the invariant's preservation. The manual approach does not support prophetic invariants; instead, users have to resort to the non-composable method of adding postconditions for mutable reference arguments. We want to see how the built-in type invariants compare to manually encoded type invariants in terms of the defined evaluation criteria. Therefore, we created a variation of each data structure where we replaced built-in invariants with manual invariants. Figure 5.1 exemplifies the manual encoding of type invariants for the `insert` method, which inserts a value into a `VecMap`.

```
#[requires(self.is_sorted())]
#[ensures((^self).is_sorted())]
#[ensures(...)]
pub fn insert(&mut self, key: K, value: V) -> Option<V> {
    // ...
}
```

Figure 5.1: Manual encoding of invariants for `VecMap::insert`.

E2: Effect of Type Invariant Optimizations on Solvers. In Section 3.3.4, we explained two optimizations that simplify trivial type invariants. Invariant elision omits trivial invariants in contracts and structural invariants. The parametricity optimization instantiates fully generic invariants such that they are trivial. We want to determine whether these optimizations improve the solver's verification performance (C2). Consequently, we modified Creusot to allow optionally disabling the optimizations and measured the impact on verification times.

Results. Table 5.1 shows the results of our experiments. There are three rows per test case corresponding to the experiments E1 (manual), E2 (built-in, no-opt), and the baseline (built-in, opt). The baseline describes the built-in type invariants with enabled optimizations, as presented in this thesis. The first column shows the contract SLOC for each test case and each experiment. Experiment E2 does not affect the specifications and thus has the same values as the baseline. The percentages inside of the parentheses signify the proportion of contracts to total SLOC. We see that manually encoding type invariants results in a plus of 3 – 4 percentage points compared to

the baseline. Furthermore, using built-in type invariants increases verification times compared to manually encoded invariants. However, our measurements are evidence of the effectiveness of the optimizations. While the verification time more than doubles for `vecmap` without optimizations (built-in, no-opt) compared to the manual version (manual), the baseline version (built-in, opt) only sees an increase of 28.5 %.

Table 5.1: Evaluation Results for Data Structures

Test	Specification SLOC	Verification Time
`vecmap` (manual)	116 (27.4 %)	14.39 s
`vecmap` (built-in, no-opt)	95 (23.6 %)	34.85 s
`vecmap` (built-in, opt)	95 (23.6 %)	18.49 s
`sparse_array` (manual)	25 (17.7 %)	7.30 s
`sparse_array` (built-in, no-opt)	18 (14.4 %)	15.87 s
`sparse_array` (built-in, opt)	18 (14.4 %)	7.33 s
`bdd` (manual)	80 (15.6 %)	45.38 s
`bdd` (built-in, no-opt)	63 (12.7 %)	95.69 s
`bdd` (built-in, opt)	63 (12.7 %)	63.12 s

5.2.2 Iterators

Creusot includes a formalization of Rust's iterators, which utilizes both type invariants and ghost code. Hence, we consider iterators as a case study, demonstrating the synergy between ghost code and type invariants.

Rust's iterators offer a flexible and composable mechanism for sequential traversal of items in collections. Central to this mechanism is the `Iterator` trait, which every iterator implements. This trait specifies the `next` method, governing the generation of subsequent values in a sequence. During this process, iterators can modify their internal state. The `next` method returns an `Option` value, where `None` indicates the end of the iteration. Notably, Rust's for-loops are built upon this iterator mechanism, continuously invoking the `next` method until it returns `None`.

Creusot supports iterators by specifying them as state machines [DJ23]. Therefore, the `Iterator` trait is augmented with the two additional ghost predicates `produces` and `completed`. The `produces` predicate encodes the transition relation and `completed` encodes the set of final states. Given two states a and b and a sequence of values p, we write $\texttt{produces}(a, p, b)$ as $a \overset{p}{\rightsquigarrow} b$. Accordingly, the `next` method has a postcondition stating that the method performs a valid state transition. Every iterator implementation must define the two predicates and prove that `produces` is reflexive and transitive.

The case study comprises two parts. In the first part, we consider the specification of consuming an iterator using a for-loop and explain how ghost code helps eliminate existential quantifiers. In the second part, we specify an iterator adaptor using a

combination of ghost code and type invariants. We perform the following additional experiment to evaluate ghost code:

E3: Comparision of Ghost Code and Existential Quantifiers. A major use case for ghost code is the explicit construction of values that could otherwise only be expressed using existential quantifiers. For example, without snapshots, users would have to existentially quantify the historical value and constrain it with a predicate capturing its relation to the current value. We want to test the impact of ghost code on the performance of solvers when compared to equivalent specifications using existential quantifiers.

Consuming Iterators. The Rust compiler desugars for-loop syntax into a more primitive loop that repeatedly invokes the iterator. Figure 5.2 shows two desugarings of a for-loop, which differ only in their use of ghost code. The desugarings are equivalent to writing `for x in it { <body> }`. In both versions, the fixed iteration scheme enables the desugaring to provide a loop invariant "for free". The invariant states that `it` is obtained from `it0` by repeatedly calling `next`, or, in other words, $\exists\, p.\ \texttt{it}_0 \overset{p}{\rightsquigarrow} \texttt{it}$, which corresponds to the version on the left. The version on the right avoids the existential quantifier (E3) by explicitly tracking the sequence of produced values `produced` in ghost code. It therefore initializes the ghost variable `p` to an empty sequence before the loop and subsequently appends each item returned by `next`.

```
let it0 = gh! { it };
#[invariant(exists<p: Seq<Item>>
    it0.produces(p, it))]
loop {
    match it.next() {
        None => break,
        Some(x) => {
            // <body>
        }
    }
}
```

```
let it0 = gh! { it };
let mut p = gh! { Seq::EMPTY };
#[invariant(it0.produces(p, it))]
loop {
    match it.next() {
        None => break,
        Some(x) => {
            p = gh! { p.push(x) };
            // <body>
        }
    }
}
```

Figure 5.2: Consuming an iterator with a for-loop.

For experiment E3, we created a test program using for-loops and a variation where we replaced the for-loops with a manual desugaring with existential quantifiers in the loop invariant.

Iterator Adaptors. While the desugaring of for-loop already demonstrates how ghost code can replace existential quantifiers, it does not yet motivate the combined use of type invariants and ghost code. Therefore, we examine the specification of an *iterator adaptor* as a case study. An iterator adaptor is an iterator that wraps another iterator and applies certain transformations on the generated items. Typical examples include, for example, the `Map` adaptor, which applies a function to each item of the inner iterator, or the `Filter` adaptor, which only yields elements that satisfy a specific criterion. In our case study, we consider the adaptor `FilterZero`, which wraps an `u32` iterator and only yields non-zero values of the inner iterator. For the purpose of our case study, we use a simplified version of Rust's iterator trait, shown in Figure 5.3. Our version differs from the original formalization in [DJ23] in the following ways:

- Our iterators only iterate over values of type `u32` whereas Rust's iterators can iterate over values of any type.
- Our iterators have no final states and so the `next` method returns a `u32` value instead of an `Option` value.
- The specification of Rust's iterators uses a `produces` predicate which is the reflexive-transitive closure of our `produces` predicate. This means implementors are not required to prove reflexivity and transitivity in our version.

```
1  pub trait Iter {
2      #[predicate]
3      fn produces(self, x: u32, other: Self) -> bool;
4
5      #[ensures(self.produces(result, ^self))]
6      fn next(&mut self) -> u32;
7  }
8
9  struct FilterZeros<I: Iter> {
10    inner: I,
11    hist: Ghost<Seq<(I, u32)>>,
12 }
```

Figure 5.3: The `Iter` trait and the `FilterZeros` adaptor.

To implement `Iter` for `FilterZero`, we first define the `produces` predicate. A `FilterZero` iterator a produces a value x if the wrapped iterator produces a sequence of one or more elements where every element is zero except the last element which must equal x. This is expressed by the following transition relation (if $n = 1$, a.`inner` only produces a single, non-zero value):

$$a \overset{x}{\rightsquigarrow} b \iff x \neq 0 \ \wedge \ \exists h_1, \ldots, h_n.\ a.\texttt{inner} = h_1 \overset{0}{\rightsquigarrow} h_2 \overset{0}{\rightsquigarrow} \cdots \overset{0}{\rightsquigarrow} h_n \overset{x}{\rightsquigarrow} b.\texttt{inner}$$

Similar to how we use ghost code to replace the existential quantifier in the desugaring of for-loops, we follow that approach in this case. While we previously stored produced items in a local ghost variable, we must now use a ghost field, that is populated by the `next` method and can be referred to by `produced`. Figure 5.3 shows the definition of the `FilterZeros` type that stores a sequence of iterator-item pairs in a ghost field `hist`. This sequence stores the (zero-valued) items generated by the inner iterator since the last non-zero value along with snapshots of the inner iterator, representing its state before each item was generated. Defining a type invariant for `FilterZeros` allows exactly specifying the properties of `hist`, as shown in the following:

$$\texttt{uinv}(b) \iff \textbf{let}\ ((h_1, x_1), \ldots, (h_n, x_n)) = b.\texttt{hist}\ \textbf{in}\ h_1 \overset{x_1}{\rightsquigarrow} \cdots \overset{x_{n-1}}{\rightsquigarrow} h_n \overset{x_n}{\rightsquigarrow} b.\texttt{inner}$$

With this type invariant, we can reformulate the `produces` relation to use the ghost field `hist` as an explicit witness for the existential quantifier:

$$\begin{aligned} a \overset{x}{\rightsquigarrow} b \iff\ & x \neq 0 \wedge \textbf{let}\ ((h_1, x_1), \ldots, (h_n, x_n)) = b.\texttt{hist} \\ & \textbf{in}\ a.\texttt{inner} = h_1 \ \wedge \ x_n = x \ \wedge \ \forall i < n.\ x_i = 0 \end{aligned}$$

Results. Table 5.2 shows the results of our experiments on iterators. There are two test cases, corresponding to the presented studies on for-loops and iterator adaptors. Each test case has one baseline variant with ghost code (ghost) and one representing experiment E3 (existential), where ghost code is removed in favor of existential quantification. In the first test case concerning for-loops, replacing ghost code with existential quantifiers doubled the verification time. In the second test case, the version of `FilterZero` without ghost code could not be verified automatically by running Why3's "Auto level 3" proof strategy. Specifically, none of the backend solves was able to instantiate the existential quantifier in the `produces` predicate, which must be proven as part of a loop invariant of the `next` function.

Table 5.2: Evaluation Results for Iterators

Test	Verification Time
Consuming for-loops (ghost)	0.42 s
Consuming for-loops (existential)	0.83 s
`FilterZero` (ghost)	2.62 s
`FilterZero` (existential)	×

5.3 Discussion

In this section, we discuss the observations gained from the presented case studies and experiments. Not all observations can be directly attributed to experimental evidence; some are anecdotal and therefore have a speculative character. Consequently, we emphasize the limitations to the validity of our claims where applicable. The structure of this section follows the criteria; both positive and negative findings are discussed for each criterion.

5.3.1 Specification Expressivity

O1.1: Built-In Type Invariants Simplify Specifications. Experiment E1 showed that built-in support for type invariants enables simpler specifications as compared to manually encoded type invariants. While manual invariants achieve a similar effect as using the built-in type invariants, they require significantly more verbose specifications. Firstly, our approach does not require pre- and postconditions for type invariants as they are automatically generated by contract elaboration. Secondly, automatic derivation of structural invariants alleviates the need to explicitly define type invariants for many purely structural types.

O1.2: Ghost Code Improves Encapsulation. Ghost code enables auxiliary logic functions and lemma functions that provide a way to break down specifications into smaller units. This reduces code duplication, reducing the overall amount of required annotations.

5.3.2 Verifiability & Prover Performance

A major factor determining the efficacy of an SMT solver is its ability to instantiate formulas containing quantifiers. Instantiation is required as most specialized theory solvers (e.g. for linear arithmetic or bit vectors) can only reason about quantifier-free terms. To control instantiation, SMT solvers use heuristics, often based on triggers. A trigger is a term or pattern within a quantified formula that, when matched in the current formula set, prompts the instantiation of the quantifier. However, selecting good triggers is challenging, and improper trigger selection can lead to inefficiencies like *matching loops* [Mos09]. This situation arises when the instantiation of a quantified formula results in new terms, which in turn trigger further instantiations of the same or other quantified formulas. This can lead to an infinite sequence of instantiations without making any real progress towards determining satisfiability. In essence, the solver gets "stuck" in a loop of generating terms without ever concluding.

O2.1: Ghost Code Helps Solvers Instantiate Existential Quantifiers. Experiment E3 showed that goals containing existential quantifiers are harder to verify for solvers. Verification of such goals requires instantiating the quantified term with an explicit

witness. In cases where the construction of such witnesses is non-trivial, verification can fail due to solver timeouts. However, experiment E3 also showed that this challenge is alleviated when users provide witnesses using ghost code.

O2.2: Guarded Quantifiers Hamper the Instantiation of Univeral Quantifiers. Experiment E1 showed that built-in type invariants cause an increase in verification times compared to manually encoded type invariants. We conjecture that this is (partly) due to how quantifiers are guarded with type invariant predicates. The encoding of type invariants presented in Chapter 3 affects the ability of SMT solvers to select optimal triggers. In particular, quantifiers in Pearlite are guarded with invariant predicates, resulting in terms of the form $\forall x.\ \texttt{inv}(x) \rightarrow P(x)$. Similarly, axioms generated from lemma functions are guarded by the invariants of their parameters. Based on these formulas, SMT solvers may choose the invariant predicate as a trigger for instantiation, which is a suboptimal choice in most cases. To what extent this explains the increased verification times, is difficult to determine. Solvers give little indication of the exact factors determining their ability to prove a specific goal. Furthermore, these factors are usually not uniform for all solvers. Ultimately, more experiments are required to better understand the correlation between type invariants are verifiability.

O2.3 Optimization of Trivial Invariants Improves Solver Performance. Experiment E2 showed that the optimizations for trivial type invariants improve verification time. The effectiveness of the optimizations can be explained with observation O2.2. Eliding trivial invariants in quantified terms partially mitigates the instantiation issues.

O2.4: Manual Unfolding of Type Invariants is Required in Some Cases. The encoding of type invariants as uninterpreted `inv` predicate and unfolding axioms per type has been observed to impair verifiability in rare cases. While solvers are usually able to automatically apply the axioms as needed, manual rewrite transformations to unfold the invariant are required in some goals. Further investigation is required to determine the exact causes of this behavior.

O2.5: Solvers Struggle to Prove Invariant Inhabitation. For complex type invariants, it is often challenging for solvers to prove that the invariant is inhabited. Provers are required to instantiate the existentially quantified invariant term, which requires the construction of a witness. An alternative to enforcing inhabitation as a trait law would be to generate a proof obligation when creating a mutable reference that asserts the invariant of the borrowed value. This would potentially result in more proof goals but avoid the existential quantifier.

5.3.3 Usability & Robustness

Comparing manually encoded type invariants and built-in type invariants shows that the latter avoid several pitfalls of the former.

O3.1: Type Invariants Prevent Oversights. We conjecture that manually encoding type invariants is prone to oversights. Users might forget to include an argument's invariant in a function contract, leading to an unprovable goal when passing the argument to another function. Furthermore, with manual encoding, it is not obvious to users which types define type invariants. Built-in support for type invariant prevents such oversights.

O3.2: Prophetic Invariants Ensure Composability. Experiment E1 showed that without prophetic invariants, users have to resort to specifying postconditions for mutable reference arguments. As discussed in Section 3.1.2, this approach is not composable with generic functions.

O3.3: Type Invariants are not Automatically Preserved in Loops. Modifying a value that must satisfy a type invariant in a loop often requires a loop invariant enforcing the preservation of the type invariant. At the moment, Creusot does not automatically generate such loop invariants, instead requiring users to manually write such loop invariants. This is inconsistent with the general approach to hide type invariants in specifications.

6 Related Work

Refinement Types. Refinement types, first introduced by Freeman and Pfenning in 1991 [FP91], constrain their set of possible values with a logical predicate. For example, `{ x: i32 | x != 0 }` describes a subtype of `i32` which is constrained to be non-zero. The same type can be described using type invariants in Creusot by implementing the `Invariant` trait for a new type wrapping an `i32` value. While similar in expressivity, a key difference between our type invariants and refinement types lies in the method of enforcement. As refinement types make the constraining properties part of the type system, they are enforced through type checking. In contrast, type invariants in Creusot are translated into VCs which are verified independently of type checking.

Refinement Types for Rust. Flux [Leh+23] extends Rust's type system with refinement types, which it calls existential types. Furthermore, it introduces indexed types, a form of dependent types [XP99]. For example, the type `RVec<T>[n]`, which is indexed by an integer `n`, lifts the length of a vector to the type system. This lets users write `{ n. RVec<T>[n] | n > 0 }` for the refined type of non-empty vectors. While Creusot does not support dependent types, abstract values can be associated with concrete values using logic functions. In particular, the model operator associates an abstract sequence to each vector. Thus the type of non-empty vectors can be specified in Creusot by defining an invariant constraining the length of the vector's model. An interesting difference between Flux and type invariants in Creusot lies in the handling of mutable references. Flux distinguishes two kinds of mutable references: invariant-preserving and type-changing, or *strong*, references. While the former prohibit modifications breaking the invariant, the latter permit arbitrary modifications but require annotating the reference's updated type. In Creusot, mutable references always preserve their invariants across function boundaries, making them similar to the invariant-preserving variant in Flux. However, Creusot also allows temporary invariant-breaking updates, which are only possible using strong references in Flux. In summary, invariants in Creusot and Flux are comparably expressive regarding the specification of mutable references. In general, however, Creusot enables more complex invariants than Flux thanks to its more powerful logic that supports quantifiers, for instance.

Foundational Rust Verification. RustBelt [Jun+17] provides a semantic model for Rust by formalizing its type system in the Iris separation logic [Jun+18]. RustHornBelt [Mat+22] extends RustBelt with prophecies and thus establishes a semantic foundation for Creusot. In contrast to automated verification tools, these foundational models

require a manual translation from Rust into λ_{Rust}, a Coq-based representation, and manual proofs in Iris. However, being built on the Iris separation logic enables the verification of programs that internally use unsafe Rust features. This is harder in non-foundational tools like Creusot because specifying such programs requires general assertions about Rust's memory model, which goes beyond Creusot's prophecy-based model. RefinedRust [Gäh+23] aims to overcome this gap, combining automation and support for unsafe Rust features. Inspired by RefinedC [Sam+21], it translates Rust into the Coq-based operational semantics Radium. Specifications are supplied similar to Flux, by refining Rust types with predicates. While verification still requires a Coq proof, high automation is possible thanks to using RefinedC's Lithium engine.

Type Invariants in Prusti. Prusti [Ast+19] is a Rust verification tool similar to Creusot. It translates Rust into the Viper separation logic framework that, similar to Why3, uses SMT solvers for automated verification. Unlike Creusot, Prusti reconstructs the aliasing guarantees from Rust's type system in its logic. Instead of prophecies, it uses *pledges*: properties that can be assumed to be true at the end of a reference's lifetime. Prusti supports a form of type invariants through `#[invariant(...)]` attributes on struct and enum definitions. A potential advantage of Creusot's trait-based approach to defining invariants is the possibility of constraining invariants with trait bounds. However, Creusot's current implementation does not yet offer that flexibility (cf. §3.5), making trait-based invariants equally expressive as Prusti's attributes. Similar to Creusot, Prusti generates structural invariants based on type definitions. Unlike Creusot, Prusti generates postconditions for mutable references in function parameters. A benefit of Prusti's pledge-based approach is support for `assert_on_expiry` assertions. In contrast to regular pledges, these specify properties that *must be proven* at the end of a reference's lifetime. This enables specifying functions returning mutable references to types with invariants, which are currently unsupported by Creusot (cf. §3.4.4).

Type Invariants in Why3. Why3 [FP13] supports defining type invariants for record types. Similar to Creusot, type invariants must be shown to be inhabited and are enforced on function boundaries. However, unlike Creusot, all type invariants must be restored at function exit instead of just the return value's invariant. Moreover, users must show all newly constructed records satisfy their invariants while in Creusot values with an open invariant can be constructed. Since Creusot is based on Why3 and both support type invariants, in an alternative design, Creusot could translate type invariants into their Why3 equivalent. However, this would oblige Creusot's type invariants to tightly follow Why3's design choices, which may not be optimal choices for Rust. For example, Why3 forbids recursive types with invariants, whereas such types are supported in Creusot thanks to the encoding via unfolding axioms. Ultimately, not relying on Why3's type invariants means more flexibility and control.

Ghost Code in Why3. The WhyML language of Why3 supports ghost code [FGP16]. While WhyML is a unified language for both programs and specifications, it distinguishes program and logic expressions. Creusot translates Rust to program expressions and Pearlite to logic expressions. Unlike Creusot's `Ghost` type, WhyML's ghost code is embedded into program code using a `ghost` modifier that can be applied to let-bindings, function arguments and return types, and record fields. Only program expressions marked as ghost expressions can read or write variables annotated with a ghost modifier. However, ghost expressions cannot modify regular variables, ensuring their erasure without changing the surrounding program's behavior. The noninterference property of this approach is proven using a type system with effects and bisimulation [FGP16], drawing parallels to noninterference in information flow control.

Ghost Code in Verus. The Verus [Lat+23] Rust verifier stands out thanks to its support for linear ghost code, i.e., ghost code where Rust's ownership and borrowing rules are enforced. This approach contrasts with Creusot's non-linear ghost code, where values can be freely duplicated. To also support use cases benefitting from non-linear features, such as snapshot variables, Verus employs a stratified design, distinguishing three modes:

- `exec` for executable program code,
- `proof` for linear ghost code, and
- `spec` for non-linear ghost code.

While `spec`-mode code is similar to ghost code in Creusot, `proof`-mode code blends characteristics of the other two modes. Like `spec` code, it is erased, must terminate and be pure. Like `exec` code, it is borrow-checked and allows mutations.

Mode annotations apply to both functions and variables, akin to the ghost modifier in Why3. As hinted by the name, `proof`-mode functions are used to write lemma functions as `spec` functions cannot define postconditions. Notably, `proof`-mode variables are instrumental in encoding *linear ghost permissions*. Reminiscent of separation logic formulas, such permissions track the evolving state of a resource, but rely entirely on Rust for enforcing linearity.

Leveraging linear ghost permissions, Verus provides replacements for unsafe Rust features that encode the respective safety conditions in their specifications. Unlike foundational tools capable of verifying unsafe Rust code directly, Verus requires programmers to refactor unsafe code to utilize the provided replacements. For example, Verus introduces the permissioned pointer type `PPtr<T>` as a safe replacement for raw heap pointers. Each `PPtr<T>` is accompanied by a ghost permission value of type `PermData<T>`, which is acquired alongside the pointer upon allocation. While `PPtr<T>` pointers are duplicable like Rust's raw pointers, the permission values are linear. Consequently, writing to a `PPtr<T>` pointer requires both a shared reference to the pointer itself and exclusive access to the corresponding `PermData<T>` ghost permission.

7 Conclusion and Future Work

7.1 Conclusion

Support for type invariants and ghost code in the Rust verifier Creusot empowers users to write more expressive specifications, improving the viability of verifying complex programs. The main contributions of this work can be summarized as follows:

- We introduced an innovative design for type invariants that ensures composability, in particular in the interaction with Creusot's prophetic encoding of mutable references. Although we did not present a comprehensive formal proof of soundness, we discussed the soundness of critical components. In particular, this work includes a correctness proof of the prophecy resolution algorithm, which is responsible for maintaining type invariants for mutable references.
- We implemented type invariants in Creusot and tested them through several case studies. Based on these case studies, we conducted experiments affirming the applicability of our design to real-world scenarios. While these studies underscored the expressive capabilities of type invariants, they also highlighted challenges in verifiability, which we addressed through targeted optimizations.
- We conducted a thorough analysis of Creusot's preexisting ghost code implementation. In particular, we mapped out the necessary conditions for the soundness of prophecies in ghost code. The existing implementation fell short of fully ensuring these conditions, prompting us to suggest and implement appropriate refinements.
- We integrated all but one of the proposed enhancements into Creusot. Furthermore, we tested how the usage of ghost code to instantiate existential quantifiers improves verifiability. Regarding Rust's iterators as a case study not only confirmed this hypothesis but also demonstrated the synergy between type invariants and ghost code.

7.2 Future Work

Validation of Type Invariant Definitions. In Section 3.3.3, we explained how contradictory user invariants can cause unsoundness. We stated the rules users must follow when defining invariants to uphold soundness. However, these rules are not yet validated by Creusot. Implementing such checks therefore remains an important future improvement.

Inference of Loop Invariants Preserving Type Invariants. When modifying a value with a non-trivial type invariant within a loop, users have to manually add a loop invariant stating that the loop preserves the type invariant. An interesting future improvement would be to automatically infer such loop invariants.

Specification of Functions Returning Reborrows. In Section 3.4.4, we discussed why prophecy resolution renders some programs not verifiable. In particular, Creusot currently does not support functions that return reborrowed references to values with invariants. Specifying such functions requires a new type of precondition that can impose conditions on how a returned reference is used in the caller. A mechanism similar to Prusti's `assert_on_expiry` is imaginable and illustrated in Figure 7.1.

```
#[requires_on_expiry((*x).a@ + (^result)@ == 10)]
fn project(x: &mut SumTo10) -> &mut i32 {
    &mut x.a // (*x).a <- ^result
    // the invariant of x is provable using the precondition
}
```

Figure 7.1: Hypothetical specification for a function returning a reborrow.

Trusted Type Invariants. For some types, Rust's type system provides additional guarantees that can not yet be leveraged in verification with Creusot. For instance, an array `a` of type `[u8; 10]` always has length 10 and thus the property `a@.len() == 10` must always be true. While this property can be expressed as a type invariant, it is different from the type invariants considered so far. Since the property is enforced by Rust itself and cannot be broken, it can be assumed as an axiom for all values of the type and users should never be required to prove it. This is similar to Creusot's concept of trusted specifications, which generate no proof obligations. Other types that would benefit from trusted invariants are mutable slices, such as the slice `s` of type `&mut [u8]`. Since Rust only allows changing the elements referenced by `s` but not its length, we could define the trusted invariant `(^s)@.len() == (*s)@.len()`.

Extensionality of Mutable References. As discussed in Section 4.3.3, the encoding of mutable references ought to be changed to prevent unsound comparisons of mutable references in ghost code. One potential approach is the addition of a third opaque value to each reference, making it impossible to exploit extensionality. However, there remain open questions concerning the correct behavior for reborrows and the impact on verifiability.

Linear Ghost Code. An intriguing future research direction for Creusot involves supporting ghost code in which Rust's ownership and borrowing paradigm is enforced.

Such ghost code would enable linear ghost permissions, which Verus [Lat+23] has shown useful in specifying resource-managing types. It would be compelling to explore to what extent linear ghost code can mimic the concept of ghost resources in the Iris [Jun+18] separation logic. This could pave the way not only for specifying types with interior mutability, as demonstrated by Verus, but also for the verification of concurrent programs. Furthermore, it merits contemplation whether Creusot's type invariants could emulate aspects of Iris' concept of invariants.

List of Figures

List of Tables

Bibliography

[Ast+19] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. "Leveraging Rust types for modular specification and verification." In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–30.

[Bar+22] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, et al. "cvc5: A versatile and industrial-strength SMT solver." In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2022, pp. 415–442.

[Coh+09] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. "VCC: A practical system for verifying concurrent C." In: *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*. Springer. 2009, pp. 23–42.

[Con+18] S. Conchon, A. Coquereau, M. Iguernlala, and A. Mebsout. "Alt-Ergo 2.2." In: *SMT Workshop: International Workshop on Satisfiability Modulo Theories*. 2018.

[CPN98] D. G. Clarke, J. M. Potter, and J. Noble. "Ownership types for flexible alias protection." In: *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 1998, pp. 48–64.

[Cre23] Creusot Contributors. *Creusot GitHub Repository (evaluation branch)*. 2023. URL: `https://github.com/voidc/creusot/tree/evaluation` (visited on 11/13/2023).

[DB08] L. De Moura and N. Bjørner. "Z3: An efficient SMT solver." In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[Dij+76] E. W. Dijkstra, E. W. Dijkstra, E. W. Dijkstra, and E. W. Dijkstra. *A discipline of programming*. Vol. 613924118. prentice-hall Englewood Cliffs, 1976.

[DJ23] X. Denis and J.-H. 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.

[DJM22] X. Denis, J.-H. Jourdan, and C. Marché. "Creusot: a foundry for the deductive verification of rust programs." In: *International Conference on Formal Engineering Methods*. Springer. 2022, pp. 90–105.

[DMM18] S. Dailler, C. Marché, and Y. Moy. "Lightweight interactive proving inside an automatic program verifier." In: *arXiv preprint arXiv:1811.10814* (2018).

[FGP16] J.-C. Filliâtre, L. Gondelman, and A. Paskevich. "The spirit of ghost code." In: *Formal Methods in System Design* 48 (2016), pp. 152–174.

[FP13] J.-C. Filliâtre and A. Paskevich. "Why3—where programs meet provers." In: *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*. Springer. 2013, pp. 125–128.

[FP91] T. Freeman and F. Pfenning. "Refinement types for ML." In: *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 1991, pp. 268–277.

[Gäh+23] L. Gäher, M. Sammler, R. Jung, R. Krebbers, and D. Dreyer. "RefinedRust: High-Assurance Verification of Rust Programs." unpublished. 2023.

[Gil21] F. Gilcher. *Ferrocene Part 3: The Road to Rust in mission- and safety-critical*. 2021. URL: `https://ferrous-systems.com/blog/ferrocene-update-three-the-road/` (visited on 11/11/2023).

[Hay23] J. Hayeß. "Verifying the Rust Runtime of Lingua Franca." MA thesis. 2023.

[Hoa69] C. A. R. Hoare. "An axiomatic basis for computer programming." In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[Jun+17] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. "RustBelt: Securing the foundations of the Rust programming language." In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–34.

[Jun+18] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic." In: *Journal of Functional Programming* 28 (2018), e20.

[Jun+21] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. "Safe systems programming in Rust." In: *Communications of the ACM* 64.4 (2021), pp. 144–152.

[Lat+23] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel. "Verus: Verifying rust programs using linear ghost types." In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA1 (2023), pp. 286–315.

[Leh+23] N. Lehmann, A. T. Geller, N. Vazou, and R. Jhala. "Flux: Liquid types for rust." In: *Proceedings of the ACM on Programming Languages* 7.PLDI (2023), pp. 1533–1557.

[Lei10] K. R. M. Leino. "Dafny: An automatic program verifier for functional correctness." In: *International conference on logic for programming artificial intelligence and reasoning*. Springer. 2010, pp. 348–370.

[LM10] K. R. M. Leino and M. Moskal. "VACID-0: Verification of Ample Correctness of Invariants of Data-structures, Edition 0." In: *Proceedings of Tools and Experiments Workshop at VSTTE*. 2010.

[Mat+22] Y. Matsushita, X. Denis, J.-H. Jourdan, and D. Dreyer. "RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code." In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022, pp. 841–856.

[MK14] N. D. Matsakis and F. S. Klock. "The rust language." In: *ACM SIGAda Ada Letters* 34.3 (2014), pp. 103–104.

[Mos09] M. Moskal. "Programming with triggers." In: *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. 2009, pp. 20–29.

[Rey83] J. C. Reynolds. "Types, abstraction and parametric polymorphism." In: *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres*. 1983, pp. 513–523.

[Sam+21] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg. "RefinedC: Automating the foundational verification of C code with refined ownership types." In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 158–174.

[XP99] H. Xi and F. Pfenning. "Dependent types in practical programming." In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1999, pp. 214–227.