Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Rust2Viper: Building a Static Verifier for Rust

Master Thesis

Florian Hahn

April 7, 2016

Advisors: Prof. Dr. Peter Müller and Dr. Alexander J. Summers
Department of Computer Science, ETH Zürich

Contents

1 Introduction **1**
1.1 Motivating Example . . . 2

2 Background **4**
2.1 The Viper Verification Infrastructure . . . 4
2.2 Rust . . . 9

3 Verifying Rust Programs **15**
3.1 From Mutability to Permissions . . . 15
3.2 Structs and Moves . . . 16
3.3 Borrows . . . 21
3.4 Enums . . . 33
3.5 Loops & Structural Specification Using Magic Wands . . . 37
3.6 Encoding Rust's Integer and Boolean Types . . . 43
3.7 Pure Functions . . . 45
3.8 Unsafe Rust . . . 47

4 Functional Specification **54**
4.1 Structs and Enums in Assertions . . . 55
4.2 Interaction with Magic Wands . . . 57

5 Technical Details **64**
5.1 Rust Compiler Plugin . . . 64
5.2 Viper as a Service . . . 65
5.3 Mapping Error Messages from Silver to Rust . . . 65

6 Evaluation **68**
6.1 Supported Subset of Rust . . . 68
6.2 Performance of Generated Silver Code . . . 69

7 Conclusion **70**
7.1 Related Work . . . 70
7.2 Future Work . . . 71
7.3 Acknowledgements . . . 71

A Rust Examples **73**

B Supported Subset of Rust **76**

Bibliography **77**

Chapter 1

Introduction

Correctness is an important property of a software system. During the last couple of years, software systems have affected more and more aspects of our lives. Increasingly, software controls systems that interact with humans. When a software system controls critical components of an airplane or a self-driving car, the confidence in that system must be very high in order to offset the risks. Testing is a well-known and widely-used dynamic technique to ensure software quality, but it cannot offer strong guarantees because for non-trivial programs the number of possible inputs is too large to cover with tests. Software verification on the other hand can be used to verify that the runtime behavior of a program matches the specified behavior under all circumstances. A common approach to automated software verification is to use code contracts to specify functional properties of programs in combination with a verifier that can prove those functional properties.

Over the last decade, there has been quite some progress in the area of software verification, especially due to improvements in computing power and in automatic theorem provers. Also over the last few years, automatic verifiers for various languages emerged, for example Microsoft's Spec# for C# [1], Autoproof for Eiffel [28], Verifast for C and Java [17] and Dafny [18].

A recent development in this area is the Viper verification infrastructure developed at ETH. It provides an intermediate language, Silver, for software verification based on implicit dynamic frames [22] and fractional permissions [6]. The goal of the Viper project is to provide a solid verification infrastructure for developers of software verification tools. At the moment, there exist front-ends for languages such as Java [5], Chalice [23] and Scala [8], which generate Silver code in order to verify functional properties of programs in those languages using the Viper verification infrastructure.

In this project, we aim to build a tool for verifying functional properties of Rust programs based on the Viper verification infrastructure. We identified Rust as a good match for supporting verification for two reasons. First, Rust comes with a powerful type system centered around ownership and borrowing, which will be introduced in the next section in more detail. This means types in Rust capture a lot of information about the program. We plan to use this information during the translation to generate permission specification automatically, which reduces the amount of specification programmers have to provide drastically. Second, Rust is already focused on safety. For example, Rust's type system prevents common memory errors by compile-time checks. Those compile-time

checks mean that Rust programmers must already make an additional effort to make the compiler accept their programs. Therefore they trade some convenience for safety. Because of this we think there are at least some Rust programmers interested in verifying functional properties of their programs in order to increase the guarantees proved statically and there is hope that they are willing to spend additional time on adding contracts for their programs.

1.1 Motivating Example

In this section we show an example program with functional specification that highlights some of the features of Rust we plan to support. Listing 1.1 shows an enum representing a linked list and some functions for linked lists with contracts. It contains functions to calculate the length of a list, access an element of a list as well as a function to increment all elements of a list. The functions `length` and `item_at` are *pure*, which means they do not have any side effects and can be used in contracts. The `item_at` function has additional preconditions that state that it can only be called with non-empty lists and the index must be valid. Finally the `increment` function is not pure, because it modifies the list. In its postconditions `length` and `item_at` are used to specify the behavior of `increment`: the length of the list stays the same, but all elements are incremented. Note that no permission-related specification is needed throughout the contracts in this example, as we intend to generate permission-related specification automatically from the Rust source program.

```
1  pub enum List {
2      Node(i32, Box<List>),
3      Nil
4  }
5  #[pure_fn]
6  pub fn length(l: &List) -> i32 {
7      match *l {
8          List::Node(_, ref tail) => 1 + length(tail),
9          List::Nil => 0
10     }
11 }
12 #[pure_fn]
13 #[precond="length(l) > 0"]
14 #[precond="i >= 0 && i < length(l)"]
15 pub fn item_at(l: &List, i: i32) -> i32 {
16     match *l {
17         List::Node(ref c, ref tail) if i == 0 => *c,
18         List::Node(ref c, ref tail) => item_at(tail, i-1),
19         List::Nil => panic!()
20     }
21 }
22 #[postcond="length(l) == old(length(l) )"]
23 #[postcond="forall i: Int :: i >= 0 && i < length(l) ==>
24                   item_at(l, i) == old(item_at(l, i)) + v"]
25 pub fn increment(l: &mut List, v: i32) {
26     match *l {
27         List::Node(ref mut c, ref mut t) => {
28             *c = *c + v;
29             increment(t, v);
30         }
31         List::Nil => {}
32     };
33 }
```

Listing 1.1: A recursive list implementation in Rust with contracts

Chapter 2

Background

2.1 The Viper Verification Infrastructure

In this section we briefly introduce Viper, a verification infrastructure for permission-based reasoning developed at ETH. It consists of an intermediate language, Silver, that includes a flexible permission model, and two back-end verifiers for Silver: Silicon and Carbon. Silicon uses symbolic execution while Carbon uses verification condition generation. In this section, we mainly focus on the intermediate language Silver and the most important features used in our encoding of Rust programs. A more in-depth introduction to Viper can be found in [20].

Silver is an imperative, object-based language with built in support for a heap and reasoning about permissions to access fields of objects on that heap based on implicit dynamic frames [22]. A Silver program consists of pure functions, imperative methods, global field declarations, predicate definitions and mathematical domains.

Silver's object model is quite simple. Silver objects do not belong to any class, but they potentially have access to all fields declared in the program. More specifically, permissions are used to model which locations can be accessed on the heap.

2.1.1 Permissions

Permissions are a central concept of Silver. They are used to control access to the heap and during the verification of a Silver program, a set of permissions is held at each program point. This set of permissions specifies which parts of the heap can be accessed at a certain program point. During the verification of a Silver program, permissions to certain locations on the heap can change, e.g. additional permissions can be gained or some permissions can be lost. Permissions are tied to field locations on the heap and the amount of permission is specified using fractions of a maximum amount of permission. In Silver, *write* is used to refer to this maximum amount and a location on the heap can only be modified while holding *write* permissions to that location. At any given program point, at most *write* permissions can be held for each location of the heap. All other strictly positive fractions that are less than *write* are referred to as *read* permissions. In order to read a location on the heap, holding some *read* permissions is sufficient. Silver provides an access predicate *acc* which can be used to specify permissions to a single location in assertions, e.g $acc(x.val, write)$ states that *write* permissions to the location *x.val* on the

heap must be held. Assertions that contain access predicates are called *non-pure* while assertions without access predicates are called *pure*. Silver supports the following types of permission expressions:

- a positive fraction p for which $none \leq p \leq write$, where *none* means no permissions
- arithmetic expressions over two permissions: addition, subtraction, division and multiplication
- the special value *wildcard*, which denotes some unspecified positive permission amount

Functions and methods in Silver have to explicitly state to which part of the heap they require access. When calling a method, the permissions required by the method's pre-conditions are transferred from the caller to the called method. In a similar fashion, the permissions stated in the post-conditions of the called method are transferred back to the caller after the call. A major advantage of using permissions during verification is that permissions can be used to restrict modifications of locations on the heap. While some permissions to a location are held, the value stored at a location can be reasoned about sequentially, i.e. as long as some permissions are held, it is safe to assume that the value cannot be changed from outside the current function, e.g. by another thread. Because of that, proving that an assertion is not affected by a method call (framing) is straight forward. As long as the caller of a method retains some positive amount of permission to a location, it can assume that the value of that location cannot be changed by the called method.

The example in Listing 2.1 shows some of the core features mentioned above. It contains a `get` method, that requires read access to `x.val` and specifies that it returns `x.val`. In the method `client`, a new object with a field `val` is created on the heap and then passed to `get`. After creating an object with `new`, we have *write* permissions to all specified fields. When `get` is called, 1/2 of the permissions to `x.val` are transferred to `get`. After the call, those permissions are returned again. Note that by only requiring *read* permissions in the contract of `get`, `client` retains a positive amount of permission to `x.val` and can assert that it cannot be changed by the call.

Silver also provides statements to add or remove permissions from the program state in method bodies. Additional permissions can be gained with the *inhale* statement. `inhale A` assumes all pure assertions in A and adds the permissions specified by the access predicates in A. The *exhale* statement can be used to remove permissions. `exhale A` asserts all pure assertions in A and removes the permissions specified with access predicates in A. With those two statements, permissions can be modified liberally, but this flexibility comes with a price. Using those statements may cause inconsistencies which allow the verifiers to assert *false*, which in most cases is undesirable. One way to cause such an inconsistency is by violating the fundamental assumption that the amount of permission to a location on the heap is at most *write*. Consider the example in Listing 2.2. More than *write* permissions are inhaled which means any assertion holds after that point.

2.1.2 Predicates

There is one important area not covered by the permission system discussed above, namely unbounded data structures on the heap. So far we have only shown how the access predicate *acc* can be used to specify permissions to a single location. In order to support unbounded data structures we need additional tools. Recursive predicates

```
1  field val: Int
2
3  method get(x: Ref) returns (res: Int)
4      requires acc(x.val, 1/2)
5      ensures acc(x.val, 1/2)
6      ensures res == old(x.val)
7  {
8      res := x.val
9  }
10 method client() {
11     var x: Ref
12     x := new(val)
13     x.val := 10
14
15     var r: Int
16     r := get(x)
17
18     assert r == x.val
19 }
```

Listing 2.1: Simple Silver example

```
1 field x: Int
2 method test(){
3     var ref1: Ref
4     inhale acc(ref1.x, write)
5     inhale acc(ref1.x, write)
6     assert false
7 }
```

Listing 2.2: Silver program using inhale incorrectly

are a well known tool from separation logic [24] that Silver supports in order to express unbounded data structures. A predicate in Silver consists of a list of arguments and an optional body. The body consists of a single, self-framing assertion. An assertion is called self-framing in Silver when the assertion contains appropriate permissions to all fields used in it. For example, if a field is accessed in a self-framing assertion, there must be an appropriate *acc* predicate giving permissions to the field in the assertion.

Instances of predicates can be used in assertions and are handled similarly to permissions. In particular, instances of predicates can be used as arguments for access predicates. For example, consider the predicate *Tree* which describes a node in a binary tree in Listing 2.3. It gives access to a field containing the value of the node as well as access to the left and right children. When the children are not *null*, they in turn point to instances of the *Tree* predicate, which means the children are instances of the *Tree* predicate themselves. Note that `t.left` and `t.right` cannot alias, because the predicate `Tree` requires write permissions to both of them.

In Silver there are two basic operations on predicates, *unfold* and *fold*. The *unfold* operation exchanges the predicate for its body. This means that after unfolding a predicate instance, the assertions of the predicate's body can be used, but access to the predicate instance itself is lost. The *fold* operation does exactly the opposite, it exchanges the predicate's body for an instance of the predicate. In order to avoid unfolding recursive defini-

```
predicate Tree(t: Ref) {
    acc(t.val, write) &&
    acc(t.left, write) &&
    acc(t.right, write) &&
    (t.left != null ==> acc(Tree(t.left), write)) &&
    (t.right != null ==> acc(Tree(t.right), write))
}
```

Listing 2.3: Silver predicate modeling a binary tree

tions indefinitely, those operations have to be specified manually. This way, the verifiers only have to inspect data structures up to the depth at which the program to be verified has explicitly inspected the corresponding data structure [16]. While *unfold* and *fold* are statements in Silver, there also are *unfolding* and *folding* expressions which can be used to temporarily fold or unfold a predicate instance when evaluating an expression. For example `unfolding acc(Tree(x), write) in x.val == 10` unfolds the instance of the *Tree* predicate, then evaluates the inner expression and folds the predicate predicate instance `Tree(x)` again.

2.1.3 Mathematical Domains

Domains in Silver are used to encode first order theories. A domain introduces a new type and contains uninterpreted functions, as well as axioms for those functions. The domain *Natural* below highlights the features of domains in Silver. It contains `zero`, `succ` and `add` functions to model natural numbers, as well as two Silver axioms corresponding to the addition axiom of the Peano axioms.

```
domain Natural {
  function zero(): Natural
  function succ(n: Natural): Natural
  function add(x: Natural, y: Natural): Natural

  axiom addition_zero {
    forall x: Natural :: add(x, zero()) == x
  }
  axiom addition_nonzero {
    forall x: Natural, y: Natural ::
        add(x, succ(y)) == succ(add(x, y))
  }
}
```

Listing 2.4: Silver domain modeling parts of the Peano axioms

2.1.4 Magic Wands

Magic wands, or separating implications, are a key component of separation logic and where first introduced in the initial paper about separation logic [21]. The semantics of the connectives are defined as:

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

Here, σ' and σ' refer to partial program states consisting of a heap and a store function mapping local variables to their values. With $\sigma' \perp \sigma$ we refer to two compatible states σ' and σ, meaning they neither disagree on the value of any local variable nor do they require access to the same heap locations. We use $\sigma \uplus \sigma'$ to denote the combination of two compatible states. Intuitively, a magic wand $A \mathrel{-\!\!*} B$ can be used to get to a state where B holds by providing a state σ' where A holds. Magic wands can be used to specify future additions to the state and therefore they can also be used to describe partial versions of data structures. While most verifiers for permission-based logics do not support magic wands, Schwerhoff and Summers added support for magic wands to the Silver language and the Silicon verifier [25]. In the following paragraphs we briefly describe their approach.

In Silver, a magic wand $P \mathrel{-\!\!*} Q$ can be used by using the ghost statement *apply*. This statement removes the assertions specified on the left side of the wand from the current program state and adds the assertions on the right side. In terms of Silver ghost operations, applying a wand $P \mathrel{-\!\!*} Q$ in a program state where P holds is equivalent to:

$$exhale\ P \mathrel{-\!\!*} Q;\ \ exhale\ P;\ \ inhale\ Q$$

In order to apply a magic wand, an instance of the magic wand is needed. Similar to predicate instances, magic wand instances must be created using ghost operations. In Silver, magic wand instances can be created with the *package* statement. When creating a magic wand instance, the state σ in the semantics above must belong to the magic wand instance. This state is called *footprint* of the wand and when a wand is packaged, a state σ must be computed, that guarantees the wand's semantics. This footprint is then removed from the current state and the wand representing it is added.

Listing 2.5 shows the syntax for the aforementioned ghost operations. The *package* statement in the example creates a magic wand instance that given a state where *write* permissions to `x.f` are held, can be used to acquire *write* permissions to `x.y`. When the wand is packaged, *write* permissions to `x.y` are removed from the program state. The magic wand instance is then applied in line 3. When the wand is applied, *write* permissions to `x.f` are removed and added to `x.y`.

```
1 method foo(x: Ref)
2   requires acc(x.f, write) && acc(x.y, write)
3   ensures acc(x.f, write) && acc(x.y, write)
4 {
5   package acc(x.f, write) --* acc(x.y, write)
6
7   apply acc(x.f, write) --* acc(x.y, write)
8 }
```

Listing 2.5: Silver ghost operations for creating and applying a magic wand

Besides the *package* and *apply* ghost operations, there also are magic wand related ghost operations that can be used on the right side of a magic wand when packaging a new magic wand instance. In this thesis, we will make use of the *applying* and *unfolding* ghost operations. Those ghost operations indicate that they should be applied during the footprint computation, not in the current state. The *applying* ghost operation allows applying a magic wand w' while packaging another wand w. The method `e_applying`

in Listing 2.6 shows how *applying* can be used. In order to package the wand $w = acc(a.x) \mathrel{-\!\!*} acc(a.y)$, the wand $w' = acc(a.x)$ && $acc(b.x) \mathrel{-\!\!*} acc(a.y)$ has to be applied. But in order to apply this wand, access to $a.x$ is required. By using *applying*, the wand w' can be applied when packaging the wand w, because w must be applied in a state where permissions to access $a.x$ are held. The *folding* ghost operation on the other hand can be used to fold a prediacte while packaging a wand. The method `e_folding` in Listing 2.6 shows how *folding* can be used. When packaging the wand $acc(a.x) \mathrel{-\!\!*} acc(P(a)$, *folding* can be used to fold a predicate instance `P(a)`, because $acc(a.x) \mathrel{-\!\!*} acc(P(a)$ must be applied in a state where permissions to $a.x$ are held.

```
1  field x: Int
2  field y: Int
3
4  method e_applying(a: Ref, b: Ref)
5    requires acc(b.x)
6    requires acc(a.x) && acc(b.x) --* acc(a.y)
7    ensures acc(a.x) --* acc(a.y)
8  {
9    package acc(a.x) --*
10           applying (acc(a.x) && acc(b.x) --* acc(a.y)) in acc(a.y)
11 }
12
13 predicate P(self: Ref) {
14   acc(self.x)
15 }
16
17 method e_folding(a: Ref)
18   ensures acc(a.x) --* acc(P(a))
19 {
20   package acc(a.x) --* folding acc(P(a)) in acc(P(a))
21 }
```

Listing 2.6: Applying and Folding ghost operations for magic wands.

2.2 Rust

Rust is a modern general purpose programming language with the goal of being "safe, concurrent and practical" [12]. Since we present a first verifier for a subset of Rust in this thesis, it will contain plenty of Rust source code in examples. Therefore we briefly introduce the most important syntax constructs, as well as the concepts of Rust's type system that are important for the remainder of this thesis. For an in-depth introduction of Rust, we refer to *The Rust Programming Language* [12], an introductory book written by the Rust developers.

2.2.1 Syntax

Rust has two different composite data types: structs and enumerations. Structs define product types and are declared with the keyword *struct*. Structs are similar to structs in C: they contain a set of fields. Those fields can be accessed with the dot operator (e.g. `p.x`). For example, Listing 2.7 declares a type *Point*, which contains two integer fields: `x` and `y`.

```
1 struct Point {
2   x: i32,
3   y: i32,
4 }
```

Listing 2.7: Rust struct declaration with two integer fields

Enumerations, or enums, on the other hand define sum types and are declared with the keyword *enum*. An enum contains one or more variants. Each of those variants has a constructor which takes a fixed number of arguments with specified types. For example, Listing 2.8 shows an enum `OptionI32`, which declares two variants: `Some` and `None`. The `Some` constructor takes a single integer as an argument while `None` takes no arguments at all.

```
1 enum OptionI32 {
2   Some(i32),
3   None
4 }
```

Listing 2.8: Rust enum declaration with two variants

As for control structures, Rust comes with the "usual suspects": *if*, *while* and *return*, which behave similarly to their Java or C counterparts. Notable additional control structures are the *match* and *for* constructs. The *match* construct provides a way of pattern matching on enums. Listing 2.9 shows how pattern matching can be used to distinguish the variants of the `OptionI32` enum.

In contrast to languages like C or Java, some control structures in Rust are expressions, not statements. For example, *if* and *match* are expressions, which means they yield a result value. For blocks with multiple expressions, the last expression is used as the result value, similar to e.g. Scala.

```
1 // if x is of variant Some, use its value, otherwise use 0
2 match x {
3   Some(v) => v // result of this pattern is v
4   None    => 0 // result of this pattern is 0
5 }
```

Listing 2.9: Pattern matching in Rust

The *fn* keyword is used for function definitions. In Rust, functions take zero or more arguments and have either zero or one return type. Listing 2.10 shows two function declarations.

Local bindings can be created using the *let* keyword. Those local bindings are stored on the stack. By default, local bindings are immutable. In order to get mutable bindings, *let*

```
1 // a function without arguments or return value
2 fn foo() {}
3
4 // an identity function that takes and returns a value of type i32
5 fn id(x: i32) -> i32 {
6   x
7 }
```

Listing 2.10: Function declarations in Rust

mut has to be used.

```
1 // creates a struct Point on the stack and binds it to x;
2 // the content of x cannot be modified
3 let x = Point { x: 0, y: 0};
4
5 // create a binding y;
6 // the content of y can be modified
7 let mut y = 1;
```

Listing 2.11: Creation of new bindings in Rust

2.2.2 Affine types and ownership

Unique ownership is a very important feature in Rust which is used to achieve one of Rust's main goals: memory safety. Rust's unique ownership model is similar to affine type systems [2], where every variable can only be used once. Next, we describe how Rust handles ownership.

In Rust, variable bindings "own" the resource they are bound to. As mentioned above, Rust enforces unique ownership. In particular this means that Rust's type system ensures that there is exactly one owner for any given resource. This property is ensured by Rust's move semantics. Consider a location that stores some data, for example an instance of the struct *Point* in Listing 2.12. This location is the current owner of that data. When this location is assigned to another location or passed to a function by value, ownership of the data is transferred to the new location. The new location gains access to the data and the old location loses access. In particular, this means the owner of a resource knows it has exclusive access to the resource and that the resource cannot be changed without the owner's permission. This information is also useful when it comes to deallocating objects, as it allows deallocating resources when their owner goes out of scope.

Any uses of the original owner after a move are forbidden and result in a compile time error. In Rust, it is also possible to move parts of a data structure. For example, a field of a struct could be moved to a new location as in the last line in Listing 2.12. The struct instance containing the moved field is now referred to as *partially* moved. In this case, the other fields of the struct instance can still be used, but the partially moved struct instance cannot be used as a whole. In our example, this means `line.end` can still be accessed, while `line` as a whole cannot be used where a variable of type *Line* is expected. In Rust all composite types use move semantics by default.

```
// p1 owns Point { x: 0, y: 0 }
let p1 = Point { x: 0, y: 0 };

// Point { x: 0, y: 0 } is moved from p1 to p2
let p2 = p1;

// use of moved value p1, compile time error
let x = p1.x;

let line = Line { start: Point { x:0, y: 0 }, end: p2 };
// move field
let start = line.start;
```

Listing 2.12: Moves in Rust

Note that while unique ownership is the default behavior for composite types, Rust also provides a way to change from move to copy semantics, if that makes sense for the data type. Instead of moving the content, a deep copy of the content is performed. This means after the copy, both source and destination location can access their respective content. In order to get copy semantics for a struct or enum, all types used in the definition of the struct or enum must use copy semantics as well.

2.2.3 Borrows and Lifetimes

To complement the ownership system, Rust provides a way to share data without giving up ownership. The process is called borrowing and the rules described below are enforced by a sub-system of the Rust compiler, called *borrow checker*.

Rust allows sharing of data without moving (or copying) it through references by using the idea of region-based memory management as introduced by Grossman et al. [14] for the Cyclone programming language. The main idea is that each reference to some data has a corresponding lifetime (in Cyclone those are referred to as regions) which represents the scope in which the reference can be used to access the data. In order to prevent "use after free" errors, the lifetime of a reference cannot be greater than the lifetime of the content it points to.

In Rust there are two types of references: immutable and mutable ones. Creating an immutable reference to some data means the data is borrowed immutably for the lifetime of the reference. The reference can then be used to access the data in a read-only fashion. This in turn also means that the owner of the data is not allowed to modify the data for the lifetime of the reference. Because the data cannot be modified as long as there is a single immutable borrow active, it is allowed to have one or more immutable references to the same data. Rust does not require manual lifetime annotations inside function bodies because it uses the lexical scopes of *let* bindings as lifetimes. The example in Listing 2.13 highlights immutable borrows.

Mutable references on the other hand can also be used to modify the contents they point to. But in order to prevent concurrent and potentially conflicting updates, Rust restricts the use mutable references. At any given time, there can only be a single mutable reference pointing to the same data. For the lifetime of a mutable reference, the owner of the

```
let mut p1 = Point { x: 0, y: 0 };
{
  // ref1 borrows p1 immutably;
  // the lifetime of the borrow is equal to the lexical
  // scope of ref1
  let ref1 = &p1;

  // for the remainder of the block, we cannot use p1
  // to modify its contents, but we can use it for read access
  let x = p1.x;
  // ...

  // the borrow of p1 expires at the end of this block
}
// p1 can be used to mutate its contents again, because all
// borrows have expired
```

Listing 2.13: Immutable borrows in Rust with different lifetimes

data itself is not allowed to access the data in any way. In fact, handing out a mutable reference can be seen as temporary transfer of ownership from the owner to the mutable reference. After the lifetime of the reference expires, ownership is transferred back to the original owner. In the example shown in Listing 2.14, creating a mutable reference to `p1` means that while this reference is active, ownership of `p1`'s value is transferred to the reference `ref1`, which means `p1` cannot by used in any way as long as `ref1` is in scope.

```
let mut p1 = Point { x: 0, y: 0 };
{
  // ref1 borrows p1 mutably;
  let ref1 = &mut p1;

  // for the remainder of the block, we cannot use p1
  // ...

  // the borrow of p1 expires at the end of this block
}
// p1 can be used to mutate its contents again, because all
// borrows have expired
```

Listing 2.14: Mutable borrows in Rust

The Borrow Checker

In order to check Rust's borrow rules the borrow checker tracks information about borrows in terms of loans. The loans are collected using dataflow analysis. Table 2.1 shows the loan information gathered by the borrow checker for the function `foo` in Listing 3.9. For both borrows, a loan is created. The recorded information includes the path of the borrowed value, providing detailed information about the parts that are borrowed, the type of the loan as well as the start and end of the loan.

Currently Rust's borrow checker is clever enough to track borrows of struct fields precisely within the scope of the current function. In particular, mutable references to fields

id	path	type	start	end
0	x.*	immutable	line 3	line 4
1	x.b.*	mutable	line 6	line 7

Table 2.1: Simplified loan information for Listing 3.9

of structs or enums can be used to modify the contents of the fields. Other constructs of the language are not handled as precisely, for example the borrow checker does not track enough information to distinguish between borrows to disjoint locations of an array. Listing 2.15 shows the two cases.

```
1  let mut p1 = Point { x: 0, y: 0 };
2  // borrowing two different fields of a struct works
3  let ref1 = &mut p1.x;
4  let ref2 = &p1.y
5
6  // a mutable borrow to an element of the array x borrows
7  // the whole array
8  let mut x = [1, 2, 3];
9  let a = &mut x[0];
10 let b = &mut x[1]; // error, x already borrowed mutably
```

Listing 2.15: Special cases for the borrow checker

2.2.4 Lifetime parameters

As mentioned previously, Rust automatically infers appropriate lifetimes for references in function bodies. For function arguments and return values, manual lifetimes are required in general in order to allow checking the borrow rules in an intra-procedural fashion. But to reduce the annotation overhead, Rust allows lifetime parameters to be elided for some common use cases. The Rust compiler will then try to use distinct lifetimes for each parameter. For references as return values, the lifetime can only be elided if the function has exactly one reference parameter without lifetime annotation. Listing 2.16 shows some examples of lifetimes in function signatures. Note that using references in structs or enums always requires explicit lifetime annotations to ensure references to a struct cannot outlive references inside the struct or enum.

```
1  fn foo(x: &str) -> &str           // OK lifetimes elided
2  fn foo<'a>(x: &'a str) -> &'a str // expanded
3
4
5  fn bar(x: &str, y: &str) -> &str                  // illegal
6  fn bar<'a>(x: &'a str, y: &'a str) -> &'a str     // OK
7  fn bar<'a, 'b>(x: &'a str, y: &'b str) -> &'b str  // OK
```

Listing 2.16: Function declarations with and without lifetime parameters

Chapter 3

Verifying Rust Programs

In this section, we describe our approach for verifying Rust programs using the Viper verification infrastructure. It is based on translating Rust programs to Silver programs, which then can be verified with either Silicon or Carbon. Besides generating Silver code for Rust language constructs, we also generate appropriate predicates and permissions for Rust programs automatically. In the following sections we describe how Rust2Viper handles different constructs of Rust.

3.1 From Mutability to Permissions

A fundamental task when writing Silver programs by hand is choosing suitable permission amounts at various program points, e.g. in pre- and postconditions or when folding and unfolding predicates. In this section, we present how Rust's type information can be used to automatically choose appropriate permissions. For now we will only focus on distinguishing between *write* permissions and some amount of *read* permissions. We will also not account for borrows in this section, but we will discuss the effect of borrows on permissions in Sections 3.2.2 and 3.3. The basic idea is straightforward. References in Rust provide information about the mutability of their content at the type level. Mutable references can be used to modify the content they point to. This corresponds to *write* permissions to the content in Silver. Immutable references provide read-only access, which corresponds to some amount of *read* permissions. For value types, the mutability of the content is defined by the mutability of the binding that stores the value and a value type does not contain information about mutability on the type level. For example, mutable and immutable integer variables both have the same integer type, e.g. *i32*. The mutability of the variables is a property of the bindings themselves. A mutable binding corresponds to *write* permissions, while an immutable binding corresponds to some *read* permissions. More formally, we define a function *perm*, which takes a variable binding in a Rust function and returns permissions for the content of that variable. In order to define this function, we also need a function *typeof* which returns the Rust type for a given binding and a set *mutable_bindings*, which contains the names of the mutable bindings in a function. Now *perm(v)* can be defined as follows:

- *typeof(v) = &mut T* $\Rightarrow$ *perm(v) = write*
- *typeof(v) = &T* $\Rightarrow$ *perm(v) = read*
- *typeof(v) = T* $\wedge$ *v* $\in$ *mutable_bindings* $\Rightarrow$ *perm(v) = write*
- *typeof(v) = T* $\wedge$ *v* $\notin$ *mutable_bindings* $\Rightarrow$ *perm(v) = read*

3.2 Structs and Moves

3.2.1 Structs

As mentioned briefly in the section about Viper, objects in Silver do not belong to a particular class, but potentially have access to all declared fields. Which fields of an object can be accessed is defined by the permissions held at the current program point.

In Rust, on the other hand, each struct declaration defines a new data type. Each struct declaration consists of a list of fields and all instances of a struct type contain exactly those declared fields. Fields of structs can have arbitrary types, which means structs can be nested (the type of a field is another composite type) or recursive (the type of a field contains the type of the defining struct).

This means the encoding of structs must be flexible enough to support complex aggregations of structs, field accesses, moving of struct instances, as well as expressing type information of arguments and return values of Rust functions. The key idea for our encoding is centered about capabilities as described by Boyland et al. [7]. Each reference has a set of capabilities associated with it. Those capabilities describe what fields can be accessed with this reference and how they can be accessed (e.g. read or write access).

A type in Rust defines the capabilities each instance of that type provides. Reference types define the mutability of instances of that type, e.g. an immutable reference *&T* gives immutable access to an instance of *T*, while a mutable reference *&mut T* gives exclusive, mutable access. Value types such as *T* with move semantics provide exclusive access to their value. In addition, struct types provide the capabilities to access their field and enum types provide the capabilities to access their variants. For example, in order to access a field *f* on a struct instance *x*, the type of *x* must provide the capability to access that field. Note that the type information provides the maximum set of capabilities. Throughout a Rust program, those capabilities can be reduced, for example by borrowing or moving parts of a data structure. For now we focus on encoding this maximum set of capabilities. We will discuss the effect of moves and borrows on capabilities in Sections 3.2.2 and 3.3.

The concept of capabilities can be mapped to Silver quite nicely. In this section we focus on structs, but we will discuss enums in section Section 3.4 as well. Access permissions can be used to express capabilities of references. But before going into the details, we have to describe the data representation of structs in Silver first. Rust structs are mapped to Silver objects with appropriate fields. Listing 3.1 shows the translation of struct declarations in Rust to Silver fields. The listing on the left side contains Rust code, while the listing on the right contains the corresponding translation to Silver. Throughout this thesis, we use a similar structure for listings. For each field in the Rust struct a corresponding Silver field declaration is created. For integer fields, Silver's integer type *Int* is used. [1] Fields with composite types are translated to Silver references which point to another object representing an instance of the respective composite type. Note that in contrast to Rust, objects in Silver can only be created on the heap. This means all instances of structs are stored on the heap in Silver, whereas in Rust they could be stored on the stack

[1] Integers are represented in a slightly different manner in the final translation. We go into more detail in Section 3.6, but for now we use the simpler representation to avoid unnecessary overhead when explaining the basic idea.

or the heap. While the distinction where objects are created in memory matters when executing actual Rust programs, it is not important from a verification perspective.

```
1 struct Bar {
2     vb: i32
3 }
4
5 struct Foo {
6     b: Bar,
7     vf: i32,
8 }
```

```
1 //
2 field Bar__vb: Int
3
4
5
6
7 field Foo__b: Ref
8 field Foo__vf: Int
```

Listing 3.1: Translation of a struct declaration in Rust to Silver fields

Note that the data representation presented above only accounts for part of the Rust type information. In particular, for the field `Foo.b`, a Silver field of type *Ref* is created. But it does not include any information about the target of the reference. This is due to the fact that all objects in Silver belong to the same class. This information must be captured by the translation of struct types as well.

Now that we discussed the data representation, we take a closer look at how capabilities are encoded. The fact that a Silver reference `ref`, that points to an instance of the struct *Bar*, can be used to mutably access an integer field *vb* can be encoded by `acc(ref.vb, write)`. This implies that the capabilities of a struct type can be modelled using a conjunction of access predicates for each field. Rust2Viper uses predicates to bundle the capabilities of a type. For each Rust type, a corresponding predicate is created, which gives access to all fields of the struct. Moreover, with those predicates, Rust's mutability information can be encode as well, by using appropriate permissions for the predicate instances. For the structs from Listing 3.1, the predicates shown in Listing 3.2 are created. Each predicate takes a single reference as an argument and gives permissions to the proper fields. Note that by using a predicate for each type, the information that `Foo.b` has type *Bar* can be reflected concisely in the translation by using the predicate generated for *Bar*. Another possibility would be to unfold the body of `valid__Bar` in `valid__Foo`, but this would only work for non-recursive structs. For recursive types, it would be necessary to detect cycles in the type structure to avoid unfolding the definitions indefinitely.

```
1  predicate valid__Foo(self: Ref) {
2      acc(self.Foo__b) &&
3      acc(valid__Bar(self.Foo__b)) &&
4      acc(self.Foo__vf)
5  }
6
7  predicate valid__Bar(self: Ref) {
8      acc(self.Bar__vb) &&
9  }
```

Listing 3.2: Predicates encoding the capabilities of the structs Foo and Bar from Listing 3.1

Instances of Rust structs can now be encoded by a Silver object with the appropriate fields and a corresponding predicate instance. Consider Listing 3.3. In order to create a new instance of struct *Bar*, a new object with the required fields is created on the heap

first. Then the values are assigned to the fields and finally the predicate for the struct type is folded. The Silver reference `x` now points to an object representing the Rust struct instance from the example. In the example in Listing 3.3, the Rust variable `x` owns the created instance of the struct *Bar*. When folding the predicate, Rust2Viper has to use the correct permissions. For bindings that own an object, the permissions depend on the mutability of the binding. A mutable binding implies *write* permissions. That reflects the fact that the content of the binding can be modified in Silver. An immutable binding on the other hand implies read-only permissions.

```
1 let mut x = Bar { vb: 10 };
2
3
4 //
```

```
1 var x: Ref
2 x := new(Bar__vb)
3 x.Bar__vb := 10
4 fold acc(valid__Bar(x), write)
```

Listing 3.3: Creation of a struct instance

In the remainder of this section, we discuss how field accesses are handled with our encoding. In the following paragraphs, we use the *perm* function defined in Section 3.1, which either returns *read* or *write* permissions. At the moment, we do not focus on choosing suitable fractions for *read* permissions, because as long as we discuss isolated field accesses without borrows, the actual amount of *read* permissions is not important. We will discuss choosing suitable *read* permissions in Section 3.3. Whenever we have access to a struct instance x in the Rust program, we also hold an instance of the corresponding predicate with $perm(x)$ permissions in the Silver translation. We will first focus on structs and discuss enums in Section 3.4.

When translating an access to a field in a struct, the predicate for the struct has to be unfolded in order to gain access to the fields. After a field access, the predicate has to be folded again, to restore all capabilities. During the translation, Rust2Viper has to emit the required *fold* and *unfold* operations automatically. We use the following rule to generate *fold* and *unfold* statements for Rust statements. First consider a single field access of the form $x_0.x_1..x_{n-1}.x_n$. For such an access, we need to unfold the appropriate predicates for x_0, $x_0.x_1$ up to $x_0.x_1..x_{n-1}$ with permissions $p = perm(x_0)$. After the field access, all predicates are folded again in the reverse order. For statements containing multiple field accesses, we first collect all accesses and then generate *unfold* and *fold* statements for all unique sub-paths. For a statement containing three accesses $x.f$, $x.f.g$ and $x.f.h$, all unique sub-paths would be x, $x.f$, $x.f.g$ and $x.f.h$ and we would only fold and unfold the predicates for x and $x.f$ once. This is important in order to avoid unfolding the same predicate instance multiple times, which would fail, e.g. when $p = write$. The way p is chosen ensures that the emitted *unfold* and *fold* statements always succeed.

In the remainder of this section we use some examples to show how this rule is applied. First consider the example in Listing 3.4. When translating the access `x.vb` in the Rust program to Silver, Rust2Viper first emits an *unfold* statement with *write* permissions (because x is a mutable binding to a value type). After the assignment, a *fold* statement is emitted to restore the predicate.

Accesses to nested structs can be handled in a similar fashion. In the example below, `x.b.vb` is accessed in a read-only fashion. To fold and unfold the predicates, permission variables are used. Both $rd1$ and $rd2$ must be valid read permissions When there are no

```
1 let mut x = Bar { vb: 10 };
2
3 x.vb = 20;
```

```
1 unfold acc(valid__Bar(x), write)
2 x.vb := 20
3 fold acc(valid__Bar(x), write)
```

Listing 3.4: Translation of a statement updating a field

borrows, it is sufficient that $rd1 = rd2$, but once there are borrows the constraints on the permissions become more complicated. This will be discussed in detail in the Section 3.3.

```
1 let x = Foo { /* .. */ };
2
3
4
5 let y = x.b.vb;
```

```
1 unfold acc(valid__Foo(x), rd1)
2 unfold acc(valid__Bar(x.Foo__b), rd2)
3 y := x.Foo__b.Bar__vb
4 fold acc(valid__Bar(x.Foo__b), rd2)
5 fold acc(valid__Foo(x), rd1)
```

Listing 3.5: Translation of a statement reading a field

3.2.2 Moves

By default composite types have *move* semantics in Rust. This means unless a special *Copy* trait is implemented by a composite type, instances of that type are moved instead of copied when they are assigned to a new binding or passed as a function argument.

An important property when it comes to moves is the following. The borrow checker guarantees that when a binding x is moved, there are no other references to the value in scope. Likewise, as long as there are references to a binding x in scope, nothing can be moved into it. Hence there must be exclusive access to both the source and the target of a move.

There are two potentially ways to translate moves. The first possibility is to generate code that copies all the fields of a struct or enum. The second possibility is to just update the reference of the source to point to the moved object. At first, the second option seems to be the better one, as it results in more concise Silver code, but there is a technical reason making that option not viable in all cases. The problematic case manifests when moving to a mutable reference, which is illustrated in Listing 3.6. In this case, Rust2Viper cannot simply update the reference x with the address of the newly created instance of *Bar*, because it is a parameter, which cannot be modified in Silver. For this reason, Rust2Viper uses the second option only when the target of the move cannot be a function argument. For example, this is the case for new variables introduced with *let*. In this case, the mutability of the target of the move may change, e.g. by moving an immutable instance to a mutable binding. In this case we use *inhale* to upgrade the permissions of the struct instance to *write* permissions. This can be done because the Rust compiler ensures the source of the move is never used again until a new value is moved to it. The Silver translation in Listing 3.7 highlights this permission upgrade. For other moves the first option is used.

Of course it is also possible to move values stored in a struct. When parts of a struct instance are moved out, this struct instance loses the capability to access the moved parts. The Rust compiler will prevent subsequent uses of moved parts of struct instances. The changed capabilities are also reflected by Rust2Viper. When a field is moved out of

```
1 fn foo(x: &mut Bar) {
2   *x = Bar { .. };
3 }
```

Listing 3.6: Move to reference

a struct instance, the predicate of the struct is unfolded in order to gain access to its content. But once a field is moved out of a struct instance, the struct itself cannot be used any longer, so the predicate is not folded after the move operation. The example in Listing 3.7 illustrates that behavior. In the example, the field `foo.b` is moved to the binding `b`. In the Silver translation the `valid__Bar` predicate is unfolded but not folded again.

```
1  struct Bar {
2      f: i32
3  }
4  struct Foo {
5      b: Bar,
6      f: i32
7  }
8
9  let foo = Foo {..};
10 let mut b = foo.b;
```

```
1  //
2
3
4
5
6   unfold acc(valid__Foo(foo), rd)
7   var b: Ref
8   b := foo.Foo__b
9   inhale acc(valid__Bar(foo.Foo__b),
10             write-perm(valid__Bar(foo.Foo__b)))
```

Listing 3.7: Translation of Rust struct declaration to Silver fields

3.2.3 Function Arguments

When translating Rust functions to Silver, a central aspect is encoding the capabilities of the argument types. In Rust, function arguments can be passed either by value or by reference. When an argument is passed to a function, all capabilities of the argument's type are available at the beginning of the function. Reference types as arguments also convey additional information, namely that at the end of the functions, the capabilities are restored again. For example, consider a function that takes a mutable reference as an argument. In the function body, the capabilities to mutate the content of the reference could be lost temporarily by creating an immutable reference to the content. But as long as this reference expires at the end of the function body, the capabilities to modify the content of the argument are restored. The only exception are functions in which references escape the function body. This case is discussed in Section 3.3.3.

In our encoding, complex types are represented as Silver objects, which means they are passed as Silver references to methods and functions. The capabilities of the Silver objects are reflected by the predicates of the Rust types they represent. Those predicates can be used in pre- and postconditions of Silver methods and functions to specify the capabilities of arguments.

Before we go more into the technical details of the translation, we would like to briefly point out a difference between permissions in Silver and mutability of references in Rust to avoid any confusion later on. In Silver, permissions are tracked on a per-object basis. This means it is possible to have multiple references to the same object, and all those

references can be used to modify the object, as long as *write* permissions for that object are held at the current program point. In Rust this is not the case. At any given point in time, at most one mutable reference to the same object can exist. The Rust compiler does not reason about permissions on objects, but uses the additional restrictions of the borrow rules to statically ensure that every mutable borrow has exclusive access to its data at compile-time.

As the first step in our translation we introduce a permission argument *rd* to all methods. This parameter represents some *read* permissions (which means $none < rd < write$ must hold). It is used to specify the permissions of arguments that are immutable references and allows the caller to choose an appropriate permission amount when calling the method, depending on the permissions it holds at the time of the call. Listing 3.8 illustrates this translation step. When calling a Silver method in our encoding, choosing a suitable permission amount to pass as `rd` is crucial. It must be small enough that the caller holds at least `rd` permissions to the predicate instances reflecting the capabilities of the arguments for the call to succeed. One way to find a value satisfying this constraint is using Silver's *abstract read permissions* [15]. For abstract read permissions the verifier adds additional constraints depending on the use of permission variables. In Silver, *constraining* can be used to add such constraints for permission variables. For the example in Listing 3.8, the following constraints will be generated for `rd`: $rd < current_perm(valid__Foo(x1)) \wedge rd < current_perm(valid__Bar(b1))$ where $current_perm$ returns the permissions held for a predicate at the current program point. Throughout the following sections we will use abstract read permissions in all our examples. In Section 3.3.4 we will also discuss a static over-approximation that can be used to calculate appropriate *read* permissions without using abstract read permissions in many cases.

```
1  fn test(x1: &Foo,
2           b1: &Bar)
3  {}
4
5
6
7
8
9  test(x)
```

```
1  method test(x1: Ref, b1: Ref rd: Perm)
2    requires none < rd && rd < write
3    requires acc(valid__Foo(x), rd)
4    requires acc(valid__Bar(b1), rd)
5    ensures acc(valid__Foo(x), rd)
6    ensures acc(valid__Bar(b1), rd) {}
7
8  fresh rd
9  constraining (rd) { test(x1, x2, rd) }
```

Listing 3.8: Translation of a Rust function to a Silver method

3.3 Borrows

Borrows are another important feature of Rust that is interesting from a permission perspective for our translation. Borrows are references to some data, e.g. to a struct or parts of a struct. In Rust, those references also have an additional property, namely an associated lifetime. The lifetime of a borrow reflects the program region for which the reference is active and can be used to access the data. In this section we describe how this lifetime information is used to handle permissions to borrowed data.

3.3.1 Basic Idea

When dealing with borrows it helps to think about them in terms of capabilities, in particular how a borrow affects the capabilities of the borrowed content. When a value is borrowed, its capabilities are restricted according to Rust's borrow rules, for as long as the borrow is "alive" (which means it is active with respect to its lifetime). We use the examples in Listing 3.9 to highlight the effect of borrows on the capabilities of the borrowed content. The immutable borrow `ref1` of the mutable borrow `x` prevents modification of the value as long as `ref1` is active. In terms of capabilities, `x` lost the capabilities to modify its contents as long as `ref1` is active, even though the static type of `x` is still *&mut Foo*. This means as long as `ref1` is active, `x` cannot be used as *&mut Foo*, because this would allow modifications of `x` while there is another borrow to `x` alive. In fact, informally one could say the borrow `ref1` "downgrades" the current type of `x` to *&Foo*. However this fact is not captured directly by Rust's type system. Rust's type-checker will treat the type of `x` as *&mut Foo* throughout the whole body of the function `foo`. Those restrictions of capabilities are enforced in a separate step by the borrow checker.

When borrowing a field of a struct, the capabilities to modify that field are lost. But even more capabilities are lost. Borrowing a field of a struct also restricts the capabilities of the instance containing the field. In Listing 3.9, `x` cannot be used as *&mut Foo* while `ref2` is active, because without that restriction, `x` could be used to modify `x.b` to which an immutable borrow is alive. This shows that relying on only static type information is not enough when translating Rust programs to Silver; this is the main motivation to include the information Rust's borrow checker provides into our translation.

```
1  fn foo(x: &mut Foo) {
2    {
3        let ref1 = &*x;
4    }
5    {
6        let ref2 = &mut x.b;
7
8        // While ref2 is active, x cannot be used as &mut Foo.
9        // For this reason, the compiler rejects the function call below.
10       foo(x)
11   }
12 }
```

Listing 3.9: Borrows highlighting restriction on capabilities

The main idea of our translation of borrows is centered around the effect of borrows on the capabilities of references. In our translation there are cases in which the capabilities lost due to a borrow are modeled explicitly. One such case is borrowing a field of a struct. In this case, the top-level predicate instance of the struct has to be unfolded in order to get access to the field. By unfolding the predicate instance, permissions to the predicate instance are lost and they cannot be restored as long as the borrow to the field is active. Otherwise the reference could not be used to access the field. From a permission perspective we model this loss of capabilities by removing some permissions to the borrowed content. Those permissions are regained when the lifetime of the borrow ends.

Magic wands (introduced in Section 2.1.4) can be used to express this behavior quite naturally. The "borrowed-from" structure can be explicitly encoded by a magic wand. Such a wand allows restoring the permissions removed temporarily by a borrow by giving up permissions to the borrowed content. This explicit encoding encoding is a good fit in general, but in some cases it is not necessary to encode the "borrow-from" structure explicitly. For example, when returning a borrow from a function, the information what permissions need to be restored once the returned borrow goes out of scope must be encoded explicitly in the function's postconditions. For borrows that do not outlive function bodies on the other hand, this explicit encoding is not necessary in many cases.

3.3.2 Translating Function Local Borrows

In this section we describe our translation of function local borrows. By function local borrows we mean borrows that do not outlive the function body. First, we discuss how magic wands, in general, can be used to encode the "borrowed-from" structure explicitly. Later we will discuss when and how magic wands can be omitted for function local borrows. We will not discuss the handling of loops in this section. Those two cases are discussed in detail in Sections 3.3.3 and 3.5.

The most interesting aspect of function local borrows is that Rust2Viper has to ensure that the capabilities of the borrowed content are restored correctly after a borrow expires. First we will show how magic wands can be used to do that. Consider the borrows in Listing 3.9 and the corresponding Silver translation without any code to restore the original capabilities in Listing 3.10. Note that in our translation, when borrowing top-level structs (e.g. the first borrow `ref1` in the example) no ghost code is emitted to account for the loss of capabilities.

When parts of structs are borrowed (e.g. the second borrow `ref2` in Listing 3.9), the translation becomes more interesting. In order to access a field, the top-level predicate has to be unfolded first. After the new reference `ref2` to the field is created, the top-level predicate cannot be folded again until this borrow expires. If the top-level predicate were folded straight after the assignment, `ref2` would immediately lose the capabilities to access the field. Because the top-level predicate remains unfolded until the borrow expires, the loss of capabilities of `x` is reflected in our translation, by the fact that permissions to the top-level predicate instance were removed by the borrow.

Now we will address restoring the capabilities using magic wands. Consider a borrow of the form `x = &y`, where `x` and `y` are Rust expressions. The magic wand for this borrow has to specify that by giving up permissions to `x`, the permissions to `y` can be restored. We use the following rule to create such magic wands. Let P_{lhs} be the predicate instance describing the capabilities of the borrower `x`. For `ref2` in Listing 3.9, this would be $acc(valid__Bar(ref2), write)$. Let P_{rhs} be the top-level predicate instance of the value that is borrowed from, `y`. For `ref2` in Listing 3.9, this would be $acc(valid__Foo(x), write)$. Using P_{lhs} and P_{rhs}, the resulting wand is $P_{lhs} \mathrel{-\!\!*} P_{rhs}$. When packaging this wand, additional ghost operations may be required, e.g. when a field of a struct is borrowed, additional *folding* expressions are needed on the right side of the wand to restore the top-level predicate instance. A *folding* expression is generated for each predicate that was unfolded in order to access the field.

Now we apply those rules to Listing 3.9. For the first borrow, the trivial wand

$$acc(valid__Foo(ref1), rd) \mathrel{-\!\!*} acc(valid__Foo(x), rd)$$

can be created in order to restore the original capabilities of `x` when `ref1` goes out of scope. For the second borrow, a more interesting wand is created. This wand has to specify that by giving up write permissions to `valid__Bar(ref2)` we regain *write* permissions to `valid__Foo(x)`, which the following wand does:

$$acc(valid__Bar(ref2), write) \mathrel{-\!\!*} acc(valid__Foo(x), write).$$

Listing 3.11 shows the ghost operations required in Silver to create this magic wand and how it is applied in order to restore the original capabilities.

```
1  method foo(x: Ref, rd: Perm)
2    requires none < rd && rd < write
3    requires acc(valid__Foo(x), write)
4    ensures acc(valid__Foo(x), write)
5  {
6        var ref1: Ref
7        ref1 := x
8
9        unfold acc(valid__Foo(x), write)
10       var ref2: Ref
11       ref2 := x.Foo__b
12 }
```

Listing 3.10: Translation of Listing 3.9 without code for restoring capabilities after borrows expired

```
1 // borrow start
2 unfold acc(valid__Foo(x), write)
3 var ref2: Ref
4 ref2 := x.Foo__b
5 package acc(valid__Bar(ref2), write) --*
6         folding acc(valid__Foo(x), write) in acc(valid__Foo(x), write)
7
8 // borrow expires
9 apply acc(valid__Bar(ref2), write) --* acc(valid__Foo(x), write)
```

Listing 3.11: Creating and applying a magic wand to encode borrows

Note that while magic wands provide a nice formalism to handle function local borrows, they have a few disadvantages. First, using magic wands would clutter the generated Silver code with magic wand related ghost operations. Second, at the moment there exists an inconvenience related to magic wands in Silicon and Carbon. When using magic wands in combination with *write* permissions, they do not automatically track updates to fields that appear on the right side of the wand. Listing 3.12 and Listing 3.13 illustrate the problem. In Listing 3.12 a magic wand is used to restore *write* permissions to `Foo(x)`. Between packaging and applying the wand, `x.f` is updated. When the wand is applied however, the information that `x` was updated (`x.f := 10`) is lost. For this simple example,

```
1 field f: Int
2 predicate Foo(x: Ref) {
3     acc(x.f)
4 }
5 method foo(x: Ref)
6   requires acc(Foo(x), write)
7 {
8   unfold acc(Foo(x), write)
9   package acc(x.f, write) --* folding acc(Foo(x)) in acc(Foo(x), write)
10  x.f := 10
11  apply acc(x.f, write) --* acc(Foo(x), write)
12  assert unfolding acc(Foo(x), 1/2) in x.f == 10
13 }
```

Listing 3.12: Use of a magic wand to restore permissions

```
1 field f: Int
2 predicate Foo(x: Ref) {
3   acc(x.f)
4 }
5 method foo(x: Ref)
6   requires acc(Foo(x), write)
7 {
8   unfold acc(Foo(x), write)
9   x.f := 10
10  fold acc(Foo(x), write)
11  assert unfolding acc(Foo(x), 1/2) in x.f == 10
12 }
```

Listing 3.13: Use of fold to restore permissions

it would be possible to add additional assertions to the right side of the wand to preserve this information. But once there are mutable borrows to multiple fields, it becomes tricky to generate those assertions automatically. When using *fold* to restore the permission to `Foo(x)` as in Listing 3.13, the verifiers automatically preserve the required information.

However, function local borrows can be handled without magic wands, because as long as the borrow is limited to the scope of the current function, there is no need to explicitly encode the "borrowed-from" structure. In this case, *fold* statements can be used to restore the permissions. For function local borrows we therefore do not need to create magic wands explicitly and when a borrow goes out of scope, the permissions are restored using appropriate *fold* statements for the predicates on the right side of the wand. This can be done as long as there are no loops in the function that repeatedly create new borrows that outlive a loop body. For borrows that outlive a loop body, the information how to restore the permissions once a borrow goes out of scope must be preserved in a loop invariant, because the permissions cannot be restored at the end of the loop iteration. This information cannot be expressed in terms of *fold* statements. In Section 3.5 we discuss how loops can be handled using magic wands.

So far we have only discussed relatively simple borrow scenarios. In the following subsections we present some technical details that are used to handle more complicated borrow

scenarios.

Borrowing Data Multiple Times

When borrowing parts of a struct multiple times, calculating suitable permissions when unfolding the top-level predicate becomes more complicated than in the previous example. In the example in Listing 3.14, the field `x.b` is borrowed twice, and `x.b.c` is borrowed using one of the borrows of `x.b`. For field accesses, Rust2Viper has to unfold the corresponding predicate instances for the struct types, in order to get access to the fields in the Silver program. For each field access, a corresponding *unfold* statement is added and we have to chose suitable permissions for those *unfold* statements. Now let us take a closer look at the Silver encoding in Listing 3.14, with permissions in mind. The first observation is that when unfolding `valid__Foo(x)` for the first time, we cannot use all permissions we currently hold for `valid__Foo(x)`, because then the second *unfold* of `valid__Foo(x)` a would fail. For both $rd1$ and $rd2$ we have to choose fractions of rd, so that $rd1 + rd2 < rd$ holds. If this constraint is satisfied, there are enough permissions for the *unfold* statements to succeed and there are still permissions left for further borrows. This means an arbitrary number of immutable borrows to the same struct instance can be handled with our encoding. Those constraints can be encoded with Silver's *constraining* statement, similarly to the example in Listing 3.8.

The second observation is that the permissions used to unfold `valid__Bar(ref2)` depend on the permissions used to unfold `valid__Foo(x)` when creating `ref2`. This adds another constraint $rd3 < rd2$. Similar constraints can be added for arbitrary nesting of field accesses and borrows.

Instead of unfolding and folding predicate instances with some permissions for each field access, we could also use a more coarse-grained approach and only focus on the distinction between *read* and *write* permissions. We could only unfold the corresponding predicate instance for the first immutable field access and then keep the instance unfolded as long as there are no statements that need *write* permission to the field. For statements that require *write* permissions, we could unfold the predicate instance with the remaining permissions, to get *write* permissions to the field. When all mutable borrows have expired, we would fold the predicate instance with *read* permissions again. We decided not to use this approach, because it requires additional logic to check if a predicate instance was already unfolded for previous field accesses. Such a check could be implemented using the information the borrow checker provides. But this approach must also take to control flow into account. Consider a function with two possible flows of control. Along one program path, a field `x.f` is accessed, but along the other it is not. After the two paths are merged again, `x.f` is accessed again. Additional care would have to be taken to ensure the predicate for `x` is unfolded along both paths before the second access is reached.

While our system works well for immutable borrows, because it is always possible to generate smaller fractions of *read* permissions, this does not apply to mutable borrows. When a field of a struct is borrowed mutably in Rust, this field cannot be borrowed again as long as the first mutable borrow is active. Additional borrows to the same struct instance must be to different fields. These two properties are ensured by the borrow checker. In our encoding, when a field of a struct is borrowed mutably, the top-level predicate has to be unfolded with full *write* permissions. Before unfolding a predicate with *write* permissions, we have to check if there are other mutable borrows to different fields of the same struct instance alive in the current scope. For this check, the loan information the borrow

checker provides can be used. When a field `x.f` is borrowed mutably, we have to check if any of the active loans at the current program point involve other parts of `x`. If that is the case, the predicate was already unfolded with *write* permissions when the existing borrow was created, hence there are already *write* permissions to all fields of the struct and there is no need to unfold the predicate again. In this case it may be necessary to update the point where the predicate is folded again. If the lifetime of the existing borrow is smaller than the lifetime of the new borrow, the predicate has to be folded when the borrow with the longest lifetime goes out of scope.

```
1  fn foo(x: &Foo) {
2      let ref1 = &x.b;
3
4
5      let ref2 = &x.b;
6
7
8      let ref3 = &ref2.c;
9
10
11
12
13 }
```

```
1  //
2  unfold acc(valid__Foo(x), rd1)
3  ref1 := x.Foo__b
4
5  unfold acc(valid__Foo(x), rd2)
6  ref2 := x.Foo__b
7
8  unfold acc(valid__Bar(ref2), rd3)
9  ref3 := ref2.Bar__vb
10
11 fold acc(valid__Bar(ref2), rd3)
12 fold acc(valid__Foo(x), rd2)
13 fold acc(valid__Foo(x), rd1)
```

Listing 3.14: Borrowing parts of a struct multiple times

Control-Flow Dependent Borrows

The last interesting scenario for local borrows we discuss are borrows that depend on the control flow of the program. The Rust function in Listing 3.15 illustrates the problem. In this example, a mutable variable `ref1` is used to store two borrows. For the borrow of `x2.b` in the *then* branch this means the lifetime of the borrow is extended to the end of the function, when `ref1` goes out of scope. Without additional care, this example would fail with our encoding. When only looking at the lifetimes of the borrows both the borrow of `x1.b` and `x2.b` expire at the end of the function. This means Rust2Viper would emit code to fold both `valid__Foo(x1)` and `valid__Foo(x2)` and the verifier would rightfully reject that code, because `valid__Foo(x2)` only gets unfolded if `cond` is true. Therefore our encoding must contain additional ghost code to avoid folding predicates that were not unfolded. Rust2Viper introduces additional boolean variables to track which borrows actually happened along a possible execution path. The Silver encoding in Listing 3.15 includes a variable `ref1_a1` which is set to true if `x2.b` is borrowed in the *then* branch or to false otherwise. This variable is then used to conditionally fold the predicate when the borrow expires. Note that Silver's *perm* function could also be used to check if a predicate is folded with some permissions, which would be helpful in this case. In our encoding however, we avoid using *perm* in this case, because the solution with boolean variables is straight-forward and does not rely on *perm* which is not thoroughly tested at the moment.

3.3.3 Returning references from functions

Besides function local borrows, borrows can also escape function bodies. This is the case when returning references to any of the input arguments. Before going further into the details of our encoding of references as return values, we briefly discuss how lifetimes of

```
1  fn foo(x1: &Foo, x2: &Foo) {
2    let mut ref1 = &x1.b;
3    if cond {
4      ref1 = &x2.b;
5    }
6  }
7
8
9
10
11
12
13 //
```

```
1    var ref1_a1: Bool := false
2    unfold acc(valid__Foo(x1), rd)
3    var ref1: Ref
4    ref1 := x1.Foo__b
5    if (cond) {
6      unfold acc(valid__Foo(x2), rd)
7      ref1 := x2.Foo__b
8      ref1_a1 := true
9    }
10   if (ref1_a1) {
11     fold acc(valid__Foo(x2), rd)
12   }
13   fold acc(valid__Foo(x1), rd)
```

Listing 3.15: Control-flow dependent borrow

borrows interact with returning references in Rust.

When returning a reference from a function, the lifetime of this reference must not outlive the lifetime of the data it references. An interesting restriction of this rule is that it is not possible to return a reference to some data allocated in the function body unless this data also escapes the method, because otherwise the lifetime of the reference would be greater than the lifetime of the data. One common way for allocated data to escape a method is by storing it in a field of a parameter that is borrowed mutably.

Checking that the lifetimes of the function arguments and return value are compatible is also done by the borrow checker. For all further examples in this subsection, we will use explicit lifetime annotations for references to make things clearer, even though in some examples they could be elided. First, we discuss a simple Rust function that returns a reference, shown in Listing 3.16, with a focus on lifetimes and borrows. The function `borrow1` takes a reference with lifetime `'a` as parameter and returns a reference with lifetime `'a`. When checking `borrow1`, the borrow checker records that `r` escapes the function and later on checks if the lifetimes of all escaping borrows are compatible with the lifetime of the return value. When `borrow1` is called, its concrete argument `b` has to be borrowed for the function call. But now the fact that `borrow1` returns a reference with the same lifetime as its argument comes into play. As long as the returned reference is alive, it can be used to access the content it points to. This in turn means that the argument the returned reference points to must remain borrowed until the returned reference goes out of scope. Otherwise the rule that data cannot be modified while there is an active immutable borrow could not be enforced by the borrow checker. This means the borrow checker needs some information about the relationship between the references passed as function arguments and the reference returned by the function. The lifetime information provided in the function signature provides this information and allows the borrow checker to analyze each function modularly. All arguments with the same lifetime as the returned reference can potentially be returned from the function, hence all those arguments must remain borrowed until the returned reference goes out of scope. For the example in Listing 3.16, this means because the return value of `borrow1` has the same lifetime as the only argument, the argument `b` is borrowed as long as `ref1` is scope. In the case of the example, `b` is borrowed until the end of `client1`'s body after the call `borrow1`.

```
1 fn borrow1<'a>(r: &'a Bar) -> &'a Bar {
2     r
3 }
4
5 fn client1(b: &mut Bar) {
6   let ref1 = borrow1(b);
7 }
```

Listing 3.16: Simple example returning a reference from a function

The main observation we use in our encoding is the fact that when returning a reference, the arguments the reference may refer to are borrowed until the returned reference goes out of scope after a function call. In our encoding, we use magic wands to model this behavior. The general idea is straight-forward. When returning a reference from a function, a magic wand is created that reflects the fact that by giving up permissions to the returned reference the permissions to the borrowed arguments are re-gained. For the simple example above, the postconditions of `borrow1` have to state that the returned reference provides read access to an instance of *Bar* and by giving up access to this reference we re-gain read access to the argument `r`. Using the appropriate predicates, this can be achieved by the following wand:

$$acc(valid__Bar(res), rd) \mathrel{-\!\!*}, acc(valid__Bar(r), rd).$$

Now consider the complete Silver encoding of Listing 3.16 in Listing 3.17. It models two important aspects of `borrow1`:

1. the returned reference provides the capabilities of type *Bar* (line 4) and the capabilities of the argument `r` are removed temporarily (by requiring them, but not adding them to the postconditions)
2. a wand to restore `r`'s capabilities is created (line 8) and "returned" (line 5).

Note that in our encoding, the caller of a function returning a reference gives up some permissions to the borrowed arguments until the returned reference goes out of scope. The wand "returned" by the function is then used to restore the permissions. When it comes to applying the wand, the variables used in the wand have to be substituted by their counterparts in the calling function, i.e. `res` by `ref1` and `r` by `b` in Listing 3.17.

Next, we discuss how the temporary removal of permissions to borrowed arguments effects other borrows of those arguments after calling a function returning a reference. Temporarily giving up permissions to an immutably borrowed argument and transferring it into a magic wand does not cause any problems further on, because they are restored when the borrow expires. For subsequent borrows of the same borrowed argument we always generate smaller fractions of *read* permissions, so removing some permissions temporarily is not an issue in this case either. The last case to consider involves mutable borrows. If a mutable reference is returned from a function, the argument it was borrowed from cannot be used until the returned reference expires. This means Rust2Viper will never generate Silver code that tries to use the argument with *write* permissions as

```
1  method borrow1(r: Ref, rd: Perm) returns (res: Ref)
2    requires none < rd && rd < write
3    requires acc(valid__Bar(r), rd)
4    ensures acc(valid__Bar(res), rd)
5    ensures acc(valid__Bar(res), rd) --* acc(valid__Bar(r), rd)
6  {
7    res := r
8    package acc(valid__Bar(res), rd) --*  acc(valid__Bar(r), rd)
9  }
10
11 method client1(b: Ref, rd: Perm)
12   requires none < rd && rd < write
13   requires acc(valid__Bar(b), rd)
14   ensures acc(valid__Bar(b), rd)
15 {
16     var ref1: Ref
17     ref1 := borrow1(b, rd)
18
19     apply acc(valid__Bar(ref1), rd) --* acc(valid__Bar(b), rd)
20 }
```

Listing 3.17: Silver encoding of Listing 3.16

long as the returned borrow is active.

To conclude this section we show how returning control-flow dependent borrows and returning borrows of fields are handled. Listing 3.18 shows a function `borrow2` which borrows either the first or the second parameter, depending on a condition. In terms of escaping borrows, this means that either `r1` or `r2` can escape, so the borrow checker has to treat them both as escaping when checking `borrow2`. This also means that both arguments remain borrowed until the returned reference goes out of scope. To express this behavior, the created wand must also account for this over-approximation. This can be done by a wand which specifies that by giving up permissions to the returned reference, the permissions to all escaping arguments are restored. This way there is no need to differentiate between the possible return values. In our encoding, the following magic wand will be created for Listing 3.18:

$$acc(valid__Bar(res), rd) \mathrel{-\!\!*} acc(valid__Bar(r2), rd) \;\&\&\; acc(valid__Bar(r2), rd)$$

.

In the other example in the listing, a reference to a field of an argument is returned. In order to access the field `f.b`, the predicate for `f` must be unfolded and it has to remain unfolded as long as the returned reference is alive. In this case, magic wands again can be used to encode this behavior, namely with the following wand

$$acc(valid__Bar(res), rd) \mathrel{-\!\!*} acc(valid__Foo(f), rd)$$

.

```
1  fn borrow2<'a>(r1: &'a Bar, r2: &'a Bar) -> &'a Bar {
2      if r1.v > 10 {
3          r1
4      } else {
5          r2
6      }
7  }
8  fn borrow3<'a>(f: &'a Foo) -> &'a Bar {
9      &f.b
10 }
```

Listing 3.18: Two more interesting borrow scenarios

3.3.4 Over-Approximation of Permissions

So far we have used abstract read permissions and Silver's *constraining* to express read permissions. In this section, we present a system to statically choose fractions of *read* permissions that satisfy the constraints described in the previous sections. This static approach allows us to compare the performance of abstract read permissions in Silicon and Carbon to a static approach, where fractions are calculated manually. In the remainder of this section, we first outline our system to find suitable fractions and then compare the performance of that system to abstract read permissions.

To begin with, we summarize the constraints that need to be handled. Note that this approach excludes loops, which are discussed in Section 3.5. In the following we use *base_perm(x)* to refer to the amount of permission with which the predicate for x was held initially. In some sense, it refers to the total amount of permission available for x. For example, for a function argument x, $base_perm(x)$ refers to the amount of permission specified in the preconditions for the predicate of x. For fields, $base_perm(x.f)$ returns the amount of permission with which the predicate for $x.f$ was held initially. In our encoding, this equals the amount of permission used to unfold the top-level predicate of x in order to get access to $x.f$. Note that the permission amount used to unfold the top-level predicate depends on the use of the field. If it is used in a read-only fashion, the top-level predicate is unfolded with *read* permissions, even if the containing struct is mutable.

1. the sum of the permissions used to unfold the predicate for a single location l must be less than or equal to $base_perm(l)$
2. for field accesses such as $x.f$, $base_perm(x.f) \leq base_perm(x)$
3. the permissions passed to a function call must be less than or equal to the minimum permissions held for the function arguments at the call site

In the following, we will motivate the three constraints. When the first constraint is satisfied, there are enough permissions for all *unfold* statements in a function, which means all *unfold* statements will succeed. The second constraint ensures that there are enough permissions when a reference to a field is used. When a field is accessed, the top-level predicate is unfolded with some permissions p which means there is access to the predicate instance of the field with p permissions. This means for this field, at most p permissions can be used for the constraints involving this field. The third constraint ensures that there are enough permissions to the argument predicates at the function call site. The caller has to choose the amount depending in the permission it holds at the

current program point.

We use the example in Listing 3.19 to illustrate the constraints. To satisfy the first constraint, $rd1 + rd2 < rd \wedge rd3 < rd2$ must hold. To satisfy the second constraint, $rd1 < rd \wedge rd2 < rd \wedge rd3 < rd2$ must hold. The first two constraints ensure that there are enough permissions for all *unfold* statements in a function. To satisfy the third constraint, $rd4 < rd1 \wedge rd4 < rd3$ must hold. This constraint ensures that there are enough permissions to the predicates of all arguments of the called function at the call site.

```
1  method test(x: Ref)
2    requires acc(valid__Foo(x), rd)
3  {
4    var ref1: Ref
5    unfold acc(valid__Foo(x), rd1)
6    ref1 := x.Foo__b
7
8    var ref2: Perm
9    unfold acc(valid__Foo(x), rd2)
10   unfold acc(valid__Bar(x.Foo__b), rd3)
11   ref2 := x.Foo__b.Bar__c
12
13   call1(ref1, ref2, rd4)
14 }
```

Listing 3.19: Silver example to illustrate the constraints for our over-approximation.

The first constraint can be satisfied by dividing the base permissions of each type by the total number of *unfold* statements in the function. The total number of *unfold* statements for a type is an upper bound of the number of unfolds involving a single location of that type. We refer to this fraction of a type A as $frac(A)$. For the second constraint, we have to analyze the type structure. We choose the base permissions for each type according to the following rule: if type A appears as a field in type B we set $frac(A) = frac(A) \cdot frac(B)$. The predicate of a variable of type A can be unfolded at most $frac(A)$ times, which means constraints (1) and (2) are satisfied. Note that this system does not work for cyclic data structures [2] or when accessing multiple elements of a recursive data structure in a single function execution, and those two cases are not supported by our over-approximation. To satisfy the third constraint, we choose the minimum fraction among the fractions for all types. This is the minimum fraction that can be used to unfold any predicate, which means all predicates are held at least with this minimum permission.

Listing 3.20 shows the program from Listing 3.19 with the concrete fractions chosen using the rules described above.

Benchmark

To compare the performance of our encoding with the over-approximation described above and with abstract read permissions [15], we used a set of 59 test cases from the Rust2Viper test suite. They represent a wide array of use cases and also contain test cases that contain a lot of *unfold* and *fold* statements. For each test case, we measured the verification run-time using abstract read permissions and our over-approximation, with both

[2] Cyclic data structures cannot be created in the subset of Rust we support.

```
1 method test(x: Ref)
2   requires acc(valid__Foo(x), rd)
3 {
4   var ref1: Ref
5   unfold acc(valid__Foo(x), rd / 2)
6   ref1 := x.Foo__b
7
8   var ref2: Perm
9   unfold acc(valid__Foo(x), rd / 2 )
10  unfold acc(valid__Bar(x.Foo__b), rd / (2 * 1))
11  ref2 := x.Foo__b.Bar__c
12
13  call1(ref1, ref2, rd / (2 * 1)
14 }
```

Listing 3.20: Listing 3.19 with concrete fractions

Verifier	# abstract permissions	# approximation
Silicon	50	9
Carbon	24	35

Table 3.1: Number of test cases that verify faster with abstract read permissions and our over-approximation.

Silicon and Carbon. We report the average verification run-time of 5 runs for each test case. The run-time only includes the time the verifiers took to verify the Silver versions of the test cases. The verification results are the same for all test cases. The Silver version of the test cases were generated by Rust2Viper before the benchmark. The verification is done using Viper-as-a-Service (introduced in Section 5.2), which avoids restarting the JVM for each test case. The benchmarks were executed on a Linux machine with an Intel i5-4670S CPU with 3.10GHz and 8 GB of RAM and the run-times are measured using *perf*, which uses the performance counter subsystem of the Linux kernel.

Table 3.1 shows the number of test cases that verify faster with abstract read permissions and our over-approximation, for both Silicon and Carbon. In our benchmark, abstract read permissions are slower in most cases when using Carbon, but faster in most cases when using Silicon. Figure 3.1 shows the verification run-time of the test cases translated with abstract read permissions and our over-approximation using Silicon and Carbon. In most cases, the difference between abstract read permissions and our over-approximation is very small. Those results indicate that even though abstract read permissions add additional constraints the verifiers have to track, they do not have a big impact on the performance of the verification, because they are relatively simple compared to the other proof obligations from the input program. Interestingly, reasoning about abstract read permissions is slightly faster in most cases in Silicon.

3.4 Enums

Besides structs, enums can also be used to declare composite types in Rust. In this section we discuss our encoding for enums and pattern matching. We will discuss pattern matching in contracts in Chapter 4.

For our encoding of enums, we reuse the basic ideas we also used for structs and were

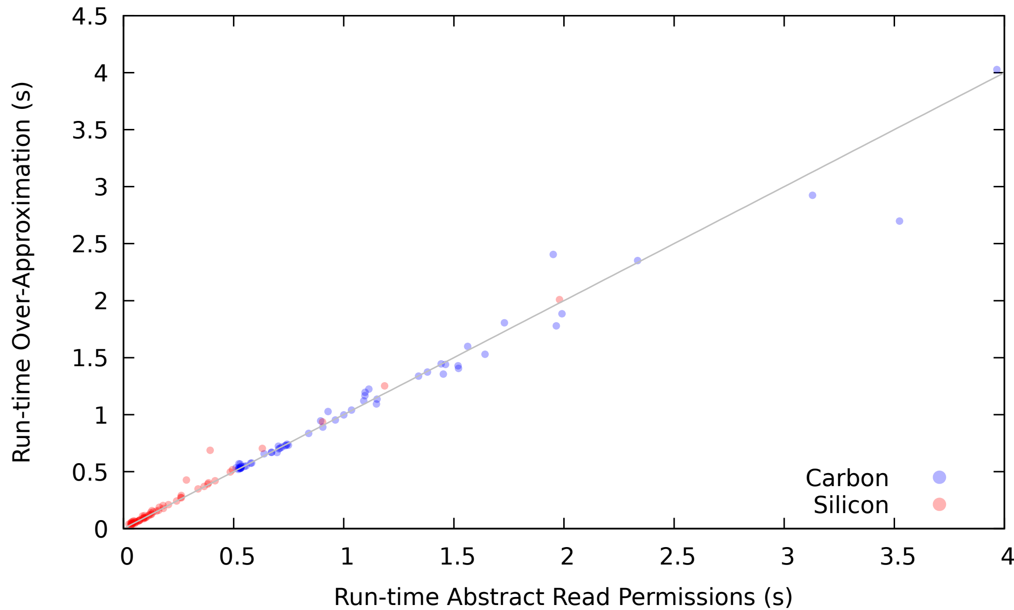

Figure 3.1: Verification run-time of programs translated with abstract read permissions and our over-approximation using Silicon and Carbon.

presented in Section 3.2.1. We use Silver objects to store the data associated with an enum. The major difference to structs is that for enumerations, the encoding also has to include information about which variant is stored in the object at the moment. In this section we will use the *MyOption* enum shown in Listing 3.21 as a running example. Instances of that enum either contain some struct *Bar* or nothing.

```
enum MyOption {
    MySome(Bar),
    MyNone
}
```

Listing 3.21: Rust enum declaration of an option type

To represent the different variants, we introduce a Silver domain for each enum. This domain contains functions and axioms to distinguish the different variants of an enum type. More specifically, a *unique* function for each enum variant is created. The value returned by a unique function is assumed to be different from all other unique functions with the same return type. The unique values of those functions are used to differentiate objects representing different enum variants. Additionally, an uninterpreted *variantOf* function is introduced. This function takes a reference as an argument and returns the value of the variant the reference points to. Finally another uninterpreted function *isEnum* is introduced. This function takes a reference as an argument and returns true if it points to an enum instance. For our running example, the domain shown in Listing 3.22 will be created. At the beginning the functions mentioned before are declared. They are followed by two axioms which describe their behavior. Note the expression { `variantOfMyOption(x)` } in the universal quantifiers. Those expressions define *triggers* for the quantifiers. Theorem provers typically use triggers to control how they instantiate universally quantified

expressions. They are a set of expressions the theorem prover must have discovered before the quantifier is instantiated and they are used to guide the theorem prover in the right direction to avoid infinite instantiations of quantifiers. For the axioms in the domains created for enums, the trigger is just `variantOfMyOption(x)` , which means when the theorem prover discovers such an expression (modulo equality), the universal quantifiers are instantiated.

```
domain MyOption {
  unique function MyOption__MySome(): MyOption
  unique function MyOption__MyNone(): MyOption
  function variantOfMyOption(self: Ref): MyOption
  function isMyOption(self: Ref): Bool

  axiom ax_variantOfMyOption {
    forall x: Ref :: { variantOfMyOption(x) }
      variantOfMyOption(x) == MyOption__MySome()||
      variantOfMyOption(x) == MyOption__MyNone()
  }
  axiom ax_isMyOption {
    forall x: Ref :: { variantOfMyOption(x) }
      isMyOption(x) == (
        variantOfMyOption(x) == MyOption__MySome()||
        variantOfMyOption(x) == MyOption__MyNone())
  }
}
```

Listing 3.22: Domain for Listing 3.21

The actual data that is contained in an enum instance is stored in fields of a Silver object. Similarly to structs, Silver fields are created for each unnamed field of the enum variants. In addition to the data fields, the object also contains a field to store the current enum variant. To set up the variant information, the functions from the enum's domain are used, e.g. to encode that an enum instance x contains a *MySome* variant, the following is assumed: $variantOfMyOption(x.MyOption__variant) == MyOption__MySome()$. The reason we use an additional field to store this information is the following: in Rust it is possible to move an instance of an enum to another binding that already contains a different enum instance. This means the enum variant stored in a binding can change, which means after a move it is necessary to update the variant information of the target binding. To do that, Rust2Viper cannot simply add another assumption with the new variant, because it could conflict with the existing variant information and cause an inconsistency. To prevent such inconsistencies, the existing variant information must be invalidated first. In Silver, which does not contain a separate *havoc* statement to remove all assumptions for a location, this can be done by exhaling and then inhaling all permissions to a field.

Finally a predicate representing instances of the enum is created. Listing 3.23 shows the field declarations and the predicate created for our running example. Note that in the predicate the functions of the *MyOption* domain are used to conditionally give access to the data stored in the *MySome* variant.

```
field MyOption__variant: Ref
field MyOption__MySome__0: Ref

predicate valid__MyOption(this: Ref) {
  acc(this.MyOption__variant) &&
  isMyOption(this.MyOption__variant) &&
  acc(this.MyOption__MySome__0) &&
  (variantOfMyOption(this.MyOption__variant) == MyOption__MySome() ==> (
    acc(valid__Bar(this.MyOption__MySome__0))
  ))
}
```

Listing 3.23: Data representation for Listing 3.21

3.4.1 Pattern Matching

The second aspect of our enum encoding discussed in this section is the handling of pattern matching. When doing pattern matching on a variable *x* in Rust, this means this variable is borrowed for the body of the *match* statement. We will refer to this variable as *match argument* later on. Note that in Rust it is possible to create mutable references to fields in a *match* arm and those references can be used to modify the content of the enum instance. Thus the type of the borrow depends on the *match* arms. If any arm creates a mutable reference to parts of the enum instance, the instance is borrowed mutably. Otherwise it is borrowed immutably. So in order to translate Rust's *match* statement, we have to unfold the appropriate enum instance first to get access to the contents of the corresponding Silver object. For each *match* arm, an *if* statement is generated. In the conditions of the *if* statements, the functions defined in the enum's domain are used to differentiate the *match* arms based on the variant of the enum instance. In the bodies of those *if* statements, new variables for the bindings in the *match* arms are created. Note that in Rust, the *match* arms must be exhaustive; all possible cases must be covered. Finally, at some point after the *match*, the predicate instance of the *match* argument must be folded again. The exact program point depends on the borrow information. There are two general cases. In the first case, no references to the content of the enum instance escape the body of the *match*. In this case, the predicate can be folded right after the *match* body. Listing 3.24 shows the translation of a *match* block in which no references escape the block.

```
match x {
  MySome(ref v) => {}





  None => {},





}
```

```
unfold acc(valid__MyOption(x), rd)
if (variantOfMyOption(x.MyOption__variant) ==
    MyOption__MySome()) {
    var v: Ref
    v := x.MyOption__MySome__0
} else
  if (variantOfMyOption(x.MyOption__variant) ==
      MyOption__MyNone()) {
  } else {
    assert false
  }
}
fold acc(valid__MyOption(x), rd)
```

Listing 3.24: Translation of pattern matching in Rust to Silver

In the second case, references to the content escape the *match* body, e.g. by returning a reference from the *match* block or by assigning a reference to a variable defined in an enclosing scope. In this case, the predicate cannot be folded immediately after the *match* block, because then the escaped reference could no longer be used to access the content. In fact, the predicate cannot be folded before all escaping references expire. In this case, Rust2Viper has to use information from the borrow checker again. When a reference escapes from the body of a *match* block, the lifetime of the borrow of the *match* argument is extended to the lifetime of the escaping references. This extension is required in order to enforce the borrow rules and can be used to fold the predicate instance at the right program point in our encoding. Listing 3.25 shows a *match* statement where a reference escapes when the *MySome* variant is matched. When *None* is matched, the program panics [3] and terminates. By assigning the escaping reference to `ref1`, the borrow of `x` is extended to the lifetime of `ref1`. When `ref1` goes out of scope, the predicate instance unfolded at the beginning of the *match* can be folded again. Note that it is also possible for multiple references to escape. In this case, the borrow checker uses the longest lifetime of the escaping references to extend the lifetime of the borrowed *match* argument and the predicate instance is folded at the right program point in this case as well.

```
1 {
2   let ref1 =  match x {
3       MySome(ref v) => v
4       None => panic!()
5   };
6   // ...
7
8   // borrow of x ends here
9 }
```

Listing 3.25: A reference escapes from a match statement

3.5 Loops & Structural Specification Using Magic Wands

Specification for loops in Silver can be provided by loop invariants. In the body of a loop, only properties ensured by loop invariants can be relied on. This includes functional specification, but also permission-related information. In order to access a field in the body of a loop, the loop invariants must guarantee permissions to access the field in the current loop iteration. The translation of loops proved to be particularly challenging, because one goal of Rust2Viper is to hide all permission-related specification from the user. This means appropriate loop invariants to preserve the required permission-related properties have to be created automatically. In this section, we first discuss how permission-related loop invariants are generated for loops that do not traverse recursive data structures. The handling of loops that traverse recursive data structures is more involved and will be discussed later in this section.

For loops, Rust2Viper has to generate loop invariants automatically for the structs and enums used in the loop body. To do that, we again use Rust's type information. The basic idea is the following: when a variable or field in the loop body is accessed or updated

[3] A *panic* in Rust causes the current thread to unwind its stack and while unwinding, all objects owned by the thread are destroyed. Single-threaded programs are terminated by a *panic*. The `panic!()` macro can be used to cause a panic in a Rust program.

in a safe Rust program, this means that the objects involved can be accessed or updated for sure, because Rust guarantees that there are no null pointers or invalid memory addresses, which could cause run-time errors like segmentation faults. More specifically, the types of the variables used in the loop body reflect the capabilities required in the loop body. In the Silver encoding those capabilities are reflected by the generated predicates for each type and those predicates can be used in the loop invariants.

To generate the permission-related loop invariants, Rust2Viper first collects the type information of all variables used in the loop body, the loop condition and the user-provided loop invariants. It then adds a loop invariant for each used variable using the predicate reflecting the capabilities of the type of the variable. By default, we use the top-level predicate for variables and field accesses, e.g. for a field access `x.f`, we would use the predicate reflecting `x`'s capabilities. The amount of permission depends on the use of each variable. For variables that are updated, e.g. because they appear on the left side of an assignment, *write* permissions are used. For all other variables, *read* permissions are used. Another possibility would be to use only Rust's type information to get the permissions for the loop invariants. This would mean however, that users would have to add loop invariants for mutable variables that are only read in the loop body. We use the example in Listing 3.26 to illustrate what kind of invariants are generated by Rustviper for a simple loop. Note that the Silver translation on the right side only shows the encoding of the loop for brevity. In the loop the field `c.v` is accessed and updated. This means in the loop body, *write* permissions to `c` are needed. The Rust compiler guarantees that during each loop iteration the variable `c` can be mutated and that the variable `c` has the capabilities of its type, *Counter*. This means that during each loop iteration the predicate instance *valid__Counter(c)* is held with *write* permissions and thus can be used as a loop invariant.

```
1 struct Counter {
2   v: i32,
3 }
4 let mut c = Counter {
5                 v: 0
6             };
7 while c.v < 10 {
8   c.v += 1;
9 }
```

```
1 while (unfolding acc(valid__Counter(c), rd) in
2           c.v < 10)
3   invariant acc(valid__Counter(c), write)
4 {
5   unfold acc(valid__Counter(c), write)
6   c.v := c.v + 1
7   fold acc(valid__Counter(c), write)
8
9 }
```

Listing 3.26: Simple counting loop

There is a case however, which must be handled with additional care: loops where only parts of a data structure are accessible, e.g. because a field of a struct was moved or borrowed mutably. Listing 3.27 shows such a case. Before the loop, the field `f.a` is moved. This means that in the loop body, only `f.b` can be accessed. In our encoding, the reduced capabilities are reflected as follows. After the move in line 7, the predicate instance for `f` remains unfolded. Hence in the loop invariants, the top-level predicate cannot be used. Instead the predicate for `f.b` has to be used. To detect those cases, we keep a set of all moves and when generating the loop invariants we check if parts of a struct used in the loop body were moved before. Note that we use boolean variables, if a loop can be reached through different paths in the control-flow graph, to track along which paths a move happened. This is similar to the technique we use to handle control-flow dependent borrows which was described in Section 3.3.2.

```
struct Foo {
  a: Bar,
  b: Bar,
}

let f = Foo {...};
let a = f.a;
while cond {
  let ref1 = &f.b;

  // ..
}
```

```
unfold acc(valid__Foo(f), rd)
var a: Ref
a := f.Foo__a

while (cond)
  invariant acc(f.Foo__b, rd)
  invariant acc(valid__Bar(f.Foo__b), rd)
{
  var ref1: Ref
  ref1 := f.Foo__b
  // ...
}
```

Listing 3.27: Loop in which only parts of a data structure can be accessed in the body

3.5.1 Loop and Break

In addition to *while* loops which iterate as long as a condition is satisfied, Rust also has a *loop* construct which loops forever, unless *break* or *return* are used to exit the loop. The *loop* construct is commonly used in Rust programs, because Rust's *for* and *while let* loop constructs are desugared to a *loop* with pattern matching in the loop body. The example in Listing 3.28 shows how *loop* and *break* can be used to calculate the length of a linked list. In the optimal Silver encoding, *loop* would be mapped to a while loop with with *true* as condition, and *break* would be mapped to a *goto* to a label just after the loop body. However at the moment there is a bug [4] in Silicon which makes that encoding unfeasible: using *goto* to jump out of a loop body is not supported. In the actual encoding, we introduce an additional boolean variable which is used as the condition of the *while* loop. A *break* statement is then translated to setting this boolean variable to false and a *goto* to the end of the loop body.

```
let mut current = &list;
loop {
    match *current {
        List::Node(_, ref tail) => {
            current = tail;
        },
        _ => break
    }
}

//
```

Listing 3.28: Traversing a list iteratively

3.5.2 Traversing Recursive Data Structures

When traversing a recursive data structure using a loop in our encoding, in each iteration the predicate of the current element has to be unfolded in order to get access to the next element. This means as the loop traverses the data structure, the predicates of the visited elements get unfolded. However those predicates cannot be folded again at the end of

[4] Silicon Bug #210 `https://bitbucket.org/viperproject/silicon/issues/210/`

each loop iteration, because then access to the next element would be lost immediately. So after the last loop iteration, the predicates of all traversed elements remain unfolded. This is problematic however. For example, if a loop traversed a list that was passed as a function argument, the *List* predicate of the head of the list must be restored at some point after the loop before returning from the function. This is the main challenge when encoding loops that traverse recursive data structures.

In Silver there are two possible ways to refold the predicates unfolded while traversing data structures described in [20]. First, additional predicates and specification-only methods could be used. Alternatively, magic wands can be used. In our encoding, we use magic wands to restore predicate instances after loops, because the wands follow nicely from our encoding of borrows. Note that in order to traverse a data structure iteratively in Rust, a reference to the beginning of the data structures is needed to start with. In each iteration of the loop, parts of the data structure are borrowed subsequently. Applying the basic idea for our encoding of borrows, a new magic wand is created for the borrows in each loop iteration. Those magic wands can be used to restore permissions to the capabilities of the traversed data structure, by giving up the capabilities of the current element. First, we will present how magic wands can be used to restore the capabilities of traversed data structures after a loop conceptually. Afterwards, we will discuss how those magic wands can be encoded in Silver.

Now consider the loop in Listing 3.28 again. In each iteration, the tail of the `current` element is borrowed, which reduces the capabilities of `current` as long as the borrow is alive. Intuitively those borrows provide the following information: when the borrow of `tail` ends, the full capabilities of `current` are restored. In the following, we use $P(x)$ to refer to the predicate instance reflecting the capabilities of variable x. In our encoding, the loss of capabilities in each iteration is reflected by the fact that $P(current)$ is unfolded. The promise, that when the borrow of `tail` ends $P(current)$ can be restored, can be expressed by the magic wand $P(tail) \mathrel{-\!\!*} P(current)$. Such a wand can be created for each loop iteration resulting in a set of magic wands $P(tail_i) \mathrel{-\!\!*} P(current_i)$ where $tail_i$ and $current_i$ refer to the values *tail* and *current* pointed to at the beginning of the i-th loop iteration. Note that $current_1 = list$ and $current_i = tail_{i-1}$. After the loop, those wands can be applied in order to restore $P(list)$:

$$P(current_n) \mathrel{-\!\!*} P(current_{n-1}) \mathrel{-\!\!*} \cdots \mathrel{-\!\!*} P(current_1) \mathrel{-\!\!*} P(list)$$

where n refers to the last loop iteration.

However, this approach cannot be used directly in our Silver encoding. Unfortunately the set of magic wands described in the previous paragraph cannot be specified in terms of loop invariants conveniently, because $current_i$ and $tail_i$ can only be referred to in iteration i. However, it is possible to summarize the set of magic wands in a way that avoids referring to all $current_i$ and $tail_i$. For each element $current_i$, the chain of wands from $current_i$ to the beginning of the list *list* can be summarized by $P(current_i) \mathrel{-\!\!*} P(list)$. This wand only refers to the current state and the final state, which can be restored with the wand. Therefore this wand can be maintained in each loop iteration and can be used as a loop invariant. After the loop, it can be used to restore $P(list)$, when the borrow to *current* expires, be giving up $P(current)$ which now points to the end of the list, $current_n$.

Now consider the encoding of the loop shown in Listing 3.29. In this example, $valid__List(x)$ reflects the capabilities of a list node x. Note that in Silver, asserting that a predicate instance P is held with permission p is done with $acc(P, p)$. In the example, the loop traverses all elements of an immutable list, which means the predicates for the list elements are held with some read permissions. Therefore we use $P(x) = acc(valid__List(x), rd)$. The loss of capabilities in each iteration is reflected by unfolding `P(current)` in line 11. Throughout the loop, we maintain the wand $P(current) -\!\!* P(list)$. In each loop iteration, the wand is maintained by applying the wand packaged in the previous loop iteration and packaging the wand again. The required ghost operations are shown in lines 16 and 17 of Listing 3.29. In order to apply the wand packaged during the previous iteration, we first need access to the value *current* pointed to at the beginning of the current iteration. We introduce an additional variable `prev`, to store this value. Now this variable can be use on the right side of the wand to fold $P(prev)$ and apply the wand from the previous iteration $P(prev) -\!\!* P(list)$ and finally the new wand $P(current) -\!\!* P(list)$ can be packaged. Note that we have to used *folding* on the right side of the wand because the reference `tail` points to a field inside `prev`, hence it cannot be folded with a *fold* statement. Otherwise access to the field `tail` points to would be lost. Finally after the loop, when `current` goes out of scope, the wand can be applied to restore $P(list)$.

```
1 define P(e) acc(valid__List(e), rd)
2 var current: Ref
3 current := list
4 package P(e) --* P(list)
5
6 while (loop_cond)
7   invariant P(current)
8   invariant P(current) --* P(list)
9 {
10   prev := current
11   unfold P(current)
12   if (variantOfList(current.List__variant) == List__Node()) {
13       var tail: Ref
14       tail := current.List__Node__1
15       current := tail
16       package P(current) --* folding P(prev) in
17         applying (P(prev) --* P(list) in P(list)
18     } else {
19       fold acc(valid__List(current), rd)
20       loop_cond := false
21       goto loop_end
22
23     }
24   label loop_end
25 }
26
27 apply P(current) --* P(list)
```

Listing 3.29: Translation of Listing 3.28, with ghost operations to maintain the magic wand

Detecting when to generate Wands

Detecting for which loops magic wands are needed and where to package them is a key aspect of our translation of loops. Magic wands are needed for loops where a borrow outlives the loop body, because in those cases, the permissions removed by the borrow cannot be restored at the end of the loop body. They have to be restored after the loop, which means we have to preserve the information of how to restore the permissions through a loop invariant. We will refer to variables storing borrows that outlive a loop body as variables that require a magic wand.

The first step is to detect borrows that outlive the loop body. For immutable borrows, the borrow checker provides this information. Consider Listing 3.30, which shows Listing 3.28 and also highlights the lifetime of the loans. In this example, the borrow checker tracks loans for `list` and `current`. The loan for `current` is created inside the loop body, but the assignment `current = tail;` extends the lifetime of the loan to the scope of `current` (which is until the end of the loop), because `tail` points inside the object the loan was created for initially. In those cases, it is necessary to use magic wands to restore the permissions lost by the borrow.

```
1  let mut current = &list;
2  loop {
3      match *current {
4          List::Node(_, ref tail) => {
5              current = tail;
6          },
7          _ => break
8      }
9  }
10
11 //
```

Listing 3.30: Listing 3.28 with borrow lifetime information

The second step is to package the magic wands correctly. Once we know for which variables magic wands are needed, ghost operations must be generated to create and maintain the wands for those variables. Whenever a variable x that needs a magic wand is updated with a value y, a magic wand must be created. This magic wand specifies that by giving up $P(x)$, $P(y)$ can be restored: $P(x) \mathrel{-\!\!*} P(y)$.

Limitations

While our implementation can handle common loop patterns that traverse recursive data structures, there are some limitations which we discuss in this section.

First, when traversing a recursive data structure using a mutable reference, the information the borrow checker provides is not precise enough, because a trick is needed in order to make the borrow checker accept such loops in the first place. Listing 3.31 highlights the problem. The structure is very similar to Listing 3.28, but in Listing 3.31 a mutable reference is used instead of an immutable one. Note that an additional variable `temp` is needed for the borrow checker to accept the loop. Without the additional variable `temp`,

the borrow checker would complain that `current` would be borrowed mutably multiple times, because in each iteration `current` is borrowed mutably until after the loop. The borrow checker does not track the fact that by updating `current`, the previous mutable borrow expires. By moving `current` to `temp` at the beginning of the loop, the scope of the loan for `temp` is limited to the loop body and the borrow checker can guarantee the borrow rules are satisfied. This in turn however means that we cannot rely on the borrow checker alone to find borrows that outlive the loop body in this case. Therefore we have to manually track alias information inside loop bodies to find out which borrows outlive the loop body. At the moment, our analysis supports loops that use a single reference to iterate over a recursive data structure, but does not support loops that iterate over multiple data structures.

The second limitation is related to traversing recursive data structures with immutable references. In our encoding, we use a fixed amount of permission *rd* when specifying that the predicate instance for the current element is held at the beginning and end of a loop iteration in the invariant, e.g. as in line 7 in Listing 3.29. This means when getting the next element of the data structure, the predicate has to be unfolded with the same permissions *rd* in order to maintain the loop invariant. If fewer permissions than *rd* are used, then there will be fewer permissions to the next element. But this also means that after unfolding the predicate, we do not hold any permissions to this predicate instance any longer and we cannot use the current element any longer. Note that this limitation does not apply for mutable references, because once the reference to the current element is borrowed mutably it cannot be used to access the current element any longer in the Rust program.

```
1   let mut current: &mut List = list;
2
3   loop {
4       let temp = current;
5       match *temp {
6           List::Node(_, ref mut tail) => {
7             current = tail;
8           },
9           List::Nil => {
10              break;
11          }
12      };
13  }
```

Listing 3.31: Traversing a list using a mutable reference

3.6 Encoding Rust's Integer and Boolean Types

Up to this point, we used Silver's *Int* type for all integer types in Rust. Our actual encoding of Rust's integer types is more involved. There are two aspects of Rust's integer types that are not captured by Silver's *Int* type.

First, in Rust it is possible to create references to all kinds of data. In particular, it is possible to create references to integer variables and integer struct or enum fields. Note that mutable references to a location storing an integer can be used to modify the stored

integer. In Silver on the other hand, references can only point to objects, not to other data types, like *Int*. This means Silver objects must be used to represent Rust integer variables in general. By representing integer variables through objects on the heap, references to integer variables can be modeled by references to those objects in Silver. Therefore we create a Silver object with with a single field *Int__v* for each integer variable in a Rust program. For integer fields in structs or enums, we directly embed the objects representing the integer fields in the predicate for the struct's or enum's type. Listing 3.32 illustrates this embedding. Note how the integer field `Bar.f` is encoded by a field `Bar__f`, which points to an object containing a field `Int__v`, which contains the actual integer value.

```
1 struct Bar {
2   f: usize
3 }
4
5
6
7
8 //
```

```
1 field Int__v: Int
2
3 field Bar__f: Ref
4
5 predicate valid__Bar(self: Ref) {
6   acc(self.Bar__f)&&
7   acc(self.Bar__f.Int__v)
8 }
```

Listing 3.32: Translation of Rust integer field using an additional object

Immutable references to integers in Rust can be encoded by references to objects on the heap, where some *read* permissions are held to a field *Int__v*. For mutable references to integers, *write* permissions to the field *Int__v* must be held. When integer variables or references to integers are passed as function arguments, references are used as well. Pre- and postconditions giving access to the field *Int__v* must be generated. For loops that use integer variables, similar loop invariants must be created. Listing 3.33 shows the translation of a Rust function that takes a reference to an integer as an argument.

```
1 fn foo(x: &usize) {}
2
3
4
5 //
```

```
1 method foo(x: Ref, rd: Perm)
2   requires none < rd && rd < write
3   requires acc(x.Int__v, rd)
4   ensures acc(x.Int__v, rd)
5 {}
```

Listing 3.33: Translation of a Rust function that takes a reference to an integer as an argument

Note that with this encoding many objects are added to the heap. It also increases the size of the resulting Silver code, because instead of just assigning to an variable of type *Int*, a new object must be created and the value must be assigned to the *Int__v* field of the object. As it turns out, in some cases, normal variables of type *Int__v* can be used. The main reason why integers are stored on the heap is to encode mutable references to integers. But when no mutable references to a location storing an integer are created, the integer does not necessarily have to be stored on the heap. For example, consider a function that takes an immutable reference to an integer as an argument. In safe Rust, no mutable references to the argument can be created and this immutable reference can be used to for exactly two things: accessing the integer value it points to or using the address the reference points to as an integer value. The second case is uncommon in

Rust, because in order to use the address as an integer value, the reference must be cast to a raw pointer first, which can then be cast to an integer type. Also note that comparing two references in Rust will not compare the addresses but the content they point to. This means, as long as an immutable reference to an integer is not cast to a raw pointer, only the value it points to can be used. In those cases it would be sufficient to use Silver's *Int* type to store the value instead of creating objects on the heap. While this will probably result in in a slightly reduced verification runtime, this optimization is not implemented in Rust2Viper at the moment.

The second aspect of Rust's integer types not covered by Silver's *Int* type is the fact that integer types in Rust only cover a finite subset of mathematical integers. In Rust there are integer types with different sizes and types that either include or exclude negative values, e.g. *u8* represents a 8-bit unsigned integer and *i32* represents a 32-bit signed integer. Silver's *Int* type, on the other hand, represents the infinite set of mathematical integers. In Silver, the range constraints of Rust's integer types can be encoded as assertions, e.g. the set of possible values of Rust's *u8* type can be encoded by $0 \leq x < 256$, where x refers to the values that can be stored in variables of type *u8*. In Rust, the semantics of the built-in integer types define integer overflows as program errors, while in many other programming languages, integers wrap around on overflows. The RFC [11] introducing this behavior requires Rust implementations to dynamically check for integer overflows at least when debug assertions are enabled and permits checks for overflows at any time. In the official Rust compiler, dynamic checks for overflows are only added in debug mode. In Silver, those checks can be encoded as assertions, which could potentially be used to remove the need for dynamic overflow checks, by proving the absence of overflows. The constraint that an unsigned integer operation $x \circ y$ does not overflow can be expressed as follows: $min(x) \leq (x \circ y) \leq max(x)$, where $\circ$ denotes an arithmetic operation and $min(x)$ and $max(x)$ return the minimum and maximum value which can be represented by the type of x [5]. Constraints of that form could be used to add proof obligations to verify no integer overflows occur in a given Rust program. Once the absence of integer overflows is verified, the dynamic overflow checks could be removed. At the moment, Rust2Viper does not automatically add proof obligations for each integer operation, but the user can add those constraints by hand easily.

Booleans are handled similarly to integers. For each boolean, a new object with a field *Bool__v* is created. This field has the type *Bool*, Silver's boolean type. Besides that, objects representing boolean values are handled exactly as objects handling integer values. Note that Silver's *Bool* type covers the same set of values as Rust's *bool* type, so there is no need for additional constraints.

3.7 Pure Functions

So far we have discussed how functions in Rust can be translated to Silver methods. But methods in Silver cannot be used to specify functional properties in contracts, because they can have side effects, such as creating new objects on the heap or modifying existing objects. In order to specify functional behavior, Silver provides *heap-dependent functions* that can be used in assertions and statements. In the following paragraph, we briefly describe some key aspects of functions in Silver. For more detailed description we refer

[5] Note that there are no implicit type conversions in Rust, so x and y must have the same static type.

to [20].

Functions in Silver are free of side-effects and this property is enforced syntactically. In Silver, the body of a functions consists of a single expression and all expressions in Silver are side-effect free. Note that this also means Silver methods cannot be called in Silver expressions. Similar to methods, functions must specify the permissions required to evaluate the function body in their preconditions. But because functions cannot modify the heap, they do not consume those permissions, which means there is no need to return permissions using postconditions. Functions can be used to encode functional properties of recursive data structure, e.g. the length of a linked list, as well as provide abstractions of the underlying data representation, e.g. creating a set of elements for a given linked list. The Viper verifiers automatically relate a function invocation to the function's body and control the unrolling of recursive function definitions [16].

In Rust2Viper, we support encoding Rust functions as pure Silver functions. This way, Rust functions can be used to specify functional properties in contracts. But only side-effect free Rust functions can be translated to Silver functions and in Rust, there is no notion of side-effect free function, there only exists the notion of general functions, which may or may not be side-effect free. Therefore we introduced an attribute *pure_fn*, which can be used to annotate functions that are side-effect free and should be translated to Silver functions. During the translation, Rust2Viper checks if the function bodies of Rust functions marked as pure adhere to Silver's definition of pureness. This check is mostly syntactic, namely: a Rust function is considered pure by Rust2Viper if its body can be translated to a single Silver expression. Only function calls in the function body need to be handled more carefully, because only Silver functions and not methods can be called in Silver expressions. Therefore we first collect all Rust functions marked as pure. When translating function bodies, we assume all functions marked as pure can be translated to Silver functions. If one of those functions marked as pure cannot be translated to a Silver function, an error is raised when translating this function. Note that Silver does not support any IO operations, therefore Rust programs containing such operations are not handled by Rust2Viper at the moment and there is no need to consider IO related side-effects.

As mentioned earlier, functions in Silver require the permissions to evaluate the function body to be specified in the function's preconditions. But because functions can neither modify objects nor consume permissions, there is no need to specify a concrete amount of permission in the precondition. We only need to specify that there is some positive amount of permission, in order to access fields in the function body in a read-only fashion. For this use case, Silver provides a special permission amount, *wildcard*, which denotes some positive, unspecified permission amount that is chosen automatically. We use *wildcard* in our translation for all permissions in Silver functions.

Now consider the pure function `length` in Listing 3.34. We use this example to highlight the actual translation of pure functions. For recursive functions, it is essential to contain conditionals to differentiate the base cases and the recursive cases. In the Rust example, pattern matching is used, but *if* statements could be used as well. Silver provides a ternary operator `cond ? a : b`, which evaluates a boolean expression `cond` and evaluates the expression `a` if `cond` is true and evaluates `b` otherwise. Listing 3.35 shows the Silver translation of Listing 3.34. The translation of the pattern matching is very similar

to the encoding of *match* statements in non-pure functions (described in Section 3.4.1). The only differences are that in order to generate a single expression for pure functions, the ternary ? operator is used instead of *if* statements, *let* is used instead of local variable declarations and *unfolding* is used instead of *unfold* statements.

```
1 #[pure_fn]
2 pub fn length(l: &List) -> i32 {
3     match *l {
4         List::Node(_, ref tail) => {
5             1 + length(tail)
6         },
7         _ => 0
8     }
9 }
```

Listing 3.34: Pure Rust function to calculate the length of a linked list.

```
1 function length(l: Ref) : Int
2   requires acc(valid__List(l), wildcard)
3
4 {
5   (unfolding acc(valid__List(l), wildcard) in
6     (variantOfList(l.List__variant) == List__Node()) ?
7       (let tail == (l.List__Node__1) in 1 + ( length(tail))) : 0)
8 }
```

Listing 3.35: Translation of Listing 3.34

Note that not all side-effect free Rust functions can be translated to Silver functions. For example, a Rust function that creates a struct instance on the function-local stack can be considered pure, as this does not modify any state observable outside the function. But there is no way to translate this function to a Silver function, because new objects can only be created using the *new* statement. This also means that we cannot use the integer encoding described in Section 3.6. Therefore in our encoding, we always use Silver's *Int* and *Bool* types, instead of objects, for integer and boolean arguments, return values and bindings. The only limitation of this approach is that the addresses of references to integers and boolean value cannot be compared. But as already mentioned in Section 3.6, such comparisons are discouraged in Rust.

3.8 Unsafe Rust

So far we only discussed translating *safe* Rust code, but in Rust there is a distinction between *safe* and *unsafe* code. For safe Rust, the compiler ensures strong static guarantees about behavior, such as the guarantees provided by the borrow rules or the guarantee that references always point to a valid memory address. In our translation so far, we relied on some of those guarantees. But the safety checks by the compiler are conservative and the compiler rejects some safe programs because it cannot verify all guarantees statically. In unsafe code, the restrictions by the compiler are partly relaxed. In Rust, unsafe code is marked with the keyword `unsafe`. There are a few places in Rust source code where `unsafe` can be used, but we will focus on unsafe blocks, blocks of Rust code that are preceded by the keyword `unsafe`.

Unsafe blocks allow programmers to do the following things, which are forbidden in safe Rust:

1. Access or update a static mutable variable
2. Dereference a raw pointer
3. Call unsafe functions.

Static mutable variables in Rust represent global mutable state, which means multiple threads can access and modify the same global variable. When dealing with mutable global state, the user has to take additional care to avoid data races. Reasoning about shared mutable data is discussed for example in [19]. A similar technique could be used to reason about unsafe code involving static mutable variables. The existing techniques however do not solve the problem in a general fashion and we focus on the remaining two unsafe scenarios in the remainder of this section.

Besides references, Rust's type system also provides raw pointer types. Raw pointers support operations similar to C pointers, e.g. pointer arithmetic, and raw pointers are not tracked by the borrow checker. Dereferencing raw pointers is *unsafe* in Rust, because for raw pointers, the type system does not guarantee that they point to valid memory, which can be dereferenced or the memory they point to is initialized correctly. For example, a raw pointer can point to an invalid memory address and then dereferencing will cause a segmentation fault.

Calling unsafe functions on the other hand allows calling arbitrary C functions. This is probably the most powerful and widespread usage of unsafe code in Rust. There are two important use cases in particular. First, Rust's core library provides unsafe functions for interacting with the underlying operating system on a low level. For example, it provides functions to allocate and deallocate memory using raw pointers and functions to cast Rust types to arbitrary other types. Second, functions from user-provided libraries written in any language can be called from Rust as an unsafe function, as long as they provide a C interface. In this section, we will not discuss calling unsafe functions from user-provided libraries. A key issue when it comes to unsafe functions is how to specify that the called functions do not violate any guarantees Rust's type system provides across programs written in different programming languages.

After briefly introducing unsafe code, we will now discuss potential benefits of reasoning about unsafe code. As mentioned initially in this section, some restrictions are relaxed for unsafe code, because the compiler cannot reason precisely enough about the code. For example, consider an unsafe code block that uses pointer arithmetic to calculate an address, which is then dereferenced. Guaranteeing the absence of segmentation faults for such an unsafe block would require reasoning about the valid ranges of memory addresses in a general fashion, which the Rust compiler does not support. In some sense, the burden to ensure an unsafe block does what it is supposed to do is partly shifted from the compiler to the programmer. In particular, the programmer has to ensure that all the invariants of Rust's type and borrow system are maintained by unsafe code. When such an invariant is violated by a Rust program, the behavior of that program is undefined. Listing 3.36 contains two functions that illustrate how two invariants can be violated using unsafe code. In the first function, `two_mutable_references`, two mutable references to the same variable `x` are created by casting a reference to a raw pointer and then dereferencing that

pointer. This violates the rule that there can be at most one mutable reference to the same data at any given time and the behavior of this function is undefined. The second function, `null_pointer`, creates a raw null pointer and dereferences it, which violates the invariant that only valid memory addresses are accessed by a Rust program. Again, the behavior of this function is undefined.

```
1  fn two_mutable_references() {
2      let mut x: u8 = 1;
3      let ref_1: &mut u8 = &mut x;
4      let ref_2;
5      unsafe {
6          let xp = ref_1 as *mut u8;
7          ref_2 = &mut *xp;
8      }
9      // oops, ref_1 and ref_2 point to the same piece
10     // of data (x) and are both usable
11 }
12
13 fn null_pointer() -> i32 {
14     let x: i32;
15     let xp: *const i32 = ptr::null();
16     unsafe {
17         // oops, dereferencing a null pointer
18         x = *xp;
19     }
20     x
21 }
```

Listing 3.36: Two functions illustrating how unsafe code can be used to cause undefined behavior in Rust.

At the moment, Rust programmers have to ensure the correctness of their unsafe code manually, for example by introducing run-time checks or by documenting important invariants. But as we showed in the previous listing, unsafe code can cause undefined behavior rather easily and the correctness of unsafe code is essential for the safety of a Rust program. And even though most Rust programs do not require unsafe code explicitly[6], most parts of Rust's core and standard library rely on unsafe code, which in turn means programs using those libraries also rely on unsafe code. Providing better tools to ensure properties of unsafe code statically can therefore improve the safety of a wide range of Rust programs.

There is a growing research interest in this area. Recently, the Foundations of Programming group at the Max Planck Institute for Software Systems announced a research project focused on the safety of Rust [26] and Toman et al. introduced Crust [27], a tool to verify properties of unsafe Rust code using model checking. In their running example they show how their tool can be used to find a bug in a simplified version of Rust's array implementation. In this thesis, we investigated how Rust2Viper can be used to prevent similar problems. Note that while Rust2Viper at the moment can only deal with a small subset of unsafe Rust code, this subset gives an idea how unsafe code could be handled

[6] Rust2Viper, which is implemented in Rust, does not require a single unsafe block, for example

by Rust2Viper in a more general setting.

```
1 struct Array {
2     ptr: * mut u32,
3     len: usize,
4 }
5
6 impl Array {
7     fn get_mut(&self, index: usize) -> &mut u32 {
8         if index >= self.len { panic!(); }
9         unsafe {
10            let ptr = self.ptr.offset(index as isize);
11            &mut *ptr
12        }
13    }
14 }
15
16 fn client(a: &mut Array) {
17     let r1: &mut u32 = a.get_mut(0);
18     let r2: &mut u32 = a.get_mut(0);
19 }
```

Listing 3.37: Simplified implementation of an array type in Rust.

3.8.1 Initial Work on Verifying Unsafe Code Dealing with Raw Pointers

In this subsection, we present our initial work on verifying unsafe code dealing with raw pointers. We focus on a very specific use case, namely specifying permissions for raw pointers that point to a continuous region of memory. At the moment, the user has to specify permissions to continuous regions of memory manually, which is a major difference to the way Rust2Viper handles safe Rust code.

To begin with, let us introduce the example from [27], which is shown in Listing 3.37. Note that our example does not contain generics, while the original version in [27] does, but this does not have any effect on the problem at hand. In the example, an array consists of a raw pointer to a continuous region of memory containing the array elements and a length. The function `get_mut` can be used to get a mutable reference to the element of the array at a given index. There are two important invariants that could be violated by `get_mut`. First, the pointer that is dereferenced could point to an invalid memory location. This invariant is preserved by the run-time check in line 8, given that `Array.ptr` points to a memory region that can hold at least `Array.len` elements. The second important invariant that needs to be maintained is that there can be at most one mutable reference to each array element at any given point in time. This invariant is not maintained by `get_mut`, however. As the calls in `client` show, `get_mut` can be used to get multiple mutable references to the same array element, meaning the aforementioned invariant is violated. The recommended fix for this problem was using `&mut self` instead of `&self` as first argument of `get_mut`. This way, the scenario in `client` is rejected by the borrow checker, because the first call to `get_mut` will extend the mutable borrow to a until the returned reference `r1` goes out of scope. When `get_mut` is called for the second time, a cannot be borrowed as mutable again, hence the borrow checker will reject the

function call. Note that this fix relies on the over-approximation of the borrow checker and prevents calls to `get_mut` that are unproblematic, e.g. as long as the indices do not overlap, calling `get_mut` for multiple indices would, in principle, be fine.

To handle data types such as `Array` in the previous listing, we first need a way to represent pointers to continuous regions of memory. In Silver, *quantified permissions* can be used to specify unbounded heap data structures [20]. They are similar to separation logic's iterated separating conjunction [24] and can be used to specify permissions *pointwise*. Syntactically, quantified permissions are expressed using a quantifier and accessibility predicates. For example, `forall x: Ref :: x in Y ==> acc(x.f)` states permissions to `x.f` are held for each reference `x` in the set `Y`. In [20], Müller et al. also show how arrays can be encoded in Silver. In Rust2Viper, we use a similar approach to model continuous regions of memory. The basic idea is to introduce a new type *Ptr* using a mathematical domain and encode accessing a pointer *ptr* with offset x with `loc(ptr, offset).val`, where `loc(ptr: Ptr, offset: Int): Ref` is a function mapping a base pointer and an offset to distinctive elements. Permissions to a continuous region of memory can then be expressed by quantified permissions ranging over all valid offsets.

With such an encoding, permissions can be used to reason about some aspects of pointers. In Rust2Viper we provide a way to handle the requirements of functions like `get_mut` more precisely. More specifically, we provide a way to make the assumptions and invariants explicit, so they can be verified using a Viper verifier. In particular, we want to encode to following property: as long as no mutable reference was returned by `get_mut` for index i, calling `get_mut` with index i is allowed. To handle this and similar cases, we add additional ghost state to keep track to which offsets for a pointer permissions are held currently. For a raw pointer, a new set, called *accSet*, is added in the encoding. The *accSet* of a pointer contains the offsets to which permissions are held currently. Listing 3.38 shows how the specification for `get_mut` can be expressed in terms of an *accSet*. The first precondition ensures the index can be accessed, given that `Array.len` refers to the maximum number of accessible elements. This property could be ensured by appropriate contracts for functions responsible for allocation memory for `Array.ptr`. The second precondition states that permissions to access the array element with offset `i` are required. Finally the postcondition states that after calling `get_mut` with index i, i is removed from the *accSet* of `self.ptr`. Once the returned reference expires, the corresponding index can be added to the *accSet* again. Therefore we have to track additional state for the returned references: the pointer to the continuous memory region and the index the returned reference points to. At the moment, Rust2Viper only tracks this additional state in the method that called `get_mut`. Using the *accSet* allows Rust2Viper to reason more precisely about permissions to memory locations than the Rust compiler currently does. Also note that while the `Array` struct presented is rather simple, the same problem exists for widely used data types, like Rust's *Vec* or *HashMap* types.

The quantified permissions for the array elements are expressed in terms of the *accSet* of `Array.ptr` in the following form:

```
forall j: Int :: j in accSet(self.Array__ptr) ==>
                    acc(loc(self.Array__ptr, j).pval),
```

where `accSet(ptr)` returns the *accSet* for pointer `ptr`. The pre- and postconditions generated for `get_mut` from Listing 3.38 are shown in Listing 3.39. Note that in our encoding, we require read access to an instance of `valid__Array(self)`, which means *unfolding*

```
struct Array {
    #[use_acc_set(Array__len)]
    ptr: * mut u32,
    len: usize,
}

impl Array {
    #[precond="index >= 0 && index < self.len"]
    #[precond="inAccSet(self.ptr, index)"]
    #[postcond="getAccSet(self.ptr) ==
                old(accSetWithout(self.ptr, index))"]
    fn get_mut(&self, index: usize) -> &mut u32 {
        unsafe {
            let ptr = self.ptr.offset(index as isize);
            acc_set_remove!(self.ptr, )
            &mut *ptr
        }
    }
}
```

Listing 3.38: Simplified implementation of an array type in Rust, with specification.

expressions must be used to access the fields of `self`. In this listing, the *unfolding* expressions are emitted when accessing fields of `self` for brevity.

Finally, we want to point out some interesting points of our design for pointers. First, note that the *accSet* and the quantified permissions are not added to the predicate generated for the `Array` struct, but are carried around outside the predicate. That way, we can require write access to the *accSet* of a pointer *ptr*, even though we only have immutable access to the pointer *ptr*. In order to be able to modify the *accSet*, the function `accSet` returns a reference with a field `acc__set`, storing the *accSet*.

```
1 method Array__get_mut(self: Ref, index: Int, rd: Perm)
2   returns (res: Ref)
3   requires none < rd && rd < write
4   requires acc(valid__Array(self), rd)
5   requires acc(accSet(self.Array__ptr).acc__set, write)
6   requires index in accSet(self.Array__ptr).acc__set
7   requires index >= 0 && index < self.Array__len
8   requires forall j: Int :: j in accSet(self.Array__ptr).acc__set
9               ==> acc(loc(self.Array__ptr, j).pval)
10  requires forall j: Int :: j in accSet(self.Array__ptr).acc__set
11              ==> acc(loc( self.Array__ptr, j).pval)
12  requires inAccSet((self.Array__ptr), index)
13  ensures acc(valid__Array(self), rd)
14  ensures acc(accSet(self.Array__ptr).acc__set, write)
15  ensures forall j: Int :: j in accSet(self.Array__ptr).acc__set
16              ==> acc(loc(self.Array__ptr, j).pval)
17  ensures forall j: Int :: j in accSet( self.Array__ptr).acc__set
18              ==> acc(loc(self.Array__ptr, j).pval.Int__v)
19  ensures acc(res.Int__v, write)
20  ensures accSet(self.Array__ptr).acc__set ==
21          old(accSetWithout(self.Array__ptr, index))
22 {
23   //...
24 }
```

Listing 3.39: Simplified implementation of an array type in Rust, with specification.

Chapter 4

Functional Specification

So far we only discussed permission-related specifications and how Rust2Viper generates this kind of specification automatically. But in order to verify functional properties, the user has to provide functional specifications. In Rust2Viper, the user can provide specifications at several places. Function pre- and postconditions can be added to Rust functions using the *precond* and *postcond* attributes. The *invariant* attribute can be used to add invariants to Rust's loop constructs. Assertions in function bodies can be added using the *static_assert!* macro. Listing 4.1 shows how these attributes and macros can be used to provide functional specification for a simple Rust function.

```
1  #[precond="x > 0"]
2  #[postcond="res > 0"]
3  fn example(x: usize) -> usize {
4      let mut c = 0;
5
6      #[invariant="c <= x "]
7      #[invariant="x > 0"]
8      while c < x {
9          c += 1;
10     }
11     static_assert!("x == c");
12
13     c
14 }
```

Listing 4.1: A simple Rust function with pre- and postconditions, as well as loop invariants and a static assertion.

The assertions are provided as strings in the Rust program. For Rust2Viper, we decided to create our own language for assertions and decided against using Silver's assertion syntax for the following reasons. First, the syntax for assertions should be closely related to the syntax of Rust expressions. For example, in assertions it should be possible to refer to fields of struct instances in the same way as in normal Rust code. Second, all permission-related specification is generated automatically, so the assertion language does not have to provide permission-related constructs, such as Silver's *acc* predicate or *folding* and *unfolding* expressions. Third, we wanted the flexibility to introduce new constructs that allow better interaction with Rust's types and features in the assertion

language.

Using Rust's expression syntax for assertions would have been another possibility. This syntax would be convenient for Rust programmers, because they are already used to it. But for assertions, additional constructs, such as quantifiers, are very useful and there is no equivalent Rust syntax for that. Also, Rust expressions are not as restrictive as Silver expressions. For example, expressions in Rust can have side-effects, e.g. by calling a function with side-effects in an expression, while expressions in Silver cannot contain side-effects; only calls to pure functions are allowed. Therefore having a separate syntax for assertions makes sense, because this way there is a clear distinction between Rust expressions and expressions in assertions.

Our assertion language contains key elements from Silver's assertion language, however. There is support for integer comparisons, quantifiers and *old* expressions, which can be used to refer to the state as it was at the time a function was called. Those elements can be translated directly to Silver. But once Rust structs or enums are involved in an assertion, the translation becomes more involved. We describe those two cases in the next section.

4.1 Structs and Enums in Assertions

Before we discuss the handling of struct instances in assertions, we briefly summarize the important aspects of our data representation described in Section 3.2.1. Struct instances are represented by Silver objects and struct fields are translated to Silver fields. The field name in Silver is prefixed by the type name of the struct. A predicate is created for each struct type and this predicate can be used to get access to the fields of the struct. When a struct instance is passed to a Rust function, we hold a predicate instance of the predicate for that struct.

We discuss assertions in pre- and postconditions first, because there we do not have to consider how borrows in the function body affect predicate instances of structs. Consider the example in Listing 4.2 which contains a precondition with a field access. When it comes to translating field accesses in pre- and postconditions, two things have to be done. First, the predicate instance of the struct has to be unfolded in order to get access to the fields. This can be done using *unfolding* expressions to unfold the predicate instance of the struct with some *read* permissions. Second, we also have to translate the field name of the struct in Rust to the appropriate field name in Silver. Listing 4.3 shows the translation of Listing 4.2. Note how *unfolding* is used in line 11 of Listing 4.3 to get access to the field of `f`.

```
1 struct Foo {
2     a: i32,
3 }
4
5 #[precond="f.a == a"]
6 fn example(f: &Foo, a: i32) {
7   // ...
8 }
```

Listing 4.2: Rust function with precondition with field access.

```
field Foo__a: Int

predicate valid__Foo(self: Ref) {
  acc(self.Foo__a)
}

method new(f: Ref, a: Ref, rd: Perm)
  requires none < rd && rd < write
  requires acc(valid__Foo(res), rd)
  requires
    (unfolding acc(valid__Foo(res), wildcard) in res.Foo__a) == a
  ensures acc(valid__Foo(res), rd)
{}
```

Listing 4.3: Translation of Listing 4.2

When it comes to translating assertions in the function body, we take a slightly different approach, because moves or borrows in the function body can affect the predicate instances of structs. In this case, we treat field accesses in assertions in the same way we treat field accesses in other statements in the function body. We collect the predicates that need to be unfolded to evaluate the expression and emit *unfold* statements for them. Then the assertion is evaluated. After the assertion was evaluated, we emit *fold* statements for the unfolded predicates.

To use enums in assertions, we added a pattern-matching construct to our assertion language. Our *matching* construct is defined by the following grammar, where *expr* refers to an arbitrary expression of our assertion language, *path* refers to an arbitrary Rust path and $*_,$ refers to an arbitrary number of repetitions separated by a comma:

$\langle matching \rangle$ ::= 'matching' $\langle expr \rangle$ 'in' '(' $\langle arm \rangle *_,$ ')'

$\langle arm \rangle$ ::= $\langle variant \rangle$ '=>' $\langle expr \rangle$

$\langle variant \rangle$ ::= $\langle path \rangle$ | $\langle path \rangle$ '(' $\langle ident \rangle *_,$ ')'

Listing 4.4 shows how our *matching* construct can be used in the postcondition of `unwrap_or` to specify that either `v` or the default value `d` are returned, depending in the variant of `x`.

```
#[postcond="matching *x in (
    MySome(v) => res == v,
    MyNone => res == d
)"]
fn unwrap_or(x: &MyOption, d: i32) -> i32 {
    match *x {
        MyOption::MySome(x) => x,
        MyOption::MyNone => d
    }
}
```

Listing 4.4: Pattern matching in a postcondition

Intuitively, the semantics of *matching* constructs of the form

$$matching\ arg\ in\ (\ v_1\ =>\ a_1,\ \cdots,v_n\ =>\ a_n\ )$$

are the following: if arg is of variant v_i, then the assertion a_i must hold. Note that in contrast to Rust's *match*, the patterns in *matching* do not need to be exhaustive. For missing patterns, *true* is used by default. This means our *matching* construct is very flexible. For example, the specification can be split in several pre- or postconditions, each dealing with a single variant. Also, only the relevant cases need to be handled with our *matching* construct. For example, consider a function with a precondition that limits the variants of a function argument. When using this argument in another pre- or postcondition, only the variants permitted by the first precondition need to be handled.

To translate the *matching* construct, we generate *unfolding* expressions to unfold the predicate instance of the argument with some *read* permissions to get access to the fields of the argument. For each arm $v_i => a_i$ of a *matching* expression, we generate an implication $variantOf(arg) = v_i \implies trans(a_i)$, where $variantOf(arg)$ returns the variant of arg and $trans(a)$ translates an assertion a in our specification language to Silver. For arms that match a variant with some fields, e.g. $MyOption :: MySome(x)$ in Listing 4.4, occurrences of the binding on the right side of the implication are replaced with an expression accessing the field. The conjunction of the implications for all match arms is the expression encoding a *matching* expression.

Listing 4.5 shows the translation of the *matching* expression in Listing 4.4. Note how *unfolding* expressions are generated each time the argument `x` is accessed. The reason we opted for this approach over just generating a single *unfolding* expression at the top-level is related to using *old* in combination with *matching*. When a single *unfolding* expression at the top-level is used, the whole *matching* expression could either be evaluated in the current state or in the function's pre-state using *old*. The advantage of our encoding is that *old* can be used in two ways in combination with *matching*. First, the whole *matching* expression can be evaluated in the old state: $old(matching\ arg\ in\ (\ldots))$. Second, only the actual pattern matching is done in the old state: $matching\ old(arg)\ in\ (\cdots)$. In this case, the check $variantOf(arg) = v_i$ is done in the old state. The assertion a_i is evaluated in the current state, but matched fields can be evaluated in the old state by wrapping them in *old()*. Listing 4.6 shows how *old* and *matching* can be used to specify that the function `inc` increments the value `v` if the variant of the argument `x` is `MySome(v)`.

4.2 Interaction with Magic Wands

In two cases in our encoding, magic wands are created to restore permissions at a later point: loops and when returning references. In those cases, parts of the specification must be added to the right side of the generated wand, because full permissions are only restored once the wand is applied.

4.2.1 Specification for Returned References

First we consider the simpler case, returning references from functions in Rust. To begin with, we illustrate the problem using the example in Listing 4.7. In this example, the function `borrow` takes a mutable reference to an instance of the struct type

```
1  method unwrap_or(x: Ref, d: Ref, rd: Perm) returns (res: Ref)
2    requires none < rd && rd < write
3    requires acc(valid__MyOption(x), rd)
4    ensures acc(valid__MyOption(x), rd)
5    ensures
6      ((unfolding acc(valid__MyOption(x), rd) in
7        variantOf(x.MyOption__variant) == MyOption__MySome()) ==>
8          (res == (unfolding acc(valid__MyOption(x), rd) in
9            x.MyOption__MySome__0))) &&
10     ((unfolding acc(valid__MyOption(x), rd) in
11       variantOf(x.MyOption__variant) == MyOption__MyNone()) ==>
12         (res == d))
13 {}
```

Listing 4.5: Translation of the matching in Listing 4.4

```
1  #[postcond="matching (old(*x)) in (
2      MySome(v) => matching *x in (MySome(v) => v == old(v) + 1),
3  )"]
4  fn inc(x: &mut MyOption) {
5      match *x {
6          MyOption::MySome(ref mut v) => {
7              *v += 1;
8          },
9          _ => {},
10     }
11 }
```

Listing 4.6: Pattern matching in combination with *old* in a postcondition

Foo and returns a reference to the field `f.v`. The listing also shows the pre- and postconditions generated for `borrow` during the translation to Silver. In particular, a wand $acc(res.Int__v, write) \twoheadrightarrow acc(valid__Foo(f), write)$ to restore the original permissions of the argument f is generated. This wand is applied when the returned reference goes out of scope. In the preconditions of `borrow`, *write* permissions to the argument `f` are required. This means, the caller of the method does not retain any permissions to `f`, the content `f` points to can be modified by the called method `borrow` and the effect of `borrow` on the content of `f` must be specified in `borrow`'s postconditions. In contrast, for immutable references, the caller always retains some permissions in our encoding, which means the content cannot be changed. If we want to preserve information about the content of `f`, additional postconditions are required. But note that in the postconditions of `borrow`, no permission to $valid__Foo(f)$ is held. Therefore, postconditions involving `f` cannot be translated using the techniques we described earlier.

From the borrow checker's perspective, the borrowed argument `f` remains borrowed mutably after the function returns, until the returned reference expires. This means, the argument `f` cannot be used as long as the returned reference is in scope, otherwise Rust's borrow rules would be violated. Or in other words, until the returned reference expires, only the returned reference can be used to access parts of the argument `f`. The borrow rules also imply that there cannot be any active references to the content of `f` at the time of a function call to `borrow`. In the remainder of this section we will refer to arguments that remain borrowed after a function call as *extended* arguments. When using

```
1 fn borrow<'a>(
2     f: &'a mut Foo)
3     -> &'a mut i32
4 {
5   f.v = 0;
6   f.z = 0;
7   &mut f.v
8 }
```

```
1 method borrow(f: Ref, rd: Perm)
2 returns (res: Ref)
3   requires none < rd && rd < write
4   requires acc(valid__Foo(f), write)
5   ensures acc(res.Int__v, write)
6   ensures acc(res.Int__v, write) --*
7           acc(valid__Foo(f), write)
8 {}
```

Listing 4.7: Postconditions generated by Rust2Viper for a function that returns a reference.

an extended argument in postconditions it is therefore sufficient if the postconditions involving extended arguments hold when the returned reference expires, because only then the extended argument can be accessed again. Or stated differently, the effects of the function `borrow` on its argument `f` can only be observed once the returned reference goes out of scope. We use this fact in our encoding by adding postconditions that involve extended arguments to the right side of the wand generated for the returned reference. Note that there is no need to add postconditions that refer to extended arguments only in the function pre-state (using *old()*) to the generated magic wand.

```
1  fn client(f: &mut Foo) {
2    {
3      let r1 = borrow(f);
4      // f remains borrowed until r1 goes out of scope
5      static_assert!("*r1 == 0"); // OK
6      static_assert!("f.v == 0"); // not OK, f.v cannot be accessed while
7                                  // r1 is in scope
8      *r1 = 20;
9    }
10   static_assert!("f.v = 20");
11   static_assert!("f.z = 0");
12 }
```

Listing 4.8: Client code calling a function `borrow`.

```
1  #[postcond="*res == 0"]
2  #[postcond="res == &f.v"]
3  #[postcond="f.z == 0"]
4  fn borrow<'a>(f: &'a mut Foo)
5                 -> &'a mut i32
6  {
7    f.v = 0;
8    f.z = 0;
9    &mut f.v
10 }
```

Listing 4.9: A version of the function `borrow` from Listing 4.7 including functional specification

To illustrate this handling of postconditions, consider the function `client` in Listing 4.8,

which calls `borrow` from Listing 4.7 and contains assertions involving the data passed to `borrow`. In order to verify the assertions in `client`, the specification of `borrow` has to include information about both fields `f.v` and `f.z`. Listing 4.9 includes postconditions for `borrow` that are sufficient to verify the assertions in `client`. The generated postconditions are shown in Listing 4.10. Note that the two postconditions involving fields of the argument `f` are added to the right side of the wand generated for the returned reference.

```
1 method borrow(f: Ref, rd: Perm) returns (res: Ref)
2   requires none < rd && rd < write
3   requires acc(valid__Foo(f), write)
4   ensures acc(res.Int__v, write)
5   ensures res.Int__v == 0
6   ensures acc(res.Int__v, write) --* acc(valid__Foo(f), write) &&
7       (res == ((unfolding acc(valid__Foo(f), rd) in  f.Foo__v))) &&
8       ((unfolding acc(valid__Foo(f), rd) in  f.Foo__z.Int__v) == 0)
9 {}
```

Listing 4.10: Postconditions generated for Listing 4.9

While the general rule is straightforward, we discuss two interesting cases in the remainder of this section. The first case involves postconditions that refer to an extended argument in the function's pre- and post-state. Such postconditions are added to the right side of the magic wand. The wand however is not applied in the called function but by a caller further up the call stack. This means when applying a wand generated by a function f, *old* expressions in the wand must refer to f's pre-state, not to the pre-state of the current function. This can be done using Silver's *labeled old* expressions. When translating calls to functions with escaping references, a new label is created just before the call. When applying the generated wand, all *old* expressions are replaced by labeled old expressions of the form *old[l]*, where l refers to the generated label. Consider a function `borrow`, that has the wand shown in Listing 4.11 as a postcondition.

```
1   acc(res.Int__v, write) --* acc(valid__Foo(f), write) &&
2       (unfolding acc(valid__Foo(f), rd) in  f.Foo__z) ==
3       old(unfolding acc(valid__Foo(f), rd) in  f.Foo__z)
```

Listing 4.11: Magic wand containing and old expression.

Listing 4.12 shows how a call to the function `borrow` is translated and how the generated wand is applied. Note that while the wand in the postconditions of `borrow` refers to the formal parameters of the function (the ones in the function declaration), when applying the wand the formal parameters are replaced by the actual parameters of the call (the ones supplied at the call site).

The second case deals with postconditions that refer to the same location the returned reference points to. Listing 4.13 shows such a case. The function `borrow` returns a mutable reference to `f.v`, but in the postcondition, `f.v` is used to refer to the same data. This means there are two references to `f.v`: the return value `res` and `f.v`. In our translation, we forbid the use of `f.v` in this case, because otherwise inconsistent magic wands could be created. For example consider a wand for `borrow`, that contains an assertion for `f.v`

```
label call1
cr := borrow(cf, rd)
// ...
apply acc(cr.Int__v, write) --* acc(valid__Foo(cf), write) &&
      (unfolding acc(valid__Foo(cf), rd) in  cf.Foo__z) ==
      old[call1](unfolding acc(valid__Foo(cf), rd) in  cf.Foo__z)
```

Listing 4.12: Translation of the call to `borrow` from Listing 4.8

`== 0` on the right side. Between calling the function `borrow` and applying the generated magic wand, the returned reference could be used to modify `f.v` which conflicts with the assertion on the right side of the generated wand. From a Rust perspective, this restriction also seems reasonable, because while the returned reference is active, `f.v` cannot be used. Note that we allow the use of fields to which no references escape in postconditions, because there can be no references that could be used to modify such fields while the returned reference is active.

```
#[postcond="f.v == 0"]
fn borrow<'a>(f: &'a mut Foo)
              -> &'a mut i32
{
  f.v = 0;
  f.z = 0;
  &mut f.v
}
```

Listing 4.13: Postcondition that refers to the field the returned reference points to.

4.2.2 Specification for Loops

For loops, the problem with functional specification is similar to the case discussed in the previous section: functions that return references. Permissions to the data structure traversed by a loop are restored at some point after the loop, which means the already traversed parts of a data structure cannot be used in loop invariants. Invariants that talk about the traversed data structure must reason about the state of the data structure as it will be, once the permissions are restored after the loop. Again, this behavior can be encoded by adding those assertions to the right side of the generated wand.

We will illustrate this approach using the example shown in Listing 4.14, which contains a loop that traverses a linked list using a mutable reference. For this loop, the following wand will be created and maintained:

$$acc(valid__List(current), write) \mathrel{-\!\!*} acc(valid__List(list), write).$$

Note the two invariants in Listing 4.14, which use a function `length` which returns the length of a list and `elems`, which returns the values stored in the list as a sequence. The first two invariants involves `current` and `list` in the method's pre-state and state that the elements from the current element to the last element have not changed. Those assertions can be maintained as a normal loop invariants. The third invariant however refers to `list` in the state as it will be once the borrow to `list` ends. Therefore this assertion is added to the right side of the generated wand, which allows us the reason about the state as it will

be once the reference to `list` expires. Note that we also use `current` in the assertion that goes on the right side of the wand. The reference `current` borrows `list` and permissions to `current` are also on the left side of the wand. In our translation, those references refer to the state in which the wand is applied, and we evaluate expressions using `lhs()` to refer to this state. In Silver, `lhs` can be used on the right side of a magic wand to evaluate an expression in the state in which the magic wand is applied. Listing 4.15 shows the generated loop invariants in Silver.

```
1 fn append_iter(list: &mut List, element: i32) {
2     let mut current: &mut List = list;
3     let mut c = 0;
4
5     #[invariant="0 <= c && c <= |old(elems(list))|"]
6     #[invariant="elems(current) == old(elems(list))[c..]"]
7     #[invariant="elems(list) == old(elems(list))[0..c] ++ elems(current)"]
8     loop {
9         let temp = current;
10        if let List::Node(_, ref mut tail) = *temp {
11            current = tail;
12            c = c + 1;
13        } else {
14            current = temp;
15            break;
16        }
17    }
18
19    *current1 = List::Node(element, Box::new(end));
20 }
```

Listing 4.14: Loop that traverses a linked list mutably with specification stating the elements remain unchanged.

```
1  define LHS acc(valid__List(current), write)
2  define RHS acc(valid__List(list), write) &&
3          elems(list) == old(elems(list))[0..c] ++ lhs(elems(current))
4
5  invariant acc(valid__List(current), write)
6  invariant 0 <= c && c <= |old(silver_elems(list))|
7  invariant elems(current) == old(silver_elems(list))[c..]
8  invariant LHS --* RHS
```

Listing 4.15: Translation of the loop invariants from Listing 4.14.

Now let us briefly summarize how loop invariants for mutable data structures are interpreted:

- assertions involving the beginning of the traversed data structure are added to the right side of the wand generated for the loop
- expressions involving the current element of the loop iteration are evaluated in the state the wand is applied in using *lhs*.

Finally, we briefly discuss functional specification for loops that traverse a data structure

using an immutable reference. When using an immutable reference to traverse a data structure in a loop, only a fraction of the available permissions are used to traverse the data structure. This in turn means we retain some permissions to the data structure outside the loop body and the verifiers can automatically deduce that the data cannot be modified by the loop. It is therefore sufficient to refer to the data structure in the state before the loop body. In our specification language, we provide a function *pre* to refer to the state before the loop body. When translating a loop, a new label is added just before the loop. Usages of *pre* in loop invariants are translated to labeled-old expressions using the label generated for the loop. Listing 4.16 shows a Rust function to calculate the sum of the elements in a list. In the invariant of the loop, *pre* is used to refer to the state before the loop and can be translated to an invariant in Silver; there is no need to add it to the generated magic wand.

```
1  #[postcond="res == old(sum_rec(list))"]
2  fn sum_iter(list: &List) -> i32 {
3      let mut current = list;
4      let mut sum = 0;
5
6      #[invariant="sum == pre(sum_rec(list)) - sum_rec(current)"]
7      loop {
8          match *current {
9              List::Node(ref v, ref tail) => {
10                 sum += *v;
11                 current = tail;
12             },
13             _ => break
14         }
15     }
16     sum
17 }
```

Listing 4.16: Loop to sum all elements of a list, with specification.

Chapter 5

Technical Details

While the previous sections provided general rules used to translate Rust programs to Silver, in this section we discuss some interesting technical aspects of Rust2Viper.

5.1 Rust Compiler Plugin

Compiler plugins in Rust provide an interface to access the high-level intermediate representation *HIR* the compiler generates for a Rust program. Rust's HIR is similar to an abstract syntax tree representation, where Rust macros are expanded already. Most commonly, compiler plugins are used to implement static checks, that use the HIR to warn about problematic patterns in the source program. Those plugins are referred to as lints and there is a wide array of lints warning about common mistakes in Rust programs, such as unused variables or variables that are written but never read. While such lints are the most common use case for compiler plugins, a compiler plugin simply provides a way to execute Rust code to analyze a Rust programs at compile time. Compiler plugins are executed after the analysis passes of the compiler, such as type and borrow checking, are done, but before code generation. This means compiler plugins have access to the full type information computed by the compiler. The Rust compiler also provides infrastructure to emit error messages and warnings from compiler plugins.

Rust2Viper is implemented as a compiler plugin. From a high level, Rust2Viper operates as follows to verify a Rust program. First, the compiler checks the program and generates the HIR for that program. Then the Rust2Viper plugin is executed. The plugin then traverses the HIR and generates a Silver program from the HIR. This Silver program is then passed on to a verifier. Error messages raised by a verifier are emitted using the same infrastructure also used to emit error messages of the Rust compiler. Mapping error messages from the verifiers back to error messages in the Rust program is discussed in Section 5.3.

Implementing Rust2Viper as a compiler plugin comes with one disadvantage, however. At the moment, compiler plugins are considered an unstable feature in Rust, because the internal data structures (e.g. the HIR or the type representation) and APIs accessible by compiler plugins are subject to change. Currently, unstable features are only supported in "nightly" builds of the Rust compiler and this means only "nightly" builds can be used to compile and run Rust2Viper. Because of the unstable nature of compiler plugins at the moment, Rust2Viper most likely requires small updates over time, to account for changes

in the unstable APIs and data structures used.

Another interesting aspect of Rust2Viper's implementation is the integration of the borrow checker. While Rust programs are checked by the borrow checker before compiler plugins are executed, the data generated by the borrow checker is just used to check Rust's borrow rules and is not passed on to later stages of the compiler. We therefore ship a slightly modified version of the borrow checker with Rust2Viper. This version of the borrow checker is then invoked directly from Rust2Viper to gather the necessary borrow information for each function in the Rust program. For Rust2Viper, we also had to make a small but important modification to the borrow checker. The original version of the borrow checker does not track information about borrows of immutable objects, because they can never be borrowed mutably, which makes checking the relevant borrow rules trivial. For our translation however, the borrow information is used to generate ghost operations and therefore we extended the borrow checker to track information for immutable objects as well. This is convenient for our current implementation of Rust2Viper, as the same procedures can be used to generate ghost operations for mutable and immutable data. But we think it should be possible the generate most ghost operations for immutable data without borrow information for that data. Only our loop analysis would require major changes, because it relies on the information provided by the borrow checker quite heavily for both immutable and mutable data.

5.2 Viper as a Service

During this thesis, we also implemented *Viper as a Service* or *VaaS*. It provides a simple HTTP API, that can be used to verify a Silver program with either Carbon or Silicon. In order to verify a Silver program, a client sends a request containing the Silver source code to VaaS, which runs either Silicon or Carbon and sends back the results. Vaas is useful for two reasons.

First, while Rust2Viper is implemented in Rust, both Silicon and Carbon are implemented in Scala and run on the Java Virtual Machine. This means, after Rust2Viper generated a Silver translation, an instance of the JVM has to be started in order to run a Viper verifier. This process takes a relatively long time, about one to two seconds on a on a Linux machine with an Intel i5-4670S CPU with 3.10GHz and 8 GB of RAM, which slows down the verification work-flow. VaaS eliminates the startup cost, because it is implemented as a server in Java itself, which means many requests can be handled by a single VaaS instance running in a single JVM instance.

Second, installing the tools of the Viper verification infrastructure requires some effort, because the tools are spread out across different repositories and require the Z3 theorem prover. Handing the actual verification off to VaaS means the users do not need to setup the Viper tools locally, which reduces the barrier of entry and makes it easier for new users to use Rust2Viper.

5.3 Mapping Error Messages from Silver to Rust

While Rust2Viper generates Silver programs and the Viper verifiers raise error messages in terms of the Silver programs, those error messages are only loosely related to the assertions the user provided in the original Rust program. Therefore the error messages for

the generated Silver program must be mapped back to the original Rust program.

Unlike other frontends for the Viper verification infrastructure, such as Scala2Sil [8] or Chalice2Sil [23], Rust2Viper is not implemented in a language that runs on the JVM. This means Rust2Viper cannot build a Silver AST directly, as Scala2Sil does for example. Rust2Viper has to generate a Silver source file, which is then passed on to a verifier. When building the AST directly using a library provided by Viper, arbitrary information can be attached to the generated AST nodes and this information can be retrieved in case errors are raised by a verifier. Thus building the AST directly has the advantage that information for error reporting can be attached to the nodes of the AST.

At a source level, the Silver language does not provide a similar way to add annotations to statements, contracts or expressions. Therefore Rust2Viper has to manually track additional information to map error messages of the verifiers back to the original Rust file. In order to do that, Rust2Viper operates under the following assumption: the automatically generated contracts and ghost operations should never cause errors during verification. Errors related to automatically generated parts of a Silver program are treated and reported as bugs in Rust2Viper. This in turn means Rust2Viper only has to map error messages involving assertions provided by the user back to the Rust source program. We use the example in Listing 5.1 to showcase the mapping of error messages back to the Rust program. Note that this listing includes all necessary attributes to invoke the Rust2Viper plugin, which were omitted in the previous examples for brevity. Figure 5.1 shows the error messages reported by Rust2Viper for Listing 5.1. Both statements that cause the verification to fail, the false assertion in line 6 of Listing 5.1 and the call in line 9, are highlighted. In addition to that, the failed precondition for the call in line 9 is highlighted as well.

```
1  #![feature(plugin, custom_attribute)]
2  #![plugin(rust2viper)]
3  #![r2s_header]
4
5  fn main() {
6      static_assert!("1 == 2");
7  }
8  fn call() {
9      test1();
10 }
11 #[precond="false"]
12 fn test1() {}
```

Listing 5.1: Rust functions with an assertion and a precondition that cannot be verified.

In order to emit such error messages, Rust2Viper tracks the locations in the Silver source file that contain assertions provided by the user and maps those locations to the locations in the original source file, where the user provided the assertions. The error messages and line numbers reported by a Viper verifier can then be used to emit the correct error message with respect to the Rust source file. While this approach works, having support for high-level error messages in the Silver language would provide a cleaner way to implement error reporting for tools that are not based on the JVM. This issue could

```
errors.rs:10:5: 10:30 error: Assertion failed
errors.rs:10     static_assert!("1 == 2");
                 ^~~~~~~~~~~~~~~~~~~~~~~~~
errors.rs:14:5: 14:12 error: A precondition might not hold
errors.rs:14     test1();
                 ^~~~~~~
errors.rs:17:1: 17:19 note: This precondition might not hold at the call site
errors.rs:17 #[precond="false"]
             ^~~~~~~~~~~~~~~~~~
error: Verification failed
```

Figure 5.1: Errors reported by Rust2Viper for Listing 5.1.

be addressed, for example, by adding annotations to the Silver language. Annotations could then be used to add error messages in terms of the original source program to the generated Silver assertions. When the verification of an assertion fails, the verifier could report the user-provided message.

Chapter 6

Evaluation

In this section we evaluate the following aspects of Rust2Viper:

- Which parts of Rust can be handled by Rust2Viper?
- What kind of assertions are required to verify functional properties of Rust programs?
- How does the verification run-times of Silver programs generated with Rust2Viper compare against handwritten Silver programs?

6.1 Supported Subset of Rust

At the moment, Rust2Viper supports most of Rust's basic control structures, such as `if`, `loop`, `while` and `match`, struct and enum declarations, as well as statements and expressions. The supported expressions include boolean and arithmetic expressions, field accesses as well as function calls. An EBNF grammar for the supported subset of Rust can be found in Listing B.1. We use $*_,$ to refer to zero or more repetitions separated by a comma. Not supported at the moment are arithmetic or boolean expressions that use non-pure functions as operands. This however is only a minor limitation, because temporary variables can be introduced for the results of non-pure functions, which then can be used in arithmetic or boolean expressions. Besides that, there is currently no support for Rust's trait system. This is currently an essential limitation of Rust2Viper, because the use of traits is very widespread in real-world Rust code. In fact, some features of Rust are based on traits, e.g. `for` loops depend on the *Iterator* trait or using copy semantics for a type is done by implementing the *Copy* trait.

But while Rust2Viper does not support traits, plenty of realistic code examples can be handled. At the moment, the Rust2Viper test suite consists of about 100 test cases. Some of those test cases are regression tests that test our encoding of certain features of Rust, e.g. testing various borrow scenarios. The rest of the test suite consists of tests that are inspired by existing Rust code. There are

- test cases inspired by Rust types, e.g by Rust's *Option* type, such as the example in Listing A.1
- test cases dealing with recursive data structures, e.g. linked list and binary tree implementations, such as the example in Listing A.2
- test cases collected from existing libraries, such as the example in Listing A.3, which shows parts of a parser written in Rust, including specification

Name	Silicon			Carbon		
	Rust2Viper	Manual	Speedup	Rust2Viper	Manual	Speedup
cell	0.199s	0.039s	5.10x	3.848s	3.263s	1.17x
counter	2.131s	0.096s	22.19x	5.619s	3.428s	1.63x
list	0.693s	0.222s	3.12x	5.992s	4.922s	1.20x

Table 6.1: Verification run-times and speedup of Silver programs generated by Rust2Viper and handwritten versions.

Those examples show that our approach for generating permission-related specification automatically based on Rust types and borrow information works well for a wide range of programs. Functional specification for those test cases can be expressed without user-provided permission specification. The only known limitation at the moment are loops. Rust2Viper supports the common pattern where a single reference is used to traverse a data structure. Some more complex loops, where multiple references are used to traverse a data structure, are not supported at the moment.

6.2 Performance of Generated Silver Code

In previous sections we already discussed potential disadvantages of our encoding from a performance perspective. In this section we compare the verification run-time of Silver programs generated by Rust2Viper to similar, handwritten Silver programs. Major difference between our encoding and handwritten Silver programs are:

- Rust2Viper always uses predicate instances for arguments in contracts, which means *unfolding* expressions must be used to access fields of arguments in contracts
- Rust2Viper uses objects to store integer and boolean values

For our benchmark, we use programs that are translated to Silver programs with several *unfolding* expressions and integer objects, that can be removed in the handwritten versions. Table 6.1 shows the result of our benchmark. In the table we report the verification run-times of the Silver programs generated by Rust2Viper and of the handwritten versions. We report the average run-time of 5 consecutive runs on a Linux machine with an Intel i5-4670S CPU with 3.10GHz and 8 GB of RAM measured using *perf*, which uses the performance counter subsystem of the Linux kernel. We also report the speedup of the handwritten programs over the ones generated by Rust2Viper. The data clearly shows that the handwritten versions beat the programs generated by Rust2Viper in all cases. Interestingly, the speedup is bigger for Silicon than for Carbon by a large margin. This suggests that Carbon deals better with the overhead introduced by Rust2Viper. By far the biggest speedup is achieved for the *counter* test case with Silicon. This test case contains contracts with pattern matching, which are translated to expressions with a high number of *unfolding* expressions. The high speedup for this test case suggests that *unfolding* expressions are expansive in Silicon. Also, adding more objects to the heap increases the verification run-time of Silicon quite substantially.

Chapter 7

Conclusion

In this thesis, we presented the ideas and initial implementation of Rust2Viper, a verifier for Rust, that translates Rust to Silver code, which can be verified using the Viper verification infrastructure.

As main contribution in this thesis, we presented rules to use information provided by Rust's type system and borrow checker to generate permission-related specification automatically. We also showed how key concepts of Rust, such as ownership and borrowing, can be encoded automatically. The ideas presented in this thesis are used to implement Rust2Viper, a verifier that uses a permission-based logic internally, but hides the permission-related specification form the user. This approach reduces the amount of specification required and allows users to focus on functional specification.

Rust2Viper uses most features available in the Silver language to translate Rust programs, which allowed us to analyze how well those features interact for a wide range of programs and we found about 20 new bugs in the existing tools. At the moment, Rust2Viper is powerful enough to verify functional properties of recursive functions and data structures as well as simple loops that iterate over recursive data structures. We also describe how specification can be added to magic wands generated by Rust2Viper in a way that is transparent to the user.

7.1 Related Work

In this thesis, we use the notion of a capability-based type system introduced by Boyland et al. [7]. We apply the idea of references with associated capabilities to Rust's type system, but capability-based type systems have also been proposed as extensions for other languages, for example Gordon et al.'s subsequent type system for C# [13].

There also exist several other academic projects, that use expressive type systems for software verification. For example, the SYMPLAR tool of Bierhoff [3] uses special type annotations to express symbolic permissions on the type-level and the Plural tool of Bierhoff and Aldrich [4] provides verification of typestate-based protocols using permissions. Those systems typically focus on checking certain functional properties, while with Rust2Viper we support verification of more general functional specification.

7.2 Future Work

There are several directions for further work.

7.2.1 Extend Unsafe Support

At the moment, the support of unsafe Rust code is very limited. We think there is some interest in the Rust community to increase the static safety guarantees of unsafe Rust code, which makes improving the support for unsafe code in Rust2Viper very interesting. Guarantees that could potentially be verified with Rust2Viper are for example, that raw pointers point to a valid memory addresses or that accessed memory was initialized correctly. A limitation of the current implementation is that information required to verify unsafe code blocks must be carried around explicitly, which requires additional specification in client code. An interesting direction for future work would also be to evaluate if and how static analysis could be used to automatically generate specification for unsafe code.

7.2.2 Support More Rust Features

Traits Traits are a fundamental element of Rust's type system, but at the moment traits are not supported by Rust2Viper. Adding support for traits would increase the number of interesting programs that can be handled by Rust2Viper by a large margin. There exists some related prior work on verifying traits. A deductive proof system for a trait-based object-oriented language is introduced in [10] and Bühlmann presented a system for permission-based verification of traits [9]. Traits in Rust are less powerful than traits in Scala (traits in Rust cannot contain data), which means some ideas from [10] and [9] could be applied to traits in Rust.

Type Parameters Type parameters are another feature not supported by Rust2Viper at the moment, but they are used extensively in library code. If Rust2Viper should be used to verify parts of Rust's standard library, supporting type parameters is crucial.

Iterators & Closures Iterators are widely used in Rust and it is considered idiomatic to use iterators and adapter functions like *fold* or *map* wherever possible. Encoding information about the content of an iterator as well as closures that can be applied to elements of an iterator poses interesting challenges. Rust's type system distinguishes between closures that take and do not take ownership of objects in the enclosing environment. Also, when a closure borrows some data from the enclosing environment, this data remains borrowed until the closure goes out of scope. This additional information should be helpful when adding specification for closures.

7.3 Acknowledgements

First and foremost, I would like to thank Dr. Alexander J. Summers for introducing me to the arcane powers of magic wands and many long and inspiring discussions, providing new perspectives on the problems at hand as well as valuable feedback on the written report. Furthermore, I would also like to thank Prof. Dr. Peter Müller for giving me the opportunity to work on this thesis at the Chair of Programming Methodology. Additionally, I would like to thank Dr. Alexander J. Summers and Malte Schwerhoff for fixing various bugs I reported and everybody else at the Chair of Programming Methodology

at ETH for their feedback regarding my work. Finally, I would like to thank my family for supporting me during my studies at ETH.

Appendix A

Rust Examples

```
1  enum MyOption {
2      MySome(i32),
3      MyNone
4  }
5
6  #[postcond="matching *x in (
7      MySome(v) => res == v,
8      MyNone => res == d
9  )"]
10 fn unwrap_or(x: &MyOption, d: i32) -> i32 {
11     match *x {
12         MyOption::MySome(x) => x,
13         MyOption::MyNone => d
14     }
15 }
```

Listing A.1: Rust implementation of an option type and an unwrap_or function with specification.

```rust
pub enum List {
    Node(i32, Box<List>),
    Nil
}

#[pure_fn]
pub fn length(l: &List) -> i32 {
    match *l {
        List::Node(_, ref tail) => {
            1 + length(tail)
        },
        List::Nil => 0
    }
}

#[pure_fn]
#[precond="length(l) > 0 && i < length(l)"]
pub fn item_at(l: &List, i: i32) -> i32 {
    match *l {
        List::Node(ref c, ref tail) =>
            if i == 0 { *c } else { item_at(tail, i-1) },
        List::Nil => 0
    }
}

#[postcond="length(l) == old(length(l) )"]
#[postcond="forall i: Int :: i >= 0 && i < length(l) ==>
    item_at(l, i) == old(item_at(l, i)) + v"]
pub fn increment(l: &mut List, v: i32) {
    match *l {
        List::Node(ref mut c, ref mut t) => {
            *c = *c + v;
            increment(t, v);
        }
        List::Nil => {}
    };
}
```

Listing A.2: Rust implementation of linked list.

```
struct TextPosition {
    row: u64,
    column: u64,
}

impl TextPosition {
    #[postcond="res.row == 0"]
    #[postcond="res.column == 0"]
    pub fn new() -> TextPosition {
        TextPosition { row: 0, column: 0 }
    }

    #[postcond="self.column == old(self.column) + count"]
    pub fn advance(&mut self, count: u8) {
        self.column += count as u64;
    }
    // ...
}
```

Listing A.3: Parts of a parser written in Rust, with specification.

Appendix B

Supported Subset of Rust

```
⟨Rust-program⟩   ::= ( ⟨Struct⟩ | ⟨Enum⟩ | ⟨Function⟩ )
⟨Ident⟩          ::= '[a-zA-Z_][a-zA-Z0-9_]*' // a Rust identifier, specified as regex
⟨Ty⟩             ::= ⟨Ident⟩ | '&' ⟨Lifetime⟩ 'mut' ⟨Ty⟩
⟨Lifetime⟩       ::= ''' ⟨Ident⟩
⟨Struct⟩         ::= 'struct' ⟨Ident⟩ '{' ⟨Struct-Field⟩ *, '}'
⟨Struct-Field⟩   ::= ⟨Ident⟩ ':' ⟨Ty⟩
⟨Enum⟩           ::= 'enum' ⟨Path⟩ '{' ⟨Enum-Variant⟩ *, '}'
⟨Enum-Variant⟩   ::= ⟨Ident⟩ | ⟨Ident⟩ '(' ⟨Ty⟩ *, ')'
⟨Function⟩       ::= 'fn' ⟨Ident⟩ '<' ⟨Lifetime⟩ *, '>' '(' ⟨Fn-Arg⟩ *, ')' '{' ⟨Block⟩ '}'
⟨Block⟩          ::= ⟨Stmt⟩ *
⟨Stmt⟩           ::= ⟨StmtDecl⟩ ';' | ⟨Expr⟩ ';'
⟨StmtDecl⟩       ::= 'let' 'mut'? '=' ⟨Expr⟩
⟨Expr⟩           ::= 'if' ⟨Expr⟩ ⟨ExprBlock⟩ ( 'else' ⟨ExprBlock⟩ )?
                   | 'match' <Expr> '{' ⟨MatchArm⟩ *, '}'
                   | 'loop' ⟨ExprBlock⟩
                   | 'while' ⟨Expr⟩ ⟨ExprBlock⟩
                   | ⟨Expr⟩ '.' ⟨Ident⟩       // field access
                   | '&' 'mut'? ⟨Expr⟩       // borrow
                   | ⟨Expr⟩ '(' ⟨Expr⟩ *, ')'       // function call
                   | ⟨Expr⟩ ⟨BinOp⟩ ⟨Expr⟩
                   | ⟨Ident⟩ '{' ⟨Field-Init⟩ *, '}'       // struct initialization
                   | 'true' | 'false' | ⟨IntConst⟩
⟨ExprBlock⟩      ::= '{' ⟨Stmt⟩* '}'
⟨MatchArm⟩       ::= ⟨EnumVariant⟩ '=>' ⟨Expr⟩
⟨Field-Init⟩     ::= ⟨Ident⟩ ':' ⟨Expr⟩
⟨BinOp⟩          ::= '+' | '-' | '*' | '/' | '||' | '&&' | '==' | '!=' | += | '-=' | '*=' | '/='
⟨IntConst⟩       ::= '[0-9]+'       // a integer constant, specified as regex
```

Listing B.1: Rust syntax supported by Rust2Viper

Bibliography

[1] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. "The Spec# programming system: An overview." In: *Construction and analysis of safe, secure, and interoperable smart devices*. Springer, 2004, pp. 49–69.

[2] Alexis Beingessner. "You can't spell trust without Rust." MA thesis. Ottawa, Ontario, Canada: Carleton University, 2015.

[3] Kevin Bierhoff. "Automated program verification made SYMPLAR: symbolic permissions for lightweight automated reasoning." In: *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. ACM. 2011, pp. 19–32.

[4] Kevin Bierhoff and Jonathan Aldrich. "PLURAL: checking protocol compliance under aliasing." In: *Companion of the 30th international conference on Software engineering*. ACM. 2008, pp. 971–972.

[5] Stefan Blom and Marieke Huisman. "The VerCors Tool for verification of concurrent programs." In: *FM 2014: Formal Methods*. Springer, 2014, pp. 127–131.

[6] John Boyland. "Checking interference with fractional permissions." In: *Static Analysis*. Springer, 2003, pp. 55–72.

[7] John Boyland, James Noble, and William Retert. "Capabilities for sharing: A generalization of uniqueness and sharing." In: *European Conference for Object-Oriented Programming (ECOOP)*. 2001.

[8] Bernhard F Brodowsky. "Translating Scala to Sil." MA thesis. Eidgenössische Technische Hochschule Zürich, Department of Computer Science, 2013.

[9] Andreas Bühlmann. "Permission-based verification of subclassing and traits." MA thesis. Eidgenössische Technische Hochschule Zürich, Department of Computer Science, 2013.

[10] Ferruccio Damiani et al. "Verifying traits: A proof system for fine-grained reuse." In: *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs*. ACM. 2011, p. 8.

[11] The Rust Developers. *RFC 560 Integer Overflow*. 2014. URL: `https://github.com/rust-lang/rfcs/blob/7fc338e5b4cdc46678cbcd20bbd2871f0ef34367/text/0560-integer-overflow.md` (visited on 03/11/2016).

[12] The Rust Developers. *The Rust Programming Language*. 2016. URL: `https://doc.rust-lang.org/stable/book/` (visited on 02/16/2016).

[13] Colin S Gordon et al. "Uniqueness and reference immutability for safe parallelism." In: *ACM SIGPLAN Notices*. Vol. 47. 10. ACM. 2012, pp. 21–40.

[14] Dan Grossman et al. "Region-based memory management in Cyclone." In: *ACM Sigplan Notices*. Vol. 37. 5. ACM. 2002, pp. 282–293.

[15] Stefan Heule et al. "Abstract read permissions: Fractional permissions without the fractions." In: *Verification, Model Checking, and Abstract Interpretation*. Springer. 2013, pp. 315–334.

[16] Stefan Heule et al. *Verification condition generation for permission logics with abstract predicates and abstraction functions*. Springer, 2013.

[17] Bart Jacobs et al. "VeriFast: A powerful, sound, predictable, fast verifier for C and Java." In: *NASA Formal Methods*. Springer, 2011, pp. 41–55.

[18] K Rustan M Leino. "Dafny: An automatic program verifier for functional correctness." In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer. 2010, pp. 348–370.

[19] K Rustan M Leino and Peter Müller. "Modular verification of static class invariants." In: *FM 2005: Formal Methods*. Springer, 2005, pp. 26–42.

[20] Peter Müller, Malte Schwerhoff, and Alexander J Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning." In: *Verification, Model Checking, and Abstract Interpretation*. Springer. 2016, pp. 41–62.

[21] Peter O'Hearn, John Reynolds, and Hongseok Yang. "Local reasoning about programs that alter data structures." In: *Computer science logic*. Springer. 2001, pp. 1–19.

[22] Matthew J Parkinson and Alexander J Summers. "The relationship between separation logic and implicit dynamic frames." In: *Programming Languages and Systems*. Springer, 2011, pp. 439–458.

[23] Viper Project. *Chalice2Silver*. 2016. URL: `https://bitbucket.org/viperproject/chalice2silver` (visited on 02/24/2016).

[24] John C Reynolds. "Separation logic: A logic for shared mutable data structures." In: *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*. IEEE. 2002, pp. 55–74.

[25] Malte Schwerhoff and Alexander J Summers. "Lightweight Support for Magic Wands in an Automatic Verifier." In: *29th European Conference on Object-Oriented Programming*. 2015, p. 614.

[26] Max Planck Institute for Software Systems The Foundations of Programming group. *Chalice2Silver*. 2016. URL: `http://plv.mpi-sws.org/rustbelt` (visited on 03/29/2016).

[27] John Toman, Stuart Pernsteiner, and Emina Torlak. "Crust: A Bounded Verifier for Rust (N)." In: *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE. 2015, pp. 75–80.

[28] Julian Tschannen et al. "AutoProof: Auto-active functional verification of object-oriented programs." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 566–580.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Rust2Viper: Building a Static Verifier for Rust

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Hahn	Florian

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Zurich, 07.04.2016

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.