Master Thesis

Extended Support for Borrowing and Lifetimes in Prusti

Lorenz Gorse

13th October 2020

Advisors:
Prof. Dr. Peter Müller and Dr. Alexander Summers

Abstract

Re-borrowing functions in the context of Rust return aliases (in the form of references) of reference-typed arguments. Prusti is a verifier for Rust programs; it supports a subset of the language, including re-borrowing functions that return a single reference. Specifications of these functions must capture the effects of the aliasing – concretely, modifications through the returned reference are eventually observable in the aliased data. This work extends the existing support for re-borrowing functions to instances where multiple references are returned. An evaluation demonstrates the practicality of the solution. The problem can be generalized further by allowing structures that contain references. We explore some of the significant challenges introduced by this and describe approaches that future work could develop further in order to add support to Prusti.

Acknowledgements

I would like to thank my main supervisor, Alexander Summers, for making this thesis possible. Our weekly discussions always left me with a deeper understanding of the problem and many directions in which to explore further. Occasional digressions on the intricacies of the English language and other more or less related topics provided a wonderful balance. I would also like to thank Federico Poli and Vytautas Astrauskas for sharing their seemingly unlimited knowledge of Prusti with me – their support was invaluable whenever I was stuck or needed feedback on an idea. Finally, I would like to thank Peter Müller for leading a research group in which everybody is welcoming and happy to help, and I would like to thank all the people that make up this group.

Contents

1. Introduction **6**

2. Background **10**
2.1. Rust 10
2.2. Polonius 13
2.3. Viper 17
2.4. Prusti 22
2.4.1. Type Encoding 23
2.4.2. Functions and Function Calls 24
2.4.3. Re-Borrowing Functions 25

3. Re-Borrows of Simple References **28**
3.1. Re-Borrow Relationships 29
3.2. Pledge Syntax and Semantics 30
3.3. Permission Management 32
3.4. Specifications 41
3.5. Clients 48
3.5.1. Generating the Expiration Statements 51
3.5.2. Assembling the Expiration CFG 53
3.6. Implementation 56
3.6.1. Pledge Parsing 58
3.6.2. Re-Borrow Relationships 58
3.6.3. Expiration Tool Construction 59
3.6.4. Expiration Tool Encoding 60
3.6.5. Pledge Dependencies 61
3.6.6. Client-Side Expiration 63
3.6.7. Differences 63
3.7. Evaluation 66
3.7.1. Examples 67
3.7.2. Performance 70
3.8. Future Work 72
3.8.1. Phrasing the Expiration Tool in Terms of Lifetimes 72
3.8.2. Re-Borrows in Loops 73
3.8.3. Shared Re-Borrows of Mutable References 74

4. References Inside Structs 76
4.1. Permission Management 76
4.2. Re-Borrow Relationships 81
4.2.1. Borrow Sources 83
4.2.2. Borrow Sinks 84
4.3. Pledges 85
4.3.1. Single Non-Recursive Struct 85
4.3.2. Single Recursive Struct 86

A. Terms and Notation 88

1. Introduction

Proving the correctness of programs is hard. This is especially true for languages like C++, where pointers (and, to some extent, references) allow the programmer to mutate memory almost at will. A correctness proof must account for this power of pointers and provide arguments that the modification of memory behind one pointer does not unexpectedly invalidate what is known about the memory behind another pointer. With the release of Rust in 2010, a much more restricted approach to memory and pointers became mainstream. Rust's novel type system allows the compiler to accurately reason about the references (pointers, effectively) that are involved in a program and enforce strong guarantees. One of these guarantees is that at any point of the execution, there is at most one way to access any given memory region mutably. This makes the verification of the following function possible without any further information:

$$
\begin{array}{l}
\textbf{fn } f_1(x : \&\textbf{mut } \mathrm{u32}, y : \&\textbf{mut } \mathrm{u32})\ \{ \\
\quad \textbf{let } n = *x; \\
\quad *y = 0; \\
\quad \mathrm{assert!}(*x = n); \\
\}
\end{array}
$$

Only because the compiler guarantees that $*x$ and $*y$ don't describe the same memory region (we say they don't alias) is the final assertion warranted. The equivalent program in C++, which does not guarantee the non-aliasing, would not be valid.

To maintain these guarantees, Rust blocks access to variables while they have an alias that is still in use. The following program is rejected by the compiler, because any access of x is forbidden while its alias rx is still used:

$$
\begin{array}{l}
\textbf{let mut } x = 0; \\
\textbf{let } rx = \&\textbf{mut } x; \\
x = x + 1; \\
*rx = *rx + 1;
\end{array}
$$

But if x is accessed after the final usage of rx, the compiler accepts the program and we

can observe the modification:

$$\begin{aligned}&\textbf{let mut } x = 0;\\&\textbf{let } rx = \&\textbf{mut } x;\\&*rx = *rx + 1;\\&\text{assert!}(x = 1);\end{aligned}$$

Re-borrowing functions, ie, functions that take mutable references as arguments and return mutable references in turn, enable another variation of this behavior. Consider the re-borrowing function

$$\begin{aligned}&\textbf{fn } f_2(rx : \&\textbf{mut } \text{u32}) \to \&\textbf{mut } \text{u32} \ \{\\&\quad \&\textbf{mut } *rx\\&\}\end{aligned}$$

and a program that calls f_2:

$$\begin{aligned}&\textbf{let mut } x = 0;\\&\textbf{let } rx = f_2(\&\textbf{mut } x);\\&*rx = *rx + 1;\\&\text{assert!}(x = 1);\end{aligned}$$

Because the rx returned by the call is an alias for x, we can observe the effects of the increment in x. The Rust compiler knows this and therefore blocks access to x while its alias rx is still used. The following program is rejected for this reason:

$$\begin{aligned}&\textbf{let mut } x = 0;\\&\textbf{let } rx = f_2(\&\textbf{mut } x);\\&x = x + 1;\\&*rx = *rx + 1;\end{aligned}$$

Prusti is a tool to verify the correctness of Rust programs. By default, it checks a program for the absence of panics and arithmetic under- and overflows. Specifications in the form of pre- and postconditions as well as assertions can contain deeper statements about the program's behavior, and Prusti verifies that these claims are valid. Prusti encodes Rust programs to Viper, a programming language with built-in support for specifications. The encoding is done in such a way that if the encoded program is accepted by Viper, the original Rust program is reported to be valid. If Viper rejects the encoded program, the reason for the rejection is translated to some part of the Rust program that is faulty. Viper implements a variation of implicit dynamic frames, which enables reasoning about a program's use of references and access to memory in general.

Under this paradigm, a piece of code holds some set of permissions and can only access memory it has permissions for. Prusti uses this feature of Viper to model the memory of the Rust programs that it verifies. Essentially, the encoded program holds, at any program location, permissions for all memory that is accessible in the corresponding program location of the original Rust program.

One part of the encoding carried out by Prusti that requires special attention concerns re-borrowing functions. First, their use of references requires advanced Viper features to maintain the property that the encoded program always holds permissions for accessible memory in the original Rust program. Second, useful descriptions of their behavior require the introduction of new specification syntax. Concretely, specifications must be able to state that modifications of the memory behind returned references is eventually reflected in data referenced by the argument references. For the function f_2 from before, this is written as follows:

$$
\begin{array}{l}
\#[\textbf{after_expiry}(*rx = \text{before_expiry}(*\text{result}))] \\
\textbf{fn } f_2(rx : \&\textbf{mut } \text{u32}) \rightarrow \&\textbf{mut } \text{u32 } \{ \\
\quad \&\textbf{mut } *rx \\
\}
\end{array}
$$

The specification (which is the part within the #[. . .]) states that after the returned reference (called result in the specification) expires (which just means it is not used again), the value of the integer referenced by the rx argument – ie, $*rx$ – equals the value of the integer referenced by the returned reference immediately before its expiration – ie, before_expiry(∗result). This specification allows a caller of f_2 to understand the relationship between the argument reference and return reference.

Today, Prusti supports re-borrowing functions with exactly one returned reference. However, Rust allows re-borrowing functions that are far more complex. The main contribution of this thesis and sole topic of Chapter 3 is the generalization of Prusti's current encoding approach to re-borrowing functions that return arbitrarily many references. We will see in Section 3.1 that the rules determining which input references are blocked after a call to a re-borrowing function get more complex, and the behavior we can observe as a result gets richer. The specification syntax, discussed in Section 3.2, must adapt to be able to describe the newly possible behavior. The more technical Sections 3.3 to 3.5 describe how the Viper encoding manages permissions and handles specifications. To this end, we introduce *expiration tools*, which are Viper resources that allow callers of re-borrowing functions to synchronize the accessible memory in Rust with the permissions they hold in the encoding. This thesis does not only describe a method for supporting re-borrowing functions with many returned references, it also implements the method in Prusti. The most important parts of the implementation are described in Section 3.6 and put in relation with Sections 3.1 to 3.5.

Re-borrowing functions can be generalized even further. Rust has structs, which are

composite types. They can contain references, as the following example illustrates:

$$\textbf{struct}\ S\langle 'a\rangle\ \{\ rx : \&'a\ \textbf{mut}\ \text{u32}\ \}$$

This defines a struct called S with a single field rx of type $\&'a$ **mut** u32. The $'a$ in the type is a lifetime which, loosely speaking, tells the Rust compiler when the data referenced by the reference is destroyed. Note that this lifetime is a generic parameter of S, which means we can create instances of S for arbitrary lifetimes. Now let's use S to define another re-borrowing function:

$$\begin{aligned}
&\textbf{fn}\ f_3\langle 'a\rangle(s_1 : \&'a\ \textbf{mut}\ S\langle 'a\rangle, s_2 : \&\textbf{mut}\ S\langle 'a\rangle)\ \{\\
&\quad s_2.rx = \&\textbf{mut}\ {*s_1}.rx;\\
&\}
\end{aligned}$$

Note the lack of a returned reference – in its stead, the s_2 argument serves as an output parameter. A program can create aliases by calling f_3, just like calling f_2 created aliases:

$$\begin{aligned}
&\textbf{let mut}\ x = 0;\\
&\textbf{let mut}\ s_1 = S\ \{rx : \&\textbf{mut}\ x\};\\
&\textbf{let mut}\ s_2 = S\ \{rx : \&\textbf{mut}\ 0\};\\
&f_3(\&\textbf{mut}\ s_1, \&\textbf{mut}\ s_2);\\
&{*s_2}.rx = {*s_2}.rx + 1;\\
&\text{assert!}(x = 1);
\end{aligned}$$

Structures with references pose new challenges. For example, they make re-borrowing functions involving a statically unbounded number of references possible. For this, just imagine a linked list type where the values are mutable references to, e.g., integers. In Chapter 4 we describe more examples and provide encoding ideas that future work can explore further.

2. Background

2.1. Rust

Rust is a new systems language aiming to achieve safety, concurrency, and speed. While low-level languages such as C are very fast, they allow the programmer to arbitrarily mutate the entire program memory using pointers, which can (and often does) introduce subtle bugs. For example, buffer overflows silently overwrite unrelated memory regions and dangling pointers access memory that has already been repurposed in the meantime. Incorrect use of pointers is the leading cause of security vulnerabilities in software [9], one of the most famous examples probably being OpenSSL's Heartbleed [10].

Rust solves a wide range of commonly occurring problems arising from incorrect use of pointers by imposing a strict memory management discipline on the programmer, enforced at compile time through the type system. At the heart of this discipline are the concepts of ownership [5], borrowing [6], and lifetime [4]:

Ownership. Every memory location is exclusively owned by a variable. When the variable goes out of scope, the memory location is deallocated.

Borrowing. A program part can operate on memory it does not own by borrowing a reference to it. A reference is nothing more than, using C lingo, a pointer, but with additional compile time checks to guarantee memory safety.

Lifetime. Every reference has a lifetime, which is the time during which the reference can be used to access the pointed-to memory location. The Rust compiler enforces two properties about lifetimes. First, every usage of the reference must happen within its lifetime (this imposes a minimal valid lifetime). Second, the variable owning the pointed-to memory location does not go out of scope during the lifetime (this imposes a maximal valid lifetime).

Besides memory safety, the second major application of these Rust concepts is to enforce safe concurrency. Specifically, Rust programs, at least when restricted to the *safe* subset of the language, cannot have data races[1]. A data race happens if one thread reads

[1] This is true as long as you discount soundness bugs in the compiler [2] that allow you to make tons of

memory that another thread is mutating at the same time. In Rust, where memory is accessed through references, this can only happen when two threads hold references to the same memory simultaneously. To ensure the absence of data races, the language distinguishes between *shared* and *mutable* references. Shared references provide read-only access to the referred memory, while mutable references provide full read-write access. Mutable references are also called *exclusive*, because the compiler checks that there is always either exactly one mutable reference or an arbitrary number of shared references to every variable. This enables a powerful property – modifying data behind a mutable reference cannot change any data behind other references, shared or mutable.

To maintain these guarantees, Rust must sometimes block access to one reference while another reference is still in use. The following example produces a compile error, because access to x is blocked while rx is still in use:

$$\begin{aligned}&\textbf{let mut } x = 0;\\&\textbf{let } rx = \&\textbf{mut } x;\\&x = x + 1;\\&*rx = *rx + 1;\end{aligned}$$

If access to x were not blocked, the memory owned by x could be mutated via two different ways – via x itself, and via rx. Flipping the last two lines resolves the error, because the last usage of the aliasing rx happens before the first access to x after taking the reference:[2]

$$\begin{aligned}&\textbf{let mut } x = 0;\\&\textbf{let } rx = \&\textbf{mut } x;\\&*rx = *rx + 1;\\&x = x + 1;\end{aligned}$$

Some function calls cause similar behavior in order to maintain the same guarantees. Consider a function that receives a mutable reference to a point and returns a mutable reference to the point's x-coordinate:

$$\begin{aligned}&\textbf{fn } f_4\langle' p\rangle(p : \&' p \textbf{ mut } \text{Point}) \rightarrow \&' p \textbf{ mut } \text{u32} \;\{\\&\quad \&\textbf{mut } p.x\\&\}\end{aligned}$$

Calling f_4 creates aliasing references, triggering the same blocking rules. The following example again produces a compile error, because p is blocked while the aliasing px is

money [7].

[2] The next section gives an alternative explanation of this behavior in terms of the (implicit) lifetimes involved in the program.

still used:

$$
\begin{aligned}
&\textbf{let mut } p = \text{Point } \{x : 0, y : 0\};\\
&\textbf{let } px = f(\&\textbf{mut } p);\\
&p.x = p.x + 1;\\
&{*px} = {*px} + 1;
\end{aligned}
$$

The compiler does not check f_4's definition for the creation of aliases. Instead, the mere possibility (as indicated by the lifetimes in the function signature) is enough.

We call the creation of references obtained via existing references *re-borrowing* (because the newly created reference borrows already borrowed data). Accordingly, f_4 is a *re-borrowing function*, because it returns a reference obtained from the argument reference. We call references passed into the function *input references* and references returned by the function *output references*.

Re-borrowing functions can involve multiple input and output references, as the following example shows:

$$
\begin{aligned}
&\textbf{fn } f_5\langle' p, ' q\rangle(p : \&' p \textbf{ mut } \text{Point}, q : \&' q \textbf{ mut } \text{Point}) \rightarrow (\&' p \textbf{ mut } \text{u32}, \&' q \textbf{ mut } \text{u32}) \ \{\\
&\quad (\&\textbf{mut } p.x, \&\textbf{mut } q.x)\\
&\}
\end{aligned}
$$

This function returns two references, one of type $\&'p$ **mut** u32 and another one of type $\&'q$ **mut** u32. At the end of the function, the first contains an alias for the data referenced by p, while the second contains an alias for the data referenced by q. The lifetimes in function signature – here $'p$ and $'q$ – restrict which input references a given output reference can re-borrow. In this case, the first output reference can only re-borrow p, while the second output reference can only re-borrow q. The rules that govern this are explored in more detail in Section 3.1. A caller of f_5 may look as follows:

$$
\begin{aligned}
&\textbf{let mut } p = \text{Point } \{x : 0, y : 0\};\\
&\textbf{let mut } q = \text{Point } \{x : 0, y : 0\};\\
&\textbf{let } (px, qx) = f(\&\textbf{mut } p, \&\textbf{mut } q);\\
&{*px} = {*px} + 1;\\
&\text{assert!}(p.x = 1);\\
&{*qx} = {*qx} + 1;\\
&\text{assert!}(q.x = 1);
\end{aligned}
$$

It is important to note that even though p is blocked while px is still used, the same does not hold for p and qx. This means we can already use p after the last usage of px, regardless of whether qx is used afterwards.

Rust also has structs, which allow the construction of composite data types. They can contain references, as the following definition of a struct S shows:

$$\textbf{struct}\ S\langle'a\rangle\ \{\ rx : \&'a\ \textbf{mut}\ \text{u32}\ \}$$

This enables another kind of re-borrowing function, where an alias for data referenced by a reference inside the struct is returned, or where an alias is stored inside a struct that the function receives a reference to (like an output parameter), or both. An example of this is the following function, which creates a reference to data referenced by the reference inside s_1 and stores it in s_2:

$$\begin{aligned}&\textbf{fn}\ f_6\langle'a\rangle(s_1 : \&'a\ \textbf{mut}\ S\langle'a\rangle, s_2 : \&\textbf{mut}\ S\langle'a\rangle)\ \{\\&\quad s_2.rx = \&\textbf{mut}\ *s_1.rx;\\&\}\end{aligned}$$

A caller of f_6 could look like this:

$$\begin{aligned}&\textbf{let mut}\ x = 0;\\&\textbf{let mut}\ s_1 = S\ \{rx : \&\textbf{mut}\ x\};\\&\textbf{let mut}\ s_2 = S\ \{rx : \&\textbf{mut}\ 0\};\\&f_6(\&\textbf{mut}\ s_1, \&\textbf{mut}\ s_2);\\&*s_2.rx = *s_2.rx + 1;\\&\text{assert!}(x = 1);\end{aligned}$$

Because $s_2.rx$ is an alias of x after the call to f_6, Rust restricts access to x while the alias is still in use.

2.2. Polonius

The Rust compiler checks the access rules for references described in Section 2.1 at compile time. The responsible module is called the *borrow checker*, implemented by an inference algorithm called *Polonius* [1]. This algorithm analyzes the borrows and lifetimes involved in a method and computes information that allows the compiler to verify the validity of reference usages. Prusti uses Polonius to implement various parts of its reference model. We will use this section to explore the core concepts of Polonius.

Polonius introduces the notion of loans, which indicate that some data is not only accessible through the owning variable, but also through an aliasing reference. They are created by borrow expressions, for example:

$$\textbf{let}\ y : \&\text{u32} = \&x.$$

The expression $\&x$ creates a loan (call it L1), which is subsequently stored in the y variable. After a loan is created at one program point, it is alive *alive* for some time until it becomes *dead* at another program point. The process of transitioning from the former to the latter state is called *expiration.* A loan is alive while it could be accessed later in the execution and dead otherwise. In the example, L1 is live while y could be dereferenced later.

Polonius tracks the liveness of loans via *regions*, which are an alternative interpretation of Rust's *lifetimes.* First, every reference type and borrow expression appearing in a function is assigned a unique region. The example would thus be changed to

$$\textbf{let } y : \&'r_1 \text{ u32} = \&'r_2\ x,$$

where the type of y is assigned the region r_1 and the borrow expression is assigned the region r_2. A region is interpreted as the set of loans that need to be alive to dereference the corresponding reference. For a borrow expression, this is straightforward – the assigned region contains the loan created by the borrow expression. The region r_2 from the example therefore contains the loan L1. Now take the assignment of the created reference to y of type $\&'r_1$ u32. Dereferencing y requires at least the loans from r_2 to be alive. Instead of listing all loans in r_1 explicitly, we just record the subset constraint $r_2 \subseteq r_1$. In Rust, this would be pronounced as r_2 outlives r_1. Branches can create multiple subset constraints for one region:

$$\textbf{let } y : \&'r_1 \text{ u32} = \textbf{if } c\ \{\ \&'r_2\ x_1\ \}\ \textbf{else}\ \{\ \&'r_3\ x_2\ \}$$

creates the constraints $r_2 \subseteq r_1$ and $r_3 \subseteq r_1$. Borrow expressions can also create subset constraints, if creating the borrow requires dereferencing other references. For example:

$$\begin{aligned}\textbf{let } y : \&'r_1 \text{ u32} = \&'r_2\ x;\\ \textbf{let } z : \&'r_3 \text{ u32} = \&'r_4\ {*y};\end{aligned}$$

The two assignments create the constraints $r_2 \subseteq r_1$ and $r_4 \subseteq r_3$. The borrow expression $\&'r_4\ {*y}$ creates another constraint $r_1 \subseteq r_4$, because creating the borrow with region r_4 requires dereferencing a reference with region r_1.

After collecting all subset constraints, Polonius can infer the concrete set values for all regions. With this information, the question whether a loan L is alive at a program point P is reduced to the question whether a region R that contains L is alive at P. How can we decide whether R is alive at P? Simply inspect all statements that could be executed after P and check whether a reference with region R is dereferenced. If this is the case, R is alive at P.

We can now explain the blocking behavior from Section 2.1 using regions and loans, instead of manually figuring out possible aliasing relationships. Recall that the following

program fails to compile, because x is blocked while the alias rx is still used. The types and regions have been made explicit:

$$\begin{aligned}&\textbf{let } x = 0;\\&\textbf{let } rx : \&'r_1 \textbf{ mut } \text{u32} = \&'r_2 \textbf{ mut } x;\\&x = x + 1;\\&{*rx} = {*rx} + 1;\end{aligned}$$

This creates one loan L1, through $\&'r_2$ **mut** x. For the lifetime of this loan, access to x is blocked. The reasoning is that the loan represents an alias for x, and as long as the loan is alive the alias may still be used. We collect all subset constraints and subsequently figure out that $r_2 = \{\text{L1}\}$ and $r_1 = \{\text{L1}\}$. The loan is alive until the very end of the program, which follows from two facts. First, rx, which has the type $\&'r_1$ **mut** u32, is dereferenced in the last line. Second, the region r_1 contains L1. It follows that r_1, and L1 by implication, are alive until after the last line. The compiler reports an error because x is read and written while L1 is alive.

We fixed the error by flipping the last two lines:

$$\begin{aligned}&\textbf{let } x = 0;\\&\textbf{let } rx : \&'r_1 \textbf{ mut } \text{u32} = \&'r_2 \textbf{ mut } x;\\&{*rx} = {*rx} + 1;\\&x = x + 1;\end{aligned}$$

Everything is still the same, except that L1 now expires before x is accessed. Because the loan is dead in the final line, x is not blocked anymore and accessing it does not produce an error.

Besides borrow expressions like $\&x$ and &**mut** x, certain other operations create loans as well. Moving or copying a reference from one place to another creates a loan.[3] For example, assuming y is a reference-typed variable, the statement

$$\textbf{let } z : \&\text{u32} = y$$

creates a new loan. The statement could be equivalently rewritten as

$$\textbf{let } z : \&\text{u32} = \&(*y),$$

which explains the created loan by making the borrow expression explicit. Function calls create loans for reference-typed argument and return places. For example, consider a function

$$\textbf{fn } f_7(y : \&\text{u32}) \rightarrow \&\text{u32}$$

[3] This is not how vanilla Polonius actually works. However, Prusti modifies Polonius to create loans also for moves and copies of references – this does not affect the correctness of Polonius' inference and simplifies the implementation of Prusti.

```
 1| fn f8<'p,'q>(
 2|     p : &'p mut Point,
 3|     q : &'q mut Point
 4| ) → (&'p mut u32, &'q mut u32) {...}

 6| fn f9() {
 7|     let mut p = Point {x : 0, y : 0};
 8|     let mut q = Point {x : 0, y : 0};
 9|     let rp1 = &mut p;                          (L0)
10|     let rq1 = &mut q;                          (L1)
11|     let rp2 = rp1;                             (L2)
12|     let (rp3, rq2) = f8(rp2, rq1);             (L3, L4, L5, L6)
13|     let rq3 = &mut *rq2;                       (L7)
14| }
```

Figure 2.1.: An example program to illustrate the information computed by Polonius. The loans created by each statement are written in parentheses on the right side.

and a call to this function:

$$\textbf{let } z : \&\text{u32} = f_7(y).$$

This creates one loan for the argument y (again explained by writing the argument as $\&(*y)$, making the borrow expression explicit) and another loan for the returned reference that is saved in z. This second loan is justified because even though we don't see it from the outside, the function must create the returned reference at some point with a borrow expression.

We now summarize the information that Polonius provides. First, we can obtain a list of all live loans at a program location P. Second, we can compute a list of all loans expiring immediately before the program location P, by taking the loans that were alive at a predecessor location but are not alive at P anymore. Third, we can ask for a list of loans kept alive by a reference-typed place at a program location P.

To better understand these queries, consider the example program from Fig. 2.1, which creates loans L0–L7. The loans L0, L1, and L7 are created by borrow expressions. The loan L2 is created by the move of a reference. The loans L3–L6 are created by the function call. The references *rp2* and *rq1* are moved into the call, creating loans L5 and L6, respectively. The call returns two references, which can borrow from the arguments.

Consequently, the loans L3 and L4 are created.

Consider liveness of loans first. The loan L0 is created in line 9 and is alive until after line 12. Even though *rp1* is not used again after line 11, the move into *rp2* keeps the loan alive while *rp2* is still alive. The loan L1 is created in line 10 and is alive until after line 13. Again, even though *rq1* is last used in line 12, the reference *rq2* could borrow from *rq1*, extending the lifetime of L1. As for the other loans, L2, L3, and L5 are alive until after line 12, and L4, L6, and L7 are alive until after line 13.

Information about the expiration of loans can be derived from the information about the liveness of loans. The loans L0, L2, L3, L5 expire after line 12, and the loans L1, L4, L6, L7 expire after line 13.

Finally, Polonius computes the loans a place is keeping alive at a given program location. For example, take the place *rp3* in line 12, which keeps the loans L0, L2, L3, L5 alive – meaning these loans cannot expire while *rp3* is still used. The loan L3 is kept alive by *rp3* because the corresponding reference is stored in this place. But L3 can borrow data from the first function argument, which is assigned the loan L5 – this means L5 is transitively kept alive by *rp3*. The same reasoning applies to L2 and L0 (L5 can borrow data from L2, and L2 can borrow data from L0).

2.3. Viper

Viper [11] is an intermediate verification language that includes tools for permission-based reasoning about heap state based on implicit dynamic frames [12]. Informally, *permissions* describe regions of the heap that they grant access to. Methods in Viper maintain a *context*, which contains the currently available permissions (or, more generally, *resources*, as we will see later). This context is updated as the execution proceeds. Two statements that modify the context are *inhale* and *exhale*. The following example illustrates their use:

$$
\begin{aligned}
&\textbf{var}\ x : \text{Ref} \\
&x.f := 1\ \ \text{// error: missing permissions for}\ x.f \\
&\textbf{inhale}\ \text{acc}(x.f) \\
&x.f := 1 \\
&\textbf{exhale}\ \text{acc}(x.f) \\
&x.f := 1\ \ \text{// error: missing permissions for}\ x.f
\end{aligned}
$$

We begin with a variable x, which is a reference. Viper's references can store data via *fields*, of which f is an example. Accessing the field f via the reference x is only allowed when also holding permissions for this field – we say the access is *framed* in this case. The

first assignment is lacking these permissions, which is why it fails. After adding them to the context with the inhale statement – $\mathrm{acc}(x.f)$ denotes the permission to access f via x – the second assignment succeeds. We can remove, or exhale, the permissions from the context again. Afterwards, the context does not contain the permission $\mathrm{acc}(x.f)$ anymore and the final assignment fails again.

Predicates bundle multiple permissions together. For example, the following predicate P contains two permissions, $\mathrm{acc}(x.f)$ and $\mathrm{acc}(x.g)$, for some reference x:

$$P(x : \mathrm{Ref}) = \mathrm{acc}(x.f) \wedge \mathrm{acc}(x.g)$$

The $\wedge$ symbol in this context is not a logical conjunction, but rather means that the permissions $\mathrm{acc}(x.f)$ and $\mathrm{acc}(x.g)$ contained in the predicate give access to separate memory regions. The context can not only contain permissions for fields, but also predicate instances (collectively known as *resources*). Having a predicate instance $P(x)$ in the context does not grant access to $x.f$ and $x.g$. The *unfold* statement instructs Viper to replace a predicate instance in the context with the permissions it contains. This means that after unfolding $P(x)$, accessing $x.f$ and $x.g$ is allowed. The inverse operation, which is called *fold*, replaces the permissions contained in a predicate instance with the predicate instance itself. The following example illustrates this:

var x : Ref
inhale $P(x)$
$x.f := 1$ // error: missing permissions for $x.f$
unfold $P(x)$
$x.f := 1$
fold $P(x)$
$x.f := 1$ // error: missing permissions for $x.f$

We begin with a variable x and add the predicate instance $P(x)$ to the context via the inhale statement. The predicate instance itself does not directly provide access to $x.f$, even though it contains the necessary permissions. After unfolding $P(x)$, the next assignment is possible. Once we fold $P(x)$, accessing $x.f$ is once more forbidden.

Viper has *old-expressions*, which allow the evaluation of expressions in past program states. Consider the following example:

var x : Ref
inhale $\mathrm{acc}(x.f)$
$x.f := 1$
label a
$x.f := 2$
assert $x.f = 2$
assert $\mathrm{old}[a](x.f) = 1$

This program defines a reference x and assigns the value 1 to $x.f$. It then gives the name a to the program state immediately after the assignment, by means of a label. It continues to overwrite $x.f$ with the value 2. The following assertion checks that $x.f$ is equal to 2, which is true because of the assignment immediately before it. The next assertion uses an old-expression to evaluate $x.f$ in an earlier program state, the one identified by the label a. Because the value of $x.f$ was 1 at this point, the assertion succeeds. An old-expression can omit the label, which evaluates the argument expression in the initial state of the containing method.

Viper supports methods that encapsulate code. An example:

$$
\begin{array}{l}
\textbf{method } f_{10}(x : \text{Ref}) \textbf{ returns } (y : \text{Ref}) \\
\quad \textbf{requires } P(x) \\
\quad \textbf{ensures } \text{acc}(y.f) \\
\{ \\
\quad \textbf{unfold } P(x) \\
\quad y := x \\
\}
\end{array}
$$

This defines a method f_{10} that takes a parameter x and returns the variable y, both references. The method has a precondition, specified using *requires*, which contains the resources that the caller is handing over to f_{10}, in this case $P(x)$. These resources make up the initial context of the method. The method also has a postcondition, specified using *ensures*, which contains the resources that f_{10} is handing over to the caller after the call, $\text{acc}(y.f)$ in this case. These resources must be contained in the final context of the method. The method body unfolds $P(x)$, which removes $P(x)$ from the context and adds $\text{acc}(x.f)$ and $\text{acc}(x.g)$ instead. After setting y and x equal, the postcondition is satisfied.

Imagine a caller of f_{10}:

$$
\begin{array}{l}
\textbf{var } x : \text{Ref} \\
\textbf{var } y : \text{Ref} \\
\textbf{inhale } \text{acc}(x.f) \\
\textbf{inhale } \text{acc}(x.g) \\
\textbf{fold } P(x) \\
y := f_{10}(x) \\
y.f := 1
\end{array}
$$

It creates references x and y, inhales permissions for $x.f$ and $x.g$, folds the predicate instance $P(x)$, and assigns the result of calling $f_{10}(x)$ to y. Calling f_{10} removes the predicate instance $P(x)$ from the context and adds permissions for $y.f$ to the context –

this follows from f_{10}'s pre- and postcondition. Notice that accessing $y.g$ is not possible after the call, because permissions for $y.g$ are not included in the postcondition of f_{10}. These permissions are lost.

The last feature of Viper that is important for this thesis are *magic wands*. We continue with the previous example and look closer at the predicate instance that is given to f_{10} due to its precondition. It is impossible to also include $P(x)$ in the postcondition, because the method body must unfold the predicate instance to obtain the permissions for $y.f$ required by the postcondition. However, the next best thing f_{10} can do is to make a promise in the postcondition: "If you give me the permissions for $y.f$ that you obtained from my postcondition, I will be able to reassemble the original predicate instance $P(x)$ that you handed over to me due to my precondition." Viper encodes such promises as magic wands, which are another kind of resource. The magic wand for this example is written as

$$\mathrm{acc}(y.f) \mathrel{-\!\!*} P(x),$$

which means that the permission $\mathrm{acc}(y.f)$ can be given up in exchange for $P(x)$. Before we discuss how f_{10} constructs this magic wand, we look at how a caller would use it. Assume the same setup as before (two variables x and y, the context contains $P(x)$) and consider the following call to f_{10}, which now includes the magic wand in its postcondition:

$$
\begin{aligned}
&1|\ \ y := f_{10}(x) \\
&2|\ \ y.f := 1 \\
&3|\ \ \textbf{apply}\ \mathrm{acc}(y.f) \mathrel{-\!\!*} P(x) \\
&4|\ \ \textbf{unfold}\ P(x) \\
&5|\ \ x.f := 2 \\
&6|\ \ x.g := 3
\end{aligned}
$$

After the call, the context contains $\mathrm{acc}(y.f)$ and the magic wand $\mathrm{acc}(y.f) \mathrel{-\!\!*} P(x)$ that we assumed to be included in the postcondition of f_{10}. Thus in line 2 we can access $y.f$, but not $x.f$ or $x.g$. After line 2 we don't need to access $y.f$ anymore, but we'd like to access $x.f$ and $x.g$ again. Thus we apply the magic wand, which removes the left-hand side – $\mathrm{acc}(y.f)$ – from the context and adds the right-hand side – $P(x)$. Consequently, we cannot access $y.f$ after line 3, but we can unfold $P(x)$ in line 4. This gives access to $x.f$ and $x.g$, as lines 5 and 6 demonstrate.

In order for a caller to be able to use the magic wand, f_{10} must include it in its postcondition. This is only possible with a prior proof that $\mathrm{acc}(y.f)$ really is the missing permission that makes it possible to reassemble $P(x)$. This proof is written in Viper as a *package* statement, shown in Fig. 2.2. The package statement contains statements that explain how to obtain the resources on the right-hand side of the magic wand under the assumption that the resources on the left-hand side are provided by the code that is applying the magic wand. It removes the resources that are required to obtain the

```
 1| method f10(x : Ref) returns (y : Ref)
 2|    requires P(x)
 3|    ensures acc(y.f)
 4|    ensures acc(y.f) −∗ P(x)
 5| {
 6|    unfold P(x)
 7|    y := x
 8|    package acc(y.f) −∗ P(x) {
 9|       fold P(x)
10|    }
11| }
```

Figure 2.2.: A version of f_{10} that has the magic wand $\mathrm{acc}(y.f) \mathrel{-\!\!*} P(x)$ in the postcondition. Viper requires a proof that the magic wand is justified, which is provided in the form of a package statement, lines 8–10.

right-hand side, unless they are included in the left-hand side. In this example, folding $P(x)$ requires $\mathrm{acc}(x.f)$ and $\mathrm{acc}(x.g)$. The package statement thus removes $\mathrm{acc}(x.g)$ from the context, but not $\mathrm{acc}(x.f)$. The latter is not removed because $x = y$ and the left-hand side includes $\mathrm{acc}(y.f)$. Besides removing $\mathrm{acc}(x.g)$, the package statement adds $\mathrm{acc}(y.f) \mathrel{-\!\!*} P(x)$ to the context. It follows that the final context contains $\mathrm{acc}(y.f)$ as well as the magic wand $\mathrm{acc}(y.f) \mathrel{-\!\!*} P(x)$, which are exactly the resources required by the postcondition.

Besides resources, magic wands can also include assertions about the memory these resources give access to. In the case of f_{10}, we notice that because the argument x points to the same memory as the returned y (recall that line 7 set them equal), modifying $y.f$ also modifies $x.f$. We can illustrate this further with another example:

```
y := f10(x)
y.f := 1
apply acc(y.f) −∗ P(x)
unfold P(x)
assert x.f = 1
```

Viper would reject this program, because it does not know about the relation between x and y. However, we are aware of it and know the final assertion is warranted. To

```
 1| method f_10(x : Ref) returns (y : Ref)
 2|    requires P(x)
 3|    ensures acc(y.f)
 4|    ensures acc(y.f) −∗ P(x) ∧ old[lhs](y.f) = x.f
 5| {
 6|    unfold P(x)
 7|    y := x
 8|    package acc(y.f) −∗ P(x) ∧ old[lhs](y.f) = x.f {
 9|       fold P(x)
10|    }
11| }
```

Figure 2.3.: The final version of f_{10} that extends the magic wand to include the relation between $x.f$ and $y.f$.

communicate this to Viper, we include the fact in the magic wand:[4]

$$\mathrm{acc}(y.f) \mathrel{-\!\!*} P(x) \land \mathrm{old}[\mathrm{lhs}](y.f) = x.f$$

The old-expression uses the lhs label, which is reserved for magic wands and identifies the program state immediately before applying the magic wand. Thus $\mathrm{old}[\mathrm{lhs}](y.f)$ evaluates to the value of $y.f$ immediately before the application and the right-hand side further claims that this value is equal to $x.f$. Modifying the magic wand in this way makes Viper accept the assertion $x.f = 1$. The final version of f_{10} with this modification is shown in Fig. 2.3.

2.4. Prusti

Rust's type system goes a long way towards verifying various memory safety properties about a program. But to statically check complex functional properties, further instruments are necessary. Prusti [8] is a tool that extends Rust with a way to provide functional specifications of functions, ie, descriptions of what a function does phrased in a formal logic, and also verify their validity. It works by encoding Rust programs and their specifications to Viper [11], an intermediate verification language, which ultimately proves their validity. Prusti already supports some of the most essential features of Rust, including certain aspects of the references and borrowing system.

[4]We omit the easily inferred *unfolding* of $P(x)$ normally required by Viper to aid readability.

2.4.1. Type Encoding

Prusti encodes Rust types as Viper predicates over references. Primitive types are encoded as permissions for a single field of Viper's corresponding primitive type. Rust's *u32* type, for example, is encoded as permissions for a single field v_{u32} of type *Int*:

$$\text{u32}(x : \text{Ref}) = \text{acc}(x.v_{\text{u32}}).$$

A struct with n elements is encoded as permissions for n fields $f_i, \ldots, f_n$ of type *Ref*, each corresponding to one struct field. The i-th field comes with permissions P_i corresponding to its encoded Rust type:

$$\text{struct}_n(x : \text{Ref}) = \bigwedge_{i=1}^{n} \text{acc}(x.f_i) \wedge P_i(x.f_i).$$

The encoding of tuple types matches the encoding of struct types.

An enum E with n variants $E_1, \ldots, E_n$ is encoded as permissions for $n + 1$ fields. One field, d, is of type *Int* and can assume values between 1 and n. The other fields, $v_1, \ldots, v_n$, are of type *Ref*. Depending on the discriminant, one of these fields comes with additional permissions. If $x.d$ has the value i, then $x.v_i$ comes with permissions P_i corresponding to the type of E_i interpreted as a struct:

$$\begin{aligned} \text{enum}_n(x : \text{Ref}) = \text{acc}(x.d) \wedge \text{acc}(x.v_1) \wedge \cdots \wedge \text{acc}(x.v_n) \wedge \\ \left(\bigwedge_{i=1}^{n} x.d = i \rightarrow P_i(x.v_i) \right). \end{aligned}$$

Finally, a mutable reference to a type T is encoded as permissions for a single field p of type *Ref* that comes with permissions corresponding to the encoding of T:

$$\text{refmut}_T(x : \text{Ref}) = \text{acc}(x.p) \wedge P_T(x.p).$$

A variable x of type T in Rust is translated to a variable x' of type *Ref* in Viper. At some program points, all memory implied by the type T is accessible through x. In this case, we own the predicate instance $P_T(x')$ in Viper. However, alive loans can restrict access to parts of x at other program points, which means the full predicate instance $P_T(x')$ is not available in Viper (because parts of $P_T(x')$ were used for the borrow that created the loan).

Every memory access in Rust is translated to a corresponding series of field accesses in Viper. The encoding must ensure that sufficient permission for these accesses are available at any point.

2.4.2. Functions and Function Calls

Prusti encodes every Rust function as one Viper function. For pre- and postconditions, it does not use the equivalent feature of Viper, but instead inhales preconditions at the beginning of the method and exhales postconditions at the end of the method. Consequently, a function call in Rust is not encoded as the corresponding function call in Viper, but instead as an exhale of the precondition followed by an inhale of the postcondition.

Permissions for the arguments and return values are handled likewise. A method starts with inhaling for every argument the predicate implied by its type and ends with exhaling the predicate implied by the return type. A function call exhales the predicate instances for the arguments and inhales the predicate instance for the return value.

In general, a function

$$
\begin{array}{l}
\#[\textbf{requires}(\text{pre})] \\
\#[\textbf{ensures}(\text{post})] \\
\textbf{fn}\ f_{11}(x_1 : T_1,\ \ldots,\ x_n : T_n) \to T_r\ \{\ldots\}
\end{array}
$$

is thus encoded as

$$
\begin{array}{l}
\textbf{method}\ f_{11}\ (x_1 : \text{Ref},\ \ldots,\ x_n : \text{Ref})\ \textbf{returns}\ (r : \text{Ref})\ \{ \\
\quad \textbf{inhale}\ \bigwedge_i P_{T_i}(x_i) \\
\quad \textbf{inhale}\ \text{pre} \\
\quad \ldots \\
\quad \textbf{exhale}\ \text{post} \\
\quad \textbf{exhale}\ P_{T_r}(r) \\
\}
\end{array}
$$

and a call $r = f_{11}(x_1, \ldots, x_n)$ to this function is encoded as

$$
\begin{array}{l}
\textbf{exhale}\ \text{pre} \\
\textbf{exhale}\ \bigwedge_i P_{T_i}(x_i) \\
\textbf{inhale}\ P_{T_r}(r) \\
\textbf{inhale}\ \text{post}
\end{array}
$$

2.4.3. Re-Borrowing Functions

Two aspects of the re-borrowing functions introduced in Section 2.1 require special attention in Prusti. First there are specifications – mutations through the returned reference are also reflected in the aliased variable. These relationships must be formulated in the specification of the function, and we see soon that classic pre- and postconditions are not sufficient for this. Second there is the permission management. The unblocking of variables referenced by function arguments upon the expiry of the reference returned by the function requires special steps in the encoding that justify the transfer of permissions from the expiring reference to the unblocked value.

Take specifications first and consider a caller of the re-borrowing function f_4 from before:

$$
\begin{aligned}
&1|\ \ \textbf{let mut}\ p = \text{Point}\ \{x : 0, y : 0\};\\
&2|\ \ \textbf{let}\ rp = \&\textbf{mut}\ p;\\
&3|\ \ \textbf{let}\ rx = f_4(rp);\\
&4|\ \ \ldots\\
&5|\ \ *rx = 1;\\
&6|\ \ \text{assert!}(p.x = 1);
\end{aligned}
$$

Recall that f_4 returns a reference to the x-coordinate of the point referenced by the argument, thus we expect to observe the effect of the assignment in line 5 also via $p.x$ in line 6. We cannot write this as an ordinary postcondition of f_4, for two reasons. First, the eventual value of $p.x$ is not known after the call returns. It is only determined when rx is assigned for the last time. Second, the encoding does not include permissions for p immediately after the call to f_4, which an assertion specifying the value of p would need to be framed.

Prusti solves this with another kind of postcondition, called a pledge, which describes what happens when the returned reference expires. In this example, the pledge would state that upon expiry of the output reference, the x-coordinate of the value referenced by the input reference has the same value as the final value of the output reference:

$$
\begin{aligned}
&\#[\textbf{after_expiry}(p.x = \text{before_expiry}(*\text{result})]\\
&\textbf{fn}\ f_4\langle 'a\rangle(p : \&'a\ \textbf{mut}\ \text{Point}) \to \&'a\ \textbf{mut}\ \text{u32}\ \{\\
&\quad \&\textbf{mut}\ p.x\\
&\}
\end{aligned}
$$

In general, pledges contain facts that become available after the expiration of the returned reference. They may use the before_expiry(...) environment to access the final value of the returned reference immediately before its expiration.

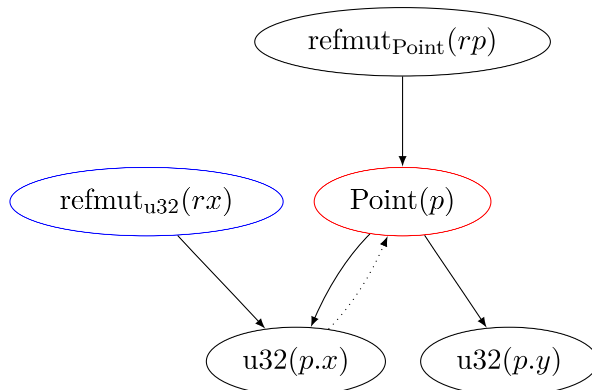


Figure 2.4.: The predicate instances after the function call. The solid arrows leaving one instance indicate the nested instances required to fold it. The instances for rx and p require the same u32-instance, explaining why they cannot be folded at the same time. Rust acknowledges this by blocking p while rx is still used. The faint arrow indicates the permissions the magic wand is converting between.

Now consider permission management. We saw in Section 2.1 how calls to re-borrowing functions can block places referenced by the input reference while the output reference is still alive. This blocking and unblocking of places in Rust is reflected in the Viper encoding by permission transfers. Section 2.4.2 explained how calling a (re-borrowing) function consumes permissions for the input reference and produces permissions for the re-borrowed output reference. Upon expiry of the output reference, permissions for it are given up in exchange for the original permissions consumed by the function call. The permission trade requires dedicated actions in the Viper encoding. We illustrate this with the call to the re-borrowing function f_4 from before. After line 1, we hold the predicate instance Point(p). After taking the mutable reference to p in line 2, this predicate instance is moved into the predicate instance $\text{refmut}_{\text{Point}}(rp)$, which encodes the mutable reference. Note that Rust blocks access to p while rp is alive, while Viper explains this necessity by the impossibility of owning the predicate instances Point(p) and $\text{refmut}_{\text{Point}}(rp)$ simultaneously. Then f_4 is called in line 3. Recall from f_4's signature that this requires us to give up permissions for the argument, ie, $\text{refmut}_{\text{Point}}(rp)$. The call returns permissions for its return value, the $\text{refmut}_{\text{u32}}(rx)$ predicate instance specifically. Note that this returned instance contains some of the permissions that we passed into the call. Specifically, the u32 predicate instance for the x-coordinate of the point is extracted and moved into the permissions for the mutable reference that is returned. Figure 2.4 visualizes the predicate instances and their relations after the call. Due to aliasing places (observable as multiple incoming edges in the visualization), not all instances can be folded.

At some point – after line 5 in the example – rx expires and Rust allows access to p again. The Viper encoding must mirror this change by restoring the Point(p) instance

and removing the $\text{refmut}_{\text{u32}}(rx)$ instance. This is a trade Viper does not readily accept – the veil of the function call hides the relation between $\text{refmut}_{\text{u32}}(rx)$ and $\text{Point}(p)$ that is apparent in Fig. 2.4 and would provide a justification. The object that can make Viper understand this relationship is a magic wand:

$$\text{u32}(rx.p) \mathrel{-\!\!*} \text{Point}(p)$$

provides the possibility of the permission transfer that should happen when rx expires in the Rust program. It is applied after line 5, which results in permissions that make the execution of line 6 possible.

The actual magic wand returned by the call is slightly different. First, it phrases the right-hand side in terms of rp, the reference moved into the call. Second, it uses labelled old-expressions to identify the exact references passed into or returned from the call. If the label before the call is *pre* and the label after the call is *post*, the magic wand is written as

$$\text{u32}(\text{old[post]}(rx.p)) \mathrel{-\!\!*} \text{Point}(\text{old[pre]}(rp.p)).$$

Viper does not allow clients to apply the magic wand without a prior proof that it is justified. This proof is found at the end of f_4, where a *package* statement explains in detail how it is possible to fold the Point(old[pre]$(rp.p)$) predicate instance given the returned u32(old[post]$(rx.p)$) predicate instance.

Pledges are embedded in the magic wands that already handle the permission transfer. For example, the pledge

$$\#[\textbf{after_expiry}(p.x = \text{before_expiry}(*\text{result})]$$

is embedded in the magic wand like this:

$$\begin{aligned}\text{u32}(\text{old[post]}(rx.p)) \mathrel{-\!\!*} {}& \text{Point}(\text{old[pre]}(rp.p)) \\ & \wedge \text{old[pre]}(rp.p).x.v_{\text{u32}} = \text{old[lhs]}(\text{old[post]}(rx.p).v_{\text{u32}}).\end{aligned}$$

The old[lhs](e) sub-expression evaluates e in the left-hand side state, ie the state immediately before applying the magic wand. Because the magic wand is applied when the returned reference expires, this is the state immediately before the expiration.

3. Re-Borrows of Simple References

Previously, we saw that re-borrowing functions pose an interesting problem for the Viper encoding of Rust programs. The issue is twofold. First there is permission management, implemented in ghost code in the Viper encoding, to synchronize the permissions in the encoded program with the accessible places in the original Rust program. To this end, the main tools are magic wands that exchange permissions for expired places with permissions for places that are unblocked by the expiration. Second there are specifications, which must allow users to capture the behavior arising from a function's use of re-borrows.

This chapter will extend Prusti's current support for re-borrowing functions, as introduced in Section 2.4.3, to support many input and output references in a single function. We begin with a formalization of Rust's rules for the blocking and unblocking of input references in Section 3.1. Section 3.2 introduces the necessary changes to the specification language that allow users to formulate properties about re-borrowing functions. The findings of the first section motivate subsequent design choices relating to the permission management, which is explained in Section 3.3. The magic wand stays fundamental, but significant machinery on top of it is made necessary by the problem that we tackle here. Section 3.4 explains how specifications are integrated with permission management, similarly to how they are embedded in magic wands in the current design. The sections introduced so far look at the problem from the angle of the callee, ie, the re-borrowing function itself. But clients of re-borrowing functions must also apply the permission management tools provided by the called function in a certain way as the output references expire. This is described in Section 3.5. As part of this thesis, support for re-borrowing functions involving multiple references has also been implemented in Prusti. Section 3.6 connects the method described in Sections 3.1 to 3.5 with the modules and functions of the implementation. Section 3.7 follows up with an evaluation, demonstrating example programs that can now be verified and taking a closer look at the performance of the implementation. Finally, Section 3.8 closes the chapter with possible improvements left for future work.

3.1. Re-Borrow Relationships

Functions cannot create re-borrows arbitrarily. Instead, the lifetimes in the function signature place a constraint on the input references a given output reference can be a re-borrow of. The *outlives* relation connects these lifetimes with each other. It is written as $'a : 'b$ to mean lifetime $'a$ outlives lifetime $'b$ – intuitively, a reference with lifetime $'a$ may be reinterpreted as a reference with lifetime $'b$, since $'a$ lives at least as long as $'b$. This relation is important for re-borrows since data from input references can only be re-borrowed into output references that live at most as long.

We define the outlives relation for a given function signature as the reflexive and transitive closure of the core relation specified in the function signature. For example, this signature has an empty core relation:

$$\mathbf{fn}\ f_{11}\langle 'a, 'b\rangle(a : \&'a\ \mathbf{mut}\ T,\ b : \&'b\ \mathbf{mut}\ T)\ \rightarrow\ (\&'a\ \mathbf{mut}\ T,\ \&'b\ \mathbf{mut}\ T)$$

Consequently, the outlives relation contains only the reflexive facts, ie, $'a : 'a$ and $'b : 'b$. This signature requires $'a : 'd$, $'b : 'c$, and $'c : 'd$ in the core relation:

$$\begin{aligned}
&\mathbf{fn}\ f_{12}\langle 'a : 'd, 'b : 'c, 'c : 'd, 'd\rangle(\\
&\quad a : \&'a\ \mathbf{mut}\ T, \\
&\quad b : \&'b\ \mathbf{mut}\ T \\
&)\ \rightarrow\ (\\
&\quad \&'c\ \mathbf{mut}\ T, \\
&\quad \&'d\ \mathbf{mut}\ T \\
&)
\end{aligned}$$

The outlives relation is therefore bigger. Besides the reflexive facts and base facts, it also requires that $'b : 'd$.

With the outlives relation, it is straightforward to determine the possible re-borrow relationships. An output reference with lifetime $'x$ can be a re-borrow of an input reference with lifetime $'y$ only if $'y$ outlives $'x$.

The re-borrow relationships directly give rise to the blocking behavior. After a client calls a re-borrowing function, an input reference is blocked while there is still a possible re-borrowing output reference alive. For a example, consider a prototypical call to f_{12}:

$$\mathbf{let}\ (c, d) = f_{12}(a, b)$$

The lifetimes tell us that c can be a re-borrow of a or b, while d can only be a re-borrow of a. Consequently, the input a is blocked in the caller while c or d are still alive, and the input b is blocked while c is still alive. A graphical representation of this information,

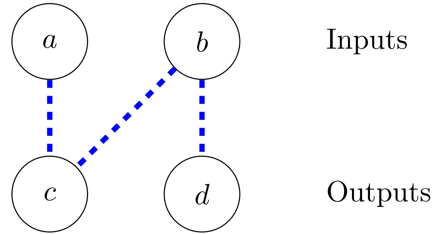


Figure 3.1.: The re-borrowing graph R for f_{12}. It has one node for every input reference at the top, and one node for every output reference at the bottom. Edges indicate that the output reference at the bottom can borrow from the input reference at the top.

which we call the *re-borrowing graph R for the function* f_{12}, is given in Fig. 3.1. This graph has one node for every input reference at the top, and one node for every output reference at the bottom. Edges indicate that the output reference at the bottom can borrow from the input reference at the top.

3.2. Pledge Syntax and Semantics

Prusti's syntax for pledges requires extensions to formulate properties about the more substantial re-borrowing functions. We illustrate this with an example:

$$
\begin{aligned}
&\textbf{fn}\ f_{13}\langle'p, 'q\rangle(\\
&\quad p : \&'p\ \textbf{mut}\ \text{Point},\\
&\quad q : \&'q\ \textbf{mut}\ \text{Point},\\
&)\ \rightarrow\ (\&'p\ \textbf{mut}\ \text{u32}, \&'q\ \textbf{mut}\ \text{u32})\ \{\\
&\quad \textbf{let}\ y = \text{rand}();\\
&\quad p.y = y;\\
&\quad q.y = y;\\
&\quad (\&\textbf{mut}\ p.x, \&\textbf{mut}\ q.x)\\
&\}
\end{aligned}
$$

We give the name px to the first item of the returned tuple, ie, result.0, and we give the name qx to the second item of the returned tuple, ie, result.0. This is both shorter to write and easier to read. Also assume that rand() returns some unknown integer.

To fully capture the behavior of this function, a specification needs to communicate three facts to the caller. First, changes to rx are visible in $p.x$ after p is unblocked. Specifically, the value of $*rx$ immediately before rx expires equals the value of $p.x$ immediately after p is unblocked. Second, the same holds for qx and $q.x$ – the value of $*qx$ immediately

before qx expires equals the value of $q.x$ immediately after q is unblocked. Third, $p.y$ and $q.y$ are equal. This part of the specification won't simply state that $p.y$ is equal to $q.y$ after the function call. Though this is true in some sense, it would cause problems later: recall that p and q are blocked until px and qx expire, respectively, and we cannot talk about $p.y$ and $q.y$ in the encoded program while p and q are still blocked (due to missing permissions). Instead, the specification will state that the value of $p.y$ after p is unblocked is equal to the value of $q.y$ after q is unblocked. In general, this relates the heap state at two different of the execution, as the following caller of f_{13} shows:

1| **let** $(px, qx) = f_{13}(\&\textbf{mut}\ p, \&\textbf{mut}\ q);$
2| $*px = 3;$
3| $p.y = p.y + 1;$
4| $*qx = 4;$
5| assert!$(p.y + 1 = q.y);$
6| assert!$(p.x = 3);$
7| assert!$(q.x = 4);$

Once p is unblocked after line 2, we increment $p.y$. When q is unblocked after line 4, the specification of f_{13} tells us that the value of $p.y$ after p was unblocked (which is the value of $p.y$ immediately after line 2) is equal to the value of $q.y$ after q was unblocked (which is the value of $q.y$ immediately after line 4). Since $p.y$ was incremented in the meantime, we should be able to prove that $p.y + 1 = q.y$ in line 5. The other two assertions in lines 6 and 7 also follow from f_{13}'s specification.

Some parts of the specification of a re-borrowing function are written using pledges. A pledge is a special kind of postcondition that can talk about input references that are blocked after the call and their relation to the output references that are blocking them. A function can have multiple pledges that talk about different subsets of the input and output references. A single pledge has the general form of

$$\#[\textbf{pledge}(\textit{body})]$$

where *body* relates some input references with each other and the state of re-borrows immediately before their expiry.

The specification of f_{13} consists of three pledges, one for each property we discussed above. We start by translating "the value of $*px$ immediately before px expires equals the value of $p.x$ immediately after p is unblocked" to a pledge. This is essentially an equality, we just have to figure out the left- and right-hand side. To write the left-hand side – "the value of $*px$ immediately before px expires" – we introduce a special environment before_expiry$[r_o](e)$. This environment takes an output reference r_o and an expression e as arguments and is defined to evaluate e immediately before r_o expires. The left-hand side of the equality is then written as

$$\text{before_expiry}[px](*px).$$

To write the right-hand side – "the value of $p.x$ immediately after p is unblocked" – we introduce another special environment called $\text{after_unblocked}[r_i](e)$. This environment takes an input reference r_i and an expression e as arguments and is defined to evaluate e immediately after r_i is unblocked. The right-hand side of the equality is then written as

$$\text{after_unblocked}[p](p.x).$$

To sum up, the first pledge looks like this:

$$\#[\textbf{pledge}(\text{before_expiry}[px](*px) = \text{after_unblocked}[p](p.x))].$$

The equivalent fact relating qx and q produces a very similar pledge:

$$\#[\textbf{pledge}(\text{before_expiry}[qx](*qx) = \text{after_unblocked}[q](q.x))].$$

Finally, one aspect of f_{13}'s behavior is still unspecified: "the value of $p.y$ after p is unblocked is equal to the value of $q.y$ after q is unblocked." We can write both sides of this equality using the after_unblocked environment, which produces this pledge:

$$\#[\textbf{pledge}(\text{after_unblocked}[p](p.y) = \text{after_unblocked}[q](q.y))].$$

We require that each occurrence of an after_unblocked or before_expiry environment contains exactly one input reference (in the first case) or output reference (in the second case). This allows alternative forms, where the square brackets together with their content are omitted:

$$\text{after_unblocked}(e_1), \text{before_expiry}(e_2).$$

3.3. Permission Management

Managing permissions at the Viper level is one central aspect of Prusti's encoding. We saw in Section 2.4.3 that re-borrowing functions provide a unique challenge in this context. The present implementation uses a single magic wand to allow clients to give up the permissions contained in the output reference and recover the permissions contained in the input reference in return. Now that multiple input and output references are involved, a single magic wand is no longer flexible enough. In the remainder of this section, we develop a tool that provides clients with a way to recover permissions for input references as the various output references expire one by one – the *expiration tool*. This is a Viper resource that allows the encoding to maintain the property that we have permissions in Viper for everything that is accessible in Rust.

We say an expiration tool for a function f *covers* a subset O of f's output references when it can be used to expire all output references in O. We write $\text{ET}'(I, O)$ for the

$$
\begin{aligned}
&\textbf{fn } f_{14}\langle 'a : 'd, 'b : 'd + 'e, 'c : 'e + 'f, 'd, 'e, 'f \rangle(\\
&\quad a : \&'a \textbf{ mut } \text{Point},\\
&\quad b : \&'b \textbf{ mut } \text{Point},\\
&\quad c : \&'c \textbf{ mut } \text{Point}\\
&) \to (\\
&\quad \&'d \textbf{ mut } \text{u32},\\
&\quad \&'e \textbf{ mut } \text{u32},\\
&\quad \&'f \textbf{ mut } \text{u32}\\
&);
\end{aligned}
$$

Figure 3.2.: The signature of a re-borrowing function. The $+$ on the right-hand side of an outlives constraint means that the lifetime on the left-hand side must outlive both lifetimes on either side of the $+$.

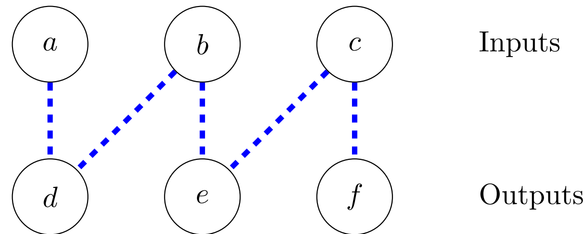


Figure 3.3.: The re-borrowing graph R for the function f_{14}.

expiration tool for f that covers the output references O, where I is the set of all input references that are still blocked by something in O. From a call to f, the caller receives an expiration tool that covers all of f's output references. Note that even though $\text{ET}'(I, O)$ depends on the function f that it is meant for, the notation does not make this explicit. The expiration tool $\text{ET}'(I, O)$ is refined later to produce an optimized expiration tool $\text{ET}(I, O)$ – hence the seemingly unmotivated prime.

We now consider the function f_{14} from Fig. 3.2 and describe how an expiration tool for this function would have to look. The function has three input references a, b, c and three output references, which we call d, e, f. The re-borrowing graph for f_{14}, derived from the lifetime constraints as described in Section 3.1, is displayed in Fig. 3.3. After a call to f_{14}, the caller owns the expiration tool $\text{ET}'(\{a, b, c\}, \{d, e, f\})$. This expiration tool must allow the caller to expire the output references one by one and return permissions for the input references as they're unblocked by the expirations. The expiration tool cannot make assumptions about the order in which d, e, f expire, because this is different from caller to caller.

We handle the unknown expiration order by simply accounting for every possibility. This means the expiration tool is just the conjunction of three *branches* B_d, B_e, and B_f that allow the expiration of d, e, and f, respectively:

$$\mathrm{ET}'(\{a,b,c\},\{d,e,f\}) := B_d \wedge B_e \wedge B_f.$$

All three branches are very similar, so we look at just one of them, B_d, in detail. If it is the case d is the first output reference to expire after the call, this branch should allow us to process the expiration of d. Otherwise, it should not give us anything. To check whether or not this is the case, we introduce a function $\text{expires_first}(x, ys)$, where x is an output reference and ys is a set of output references. It is defined to return true if x expires before any reference from ys. With this definition, $\text{expires_first}(d, \{e, f\})$ is true if d is the first output reference to expire after the call, and false otherwise. The branch B_d is then written as the following implication:

$$B_d := \text{expires_first}(d, \{e, f\}) \rightarrow W_d.$$

The right-hand side of the implication is the Viper resource that allows us to actually expire d. Before we look closer at the construction of this resource – the name W_d already hints it will be a magic wand – review the expiration tool $\mathrm{ET}'(\{a,b,c\},\{d,e,f\})$ now with the branches B_d, B_e, B_f filled in:

$$\begin{aligned}&(\text{expires_first}(d, \{e, f\}) \rightarrow W_d) \wedge \\ &(\text{expires_first}(e, \{d, f\}) \rightarrow W_e) \wedge \\ &(\text{expires_first}(f, \{d, e\}) \rightarrow W_f).\end{aligned}$$

Every branch first checks if the corresponding output reference is the first to expire, and if this is the case, it provides us with a resource that can expire this output reference. Note that the branches are mutually exclusive – if the condition of the first branch is true, the other two conditions must be false. It remains to define W_d, W_e, and W_f. Because they're all constructed similarly, we just consider W_d in the following. It must allow the caller to expire d, which amounts to exchanging permissions for the value referred to by d for permissions for all values that are unblocked by the expiration. To determine what is unblocked, we consult the re-borrowing graph from Fig. 3.3 and compare the input references that were blocked before the expiration – a, b, and c, in this case – with the input references that are still blocked after the expiration – b and c, in this case. This means a is unblocked after the expiration. Exchanging one set of permissions for another is achieved with a magic wand:

$$W_d := \text{u32}(\text{old[post]}(d.p)) \mathrel{-\!\!*} \text{Point}(\text{old[pre]}(a.p)).$$

Let's take this apart. The expression $\text{old[post]}(d.p)$ is the memory that the expiring output reference d points to after the call. The left-hand side of the magic wand thus contains permissions for the integer that the output reference d aliased. The expression $\text{old[pre]}(a.p)$ is the memory that the unblocked reference pointed to before the call.

Given the permissions for the integer from the left-hand side, the magic wand is able to reconstruct permissions for the point that was passed into the call via the reference a and that d may have aliased. The expiration tool $\mathrm{ET}'(\{a, b, c\}, \{d, e, f\})$ now looks like this:

$$
\begin{aligned}
&(\text{expires_first}(d, \{e, f\}) \rightarrow \text{u32}(\text{old[post]}(d.p)) -\!\!* \text{Point}(\text{old[pre]}(a.p))) \wedge \\
&(\text{expires_first}(e, \{d, f\}) \rightarrow \text{u32}(\text{old[post]}(e.p)) -\!\!* \top) \wedge \\
&(\text{expires_first}(f, \{d, e\}) \rightarrow \text{u32}(\text{old[post]}(f.p)) -\!\!* \top).
\end{aligned}
$$

The right-hand sides of the magic wands for the last two branches contain no permissions, because the expiration of e or f does not unblock any input references.

This expiration tool now allows callers to expire one output reference of their choosing after the call. But it is not done yet, because callers have no way to also expire a second output reference afterwards. To solve this, we revisit the right-hand sides of the magic wands W_d, W_e, W_f. Instead of including only permissions for the unblocked input references, we also include another expiration tool that enables the expiration of the remaining output references. The expiration tool $\mathrm{ET}'(\{a, b, c\}, \{d, e, f\})$ then looks as follows:

$$
\begin{aligned}
&\left(\begin{array}{l}\text{expires_first}(d, \{e, f\}) \rightarrow \\ \quad \text{u32}(\text{old[post]}(d.p)) -\!\!* \text{Point}(\text{old[pre]}(a.p)) \wedge \mathrm{ET}'(\{b, c\}, \{e, f\})\end{array}\right) \wedge \\
&\left(\begin{array}{l}\text{expires_first}(e, \{d, f\}) \rightarrow \\ \quad \text{u32}(\text{old[post]}(e.p)) -\!\!* \mathrm{ET}'(\{a, b, c\}, \{d, f\})\end{array}\right) \wedge \\
&\left(\begin{array}{l}\text{expires_first}(f, \{d, e\}) \rightarrow \\ \quad \text{u32}(\text{old[post]}(f.p)) -\!\!* \mathrm{ET}'(\{a, b, c\}, \{d, e\})\end{array}\right)
\end{aligned}
$$

After expiring d, the caller receives not only permissions for the unblocked point, but also the expiration tool $\mathrm{ET}'(\{b, c\}, \{e, f\})$, which covers e and f – exactly the output references that haven't expired yet. Note that the input reference a is removed from the first argument, because it was unblocked by the expiration. After expiring e, the caller receives the expiration tool $\mathrm{ET}'(\{a, b, c\}, \{d, f\})$, which again covers the output references that haven't expired yet after expiring e. The first argument still contains a, because it is still blocked in this setting.

After expanding the three nested expiration tools, we obtain the resource shown in Fig. 3.4. This figure introduces a shortcut notation. We often need to refer to the permissions for a value passed into a function call via a reference (written $\text{Point}(\text{old[pre]}(x.p))$ for an input reference x that points to a point) and the permissions for a value returned from a function call via a reference (written $\text{u32}(\text{old[post]}(y.p))$ for an output reference y that points to an integer). We will shorten these to $Q(x)$ and $Q(y)$, respectively. This allows us to omit both the exact Viper predicate that models the Rust type and the labelled old-expression that follows the reference in the pre- or post-state of the function

$$
\begin{array}{l}
(\\
\quad \text{expires_first}(d, \{e, f\}) \to Q(d) \mathrel{-\!\!*} Q(a) \land (\\
\quad\quad \text{expires_first}(e, \{f\}) \to Q(e) \mathrel{-\!\!*} Q(b) \land (\\
\quad\quad\quad \text{expires_first}(f, \{\}) \to Q(f) \mathrel{-\!\!*} Q(c)\\
\quad\quad)\\
\quad) \land (\\
\quad\quad \text{expires_first}(f, \{e\}) \to Q(f) \mathrel{-\!\!*} (\\
\quad\quad\quad \text{expires_first}(e, \{\}) \to Q(e) \mathrel{-\!\!*} Q(b) \land Q(c)\\
\quad\quad)\\
\quad)\\
) \land (\\
\quad \text{expires_first}(e, \{d, f\}) \to Q(e) \mathrel{-\!\!*} (\\
\quad\quad \text{expires_first}(d, \{f\}) \to Q(d) \mathrel{-\!\!*} Q(a) \land Q(b) \land (\\
\quad\quad\quad \text{expires_first}(f, \{\}) \to Q(f) \mathrel{-\!\!*} Q(c)\\
\quad\quad)\\
\quad) \land (\\
\quad\quad \text{expires_first}(f, \{d\}) \to Q(f) \mathrel{-\!\!*} Q(c) \land (\\
\quad\quad\quad \text{expires_first}(d, \{\}) \to Q(d) \mathrel{-\!\!*} Q(a)\\
\quad\quad)\\
\quad)\\
) \land (\\
\quad \text{expires_first}(f, \{d, e\}) \to Q(f) \mathrel{-\!\!*} (\\
\quad\quad \text{expires_first}(d, \{e\}) \to Q(d) \mathrel{-\!\!*} Q(a) \land (\\
\quad\quad\quad \text{expires_first}(e, \{\}) \to Q(e) \mathrel{-\!\!*} Q(b) \land Q(c)\\
\quad\quad)\\
\quad) \land (\\
\quad\quad \text{expires_first}(e, \{d\}) \to Q(e) \mathrel{-\!\!*} Q(c) \land (\\
\quad\quad\quad \text{expires_first}(d, \{\}) \to Q(d) \mathrel{-\!\!*} Q(a) \land Q(b)\\
\quad\quad)\\
\quad)\\
)
\end{array}
$$

Figure 3.4.: The expiration tool $\mathrm{ET}'(\{a, b, c\}, \{d, e, f\})$ generated for the function f_{14} from Fig. 3.2.

call. The ambiguity arising from using the same name Q for both variants is easily resolved by checking whether the argument is an input or output reference.

To better understand this expiration tool, consider a caller with the concrete expiry order d, e, f. This means these expressions are *true*:

$$\text{expires_first}(d, \{e, f\}), \text{expires_first}(e, \{f\}), \text{expires_first}(f, \{\}),$$

while these expressions (among others) are *false*:

$$\text{expires_first}(e, \{d, f\}), \text{expires_first}(f, \{d, e\}), \text{expires_first}(f, \{e\}).$$

Given this information, the expression from Fig. 3.4 can be vastly simplified to:

$$Q(d) \mathrel{-\!\!*} Q(a) \wedge (Q(e) \mathrel{-\!\!*} Q(b) \wedge (Q(f) \mathrel{-\!\!*} Q(c))).$$

Immediately after the call, the client has four resources at its disposal, namely the three lists referred to by the output references and the magic wand from above:

$$\{Q(d), Q(e), Q(f), Q(d) \mathrel{-\!\!*} Q(a) \wedge (Q(e) \mathrel{-\!\!*} Q(b) \wedge (Q(f) \mathrel{-\!\!*} Q(c)))\}.$$

Once the first output reference expires (which we know to be d), the client applies the magic wand and is granted access to $Q(a)$ again, plus another magic wand to continue from there. (Note that the inputs unblocked by the expiration of d depend on whether e already expired before d. In the current example the expiration of d does not grant access to $Q(b)$, because it is still blocked by e. If we had chosen the expiration order e, d, f, the expiration tool from Fig. 3.4 would have collapsed to a magic wand that grants access to both $Q(a)$ and $Q(b)$ upon expiry of d.) The client now owns these resources:

$$\{Q(a), Q(e), Q(f), Q(e) \mathrel{-\!\!*} Q(b) \wedge (Q(f) \mathrel{-\!\!*} Q(c))\}.$$

Then the second output reference expires, which we know is e. The client applies the magic wand it receives $Q(b)$, plus another magic wand to continue from there. After the application, it owns these resources:

$$\{Q(a), Q(b), Q(f), Q(f) \mathrel{-\!\!*} Q(c)\}.$$

Finally the last output reference expires. The client once again applies the magic wand it owns and is granted access to $Q(c)$, which is the last missing permission. It now owns exactly the resources it had before the call:

$$\{Q(a), Q(b), Q(c)\}.$$

The sequence of held permissions for the expiration order f, d, e would be

$$\begin{aligned}
\{Q(d), Q(e), Q(f), Q(f) \mathrel{-\!\!*} (Q(d) \mathrel{-\!\!*} Q(a) \wedge (Q(e) \mathrel{-\!\!*} Q(b) \wedge Q(c)))\} &\quad \text{(after the call)} \\
\{Q(d), Q(e), Q(d) \mathrel{-\!\!*} Q(a) \wedge (Q(e) \mathrel{-\!\!*} Q(b) \wedge Q(c))\} &\quad (f \text{ expired}) \\
\{Q(e), Q(a), Q(e) \mathrel{-\!\!*} Q(b) \wedge Q(c)\} &\quad (d \text{ expired}) \\
\{Q(a), Q(b), Q(c)\} &\quad (e \text{ expired})
\end{aligned}$$

We now give a description of the ET$'$ algorithm that constructs expiration tools for an arbitrary function f. The algorithm takes as inputs the input references I and output references O. It also has implicit access to the re-borrowing graph for f. In its most succinct form, the algorithm is written as follows:

$$\begin{aligned}
\mathrm{ET}'(I, O) &:= \bigwedge_{o \in O} \mathrm{expires_first}(o, O \setminus o) \rightarrow \mathrm{W}(I, O, o), \\
\mathrm{W}(I, O, o) &:= Q(o) \mathrel{-\!\!*} \left(\bigwedge_{i \in I'} Q(i) \right) \wedge \mathrm{ET}'(I \setminus I', O \setminus o), \\
I' &:= \mathrm{accessible}(I, O \setminus o),
\end{aligned}$$

where accessible$(I, O \setminus o)$ denotes the subset of I that is not blocked by any reference in $O \setminus o$, determined using the re-borrowing graph for f.

The resource produced by ET$'(I, O)$ includes magic wands to expire any output references from O. The magic wand to expire output $o \in O$, constructed by $\mathrm{W}(I, O, o)$, is conditioned on the expression expires_first$(o, O \setminus o)$ to make sure it is only available if o is actually the next reference to expire. (This is necessary because owning all of these magic wands simultaneously would allow a client to duplicate permissions.) The left-hand side includes the permission for the output reference that expires. The right-hand side includes permissions for input references immediately unblocked by the expiration, as well as the expiration tool that covers the remaining output references. This expiration tool is constructed by a recursive call to ET$'$, where the parameters reflect the changed situation after o expired. Only inputs still blocked by something in O are included, because we just returned permissions for the other inputs. The output o is excluded because it just expired.

There is an important optimization that reduces the size of the expiration tool. Recall that we have to nest magic wands via the recursive call to ET$'$ because the effect of an expiration depends on which other outputs expired before. In the example from Fig. 3.2, expiring d before e unblocks just a, while expiring d after e unblocks a and b. However, such interactions are only possible between outputs that appear in the same connected component of the re-borrowing graph. We can therefore consider every connected component in isolation, which reduces the depth of the recursion and consequently the size of the final expiration tool. The optimized algorithm ET works like this:

$$\mathrm{ET}(I, O) = \bigwedge_{i=1}^{n} \mathrm{ET}'(I_i, O_i),$$

where $(I_1, O_1), \ldots, (I_n, O_n)$ are the connected components of the re-borrowing graph. The original ET$'$ algorithm must be modified to recursively call ET instead of itself. The result of constructing $\mathrm{ET}(\{a, b, c\}, \{d, e, f\})$ for f_{14} is shown in Fig. 3.5

So far we've seen how to write the expiration tool that is passed to clients via the postcondition. But because magic wands are involved, Viper expects us to be explicit

$$
\begin{array}{l}
(\\
\quad \text{expires_first}(d, \{e, f\}) \to Q(d) -\!\!* \; Q(a) \wedge (\\
\quad\quad \text{expires_first}(e, \{f\}) \to Q(e) -\!\!* \; Q(b) \wedge (\\
\quad\quad\quad \text{expires_first}(f, \{\}) \to Q(f) -\!\!* \; Q(c)\\
\quad\quad)\\
\quad) \wedge (\\
\quad\quad \text{expires_first}(f, \{e\}) \to Q(f) -\!\!* \; (\\
\quad\quad\quad \text{expires_first}(e, \{\}) \to Q(e) -\!\!* \; Q(b) \wedge Q(c)\\
\quad\quad)\\
\quad)\\
) \wedge (\\
\quad \text{expires_first}(e, \{d, f\}) \to Q(e) -\!\!* \; (\\
\quad\quad \text{expires_first}(d, \{\}) \to Q(d) -\!\!* \; Q(a) \wedge Q(b)\\
\quad) \wedge (\\
\quad\quad \text{expires_first}(f, \{\}) \to Q(f) -\!\!* \; Q(c)\\
\quad)\\
) \wedge (\\
\quad \text{expires_first}(f, \{d, e\}) \to Q(f) -\!\!* \; (\\
\quad\quad \text{expires_first}(d, \{e\}) \to Q(d) -\!\!* \; Q(a) \wedge (\\
\quad\quad\quad \text{expires_first}(e, \{\}) \to Q(e) -\!\!* \; Q(b) \wedge Q(c)\\
\quad\quad)\\
\quad) \wedge (\\
\quad\quad \text{expires_first}(e, \{d\}) \to Q(e) -\!\!* \; Q(c) \wedge (\\
\quad\quad\quad \text{expires_first}(d, \{\}) \to Q(d) -\!\!* \; Q(a) \wedge Q(b)\\
\quad\quad)\\
\quad)\\
)
\end{array}
$$

Figure 3.5.: The expiration tool $\mathrm{ET}(\{a, b, c\}, \{d, e, f\})$ generated for the function f_{14} from Fig. 3.2.

```
macro CET(I, O) {
    for (I', O') in connected_components(I, O) {
        CET'(I', O')
    }
}

macro CET'(I, O) {
    for o in O
        "if expires_first(${o}, ${O \ o}) {"
            CW(I, O, o)
        "}"
    end
}

macro CW(I, O, o) {
    I' := accessible(I, O \ o)
    "package ${W(I, O, o)} {"
        "expire ${o}"
        for i in I'
            "fold Q(${i})"
        end
        CET(I \ I', O \ o)
    "}"
}
```

Figure 3.6.: The algorithm that generates Viper code to construct the expiration tool. Quotes indicate Viper code that is emitted by the algorithm. Instances of ${e} inside quotes are replaced by the evaluated form of e. The statement **expire** ${o} is meant to expand to the Viper statements that Prusti's encoding uses to model the expiration of o.

about how this resource is constructed from the permissions available at the end of the method, in the form of *package* statements. The algorithm to explain Viper how to construct the expiration tool, called CET, is given in Fig. 3.6. It has the same basic shape as the ET algorithm – every predicate emitted by ET is replaced with the corresponding *fold* operation in CET, every implication is replaced with the analogous *if* statement, and every magic wand instance is replaced with the *package* statement constructing this magic wand.

3.4. Specifications

The ET/CET algorithms in their current form provide clients with an expiration tool which they use to recover permissions for input references that are blocked immediately after the call. This section explains how these algorithms are extended with specifications, ie, with the functional properties provided by users via pledges. Essentially, there are two questions two answer. First, how are the expressions inside pledges encoded as Viper expressions? For logical connectives and function calls this does not deviate from how normal postconditions are handled, but before_expiry and after_unblocked environments are unique to pledges and require special treatment. Second, how are individual encoded pledges embedded in the expiration tool?

Let's consider the latter question first. A pledge p mentions input references $I(p)$ (within after_unblocked expressions) and output references $O(p)$ (within before_expiry expressions). The idea is to embed the encoded pledge at every position in the expiration tool where it is true for the first time that every input from $I(p)$ is unblocked and every output from $O(p)$ is expired (assuming time is measured by the magic wand nesting level). Consider the function f_{14} (which has the expiration tool from Fig. 3.5) and this pledge p:

$$\#[\textbf{pledge}(\text{before_expiry}(*e) = \text{after_unblocked}(b.x))]$$

To recap, this pledge states that $b.x$ after b is unblocked equals whatever the expression $*e$ evaluates to immediately before e's expiry. We have $O(p) = \{e\}$ and $I(p) = \{b\}$. Embedding this pledge into the expiration tool from Fig. 3.5 produces the expiration tool from Fig. 3.7, where (•) indicates the positions where the encoded pledge would be inserted. Experimenting with this expiration tool, we eventually see that any concrete expiration order causes it to collapse to an expression that contains exactly one (•).

$$
\begin{array}{l}
(\\
\quad \text{expires_first}(d, \{e, f\}) \to Q(d) \mathrel{-\!\!*} Q(a) \land (\\
\quad\quad \text{expires_first}(e, \{f\}) \to Q(e) \mathrel{-\!\!*} Q(b) \land (\bullet) \land (\\
\quad\quad\quad \text{expires_first}(f, \{\}) \to Q(f) \mathrel{-\!\!*} Q(c) \\
\quad\quad) \\
\quad) \land (\\
\quad\quad \text{expires_first}(f, \{e\}) \to Q(f) \mathrel{-\!\!*} (\\
\quad\quad\quad \text{expires_first}(e, \{\}) \to Q(e) \mathrel{-\!\!*} Q(b) \land Q(c) \land (\bullet) \\
\quad\quad) \\
\quad) \\
) \land (\\
\quad \text{expires_first}(e, \{d, f\}) \to Q(e) \mathrel{-\!\!*} (\\
\quad\quad \text{expires_first}(d, \{\}) \to Q(d) \mathrel{-\!\!*} Q(a) \land Q(b) \land (\bullet) \\
\quad) \land (\\
\quad\quad \text{expires_first}(f, \{\}) \to Q(f) \mathrel{-\!\!*} Q(c) \\
\quad) \\
) \land (\\
\quad \text{expires_first}(f, \{d, e\}) \to Q(f) \mathrel{-\!\!*} (\\
\quad\quad \text{expires_first}(d, \{e\}) \to Q(d) \mathrel{-\!\!*} Q(a) \land (\\
\quad\quad\quad \text{expires_first}(e, \{\}) \to Q(e) \mathrel{-\!\!*} Q(b) \land Q(c) \land (\bullet) \\
\quad\quad) \\
\quad) \land (\\
\quad\quad \text{expires_first}(e, \{d\}) \to Q(e) \mathrel{-\!\!*} Q(c) \land (\\
\quad\quad\quad \text{expires_first}(d, \{\}) \to Q(d) \mathrel{-\!\!*} Q(a) \land Q(b) \land (\bullet) \\
\quad\quad) \\
\quad) \\
)
\end{array}
$$

Figure 3.7.: The expiration tool from Fig. 3.5 with an embedded pledge, indicated by $(\bullet)$.

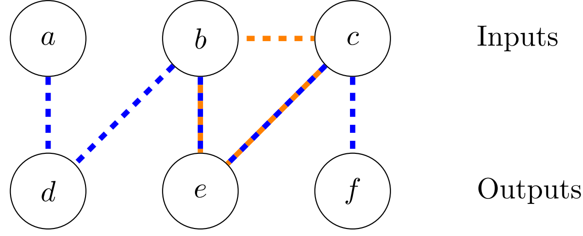


Figure 3.8.: Re-borrowing–with–pledges graph RP. Orange edges indicate references that are related via a pledge, while blue edges indicate references that are related via a re-borrowing relationship. The pairs $b - e$ and $c - e$ are related via both.

Now modify this pledge slightly to get p':

$$
\begin{aligned}
&\#[\textbf{pledge}(\\
&\quad (\text{before_expiry}(*e) = \text{after_unblocked}(b.x)) \lor \\
&\quad (\text{before_expiry}(*e) = \text{after_unblocked}(c.x)) \\
&)]
\end{aligned}
$$

For this pledge, $O(p') = \{e\}$ and $I(p') = \{b, c\}$. But embedding p' into the expiration tool from Fig. 3.5 as before reveals a problem, specifically for the expiration orders e, d, f and e, f, d. The simplified expiration tool (after evaluating the expires_first expressions) for both expiration orders looks like this:

$$Q(e) \mathrel{-\!\!*} \Big(Q(d) \mathrel{-\!\!*} Q(a) \land Q(b)\Big) \land \Big(Q(f) \mathrel{-\!\!*} Q(c)\Big).$$

Notice that this expression does not actually contain the pledge, since there is no point where $Q(b)$ and $Q(c)$ are simultaneously available, because these resources are managed by sibling magic wands. This happens because in an effort to reduce the nesting depth, the ET algorithm treats inputs/outputs in isolation if they are unrelated w.r.t. the possible re-borrowing relationships. But treating b and c in isolation from each other is exactly the wrong choice – they are actually related. Not via re-borrowing, but via a pledge. The solution is to generalize the ET algorithm's notion of relatedness by extending the re-borrowing graph from Fig. 3.3 with edges indicating relations introduced by pledges. This produces the re-borrowing–with–pledges graph RP, which is shown in Fig. 3.8 for the pledge p'. Using this generalized notion of relatedness, the expiration tool with embedded pledges takes on the form shown in Fig. 3.9. The simplified expiration tool for the expiration order e, d, f now contains the pledge:

$$Q(e) \mathrel{-\!\!*} \Big(Q(d) \mathrel{-\!\!*} Q(a) \land Q(b) \land \Big[Q(f) \mathrel{-\!\!*} Q(c) \land (\bullet)\Big]\Big).$$

Next we consider the encoding of pledges. We saw in Section 3.2 that pledges are mostly just logical expressions similar to the ones making up postconditions, and are encoded to

$$
\begin{array}{l}
(\\
\quad \text{expires_first}(d, \{e, f\}) \rightarrow Q(d) \mathrel{-\!\!*} Q(a) \wedge (\\
\quad\quad \text{expires_first}(e, \{f\}) \rightarrow Q(e) \mathrel{-\!\!*} Q(b) \wedge (\\
\quad\quad\quad \text{expires_first}(f, \{\}) \rightarrow Q(f) \mathrel{-\!\!*} Q(c) \wedge (\bullet)\\
\quad\quad)\\
\quad) \wedge (\\
\quad\quad \text{expires_first}(f, \{e\}) \rightarrow Q(f) \mathrel{-\!\!*} (\\
\quad\quad\quad \text{expires_first}(e, \{\}) \rightarrow Q(e) \mathrel{-\!\!*} Q(b) \wedge Q(c) \wedge (\bullet)\\
\quad\quad)\\
\quad)\\
) \wedge (\\
\quad \text{expires_first}(e, \{d, f\}) \rightarrow Q(e) \mathrel{-\!\!*} (\\
\quad\quad \text{expires_first}(d, \{f\}) \rightarrow Q(d) \mathrel{-\!\!*} Q(a) \wedge Q(b) \wedge (\\
\quad\quad\quad \text{expires_first}(f, \{\}) \rightarrow Q(f) \mathrel{-\!\!*} Q(c) \wedge (\bullet)\\
\quad\quad)\\
\quad) \wedge (\\
\quad\quad \text{expires_first}(f, \{d\}) \rightarrow Q(f) \mathrel{-\!\!*} Q(c) \wedge (\\
\quad\quad\quad \text{expires_first}(d, \{\}) \rightarrow Q(d) \mathrel{-\!\!*} Q(a) \wedge Q(b) \wedge (\bullet)\\
\quad\quad)\\
\quad)\\
) \wedge (\\
\quad \text{expires_first}(f, \{d, e\}) \rightarrow Q(f) \mathrel{-\!\!*} (\\
\quad\quad \text{expires_first}(d, \{e\}) \rightarrow Q(d) \mathrel{-\!\!*} Q(a) \wedge (\\
\quad\quad\quad \text{expires_first}(e, \{\}) \rightarrow Q(e) \mathrel{-\!\!*} Q(b) \wedge Q(c) \wedge (\bullet)\\
\quad\quad)\\
\quad) \wedge (\\
\quad\quad \text{expires_first}(e, \{d\}) \rightarrow Q(e) \mathrel{-\!\!*} Q(c) \wedge (\\
\quad\quad\quad \text{expires_first}(d, \{\}) \rightarrow Q(d) \mathrel{-\!\!*} Q(a) \wedge Q(b) \wedge (\bullet)\\
\quad\quad)\\
\quad)\\
)
\end{array}
$$

Figure 3.9.: The expiration tool with the generalized notion of relatedness and an embedded pledge, indicated by $(\bullet)$.

Viper in much the same way. Two elements of pledges require special thought, which are the before_expiry and after_unblocked expressions. The meaning of these is straightforward. For an expression e_1 that mentions an output reference r_o, before_expiry$[r_o](e_1)$ evaluates e_1 not in the current heap state, but in the heap state immediately before r_o expires. In a similar manner, for an expression e_2 that mentions an input reference r_i, after_unblocked$[r_i](e_2)$ evaluates e_2 in the heap state immediately after r_i is unblocked. These heap states correspond directly to certain positions in the expiration tool. The heap state immediately before an output reference r_o expires corresponds to all left-hand sides of magic wands that expire r_o. The heap state immediately after an input reference r_i is unblocked corresponds to all right-hand sides of magic wands that contain permission for r_i. This is justified because the expiration tool is constructed (and later used) such that obtaining permission for r_i as the result of a magic wand application is accompanied by the unblocking of r_i at the corresponding position in the Rust program, and Viper evaluates the right-hand side of a magic wand when it is applied.

The key to encoding before_expiry and after_unblocked is thus to be able to refer to the left-hand and right-hand sides of parent magic wands in a nested context. To understand better what this means, consider the following example:

$$P(x') \mathrel{-\!\!*_a} P(x) \land \Big(P(y') \mathrel{-\!\!*_b} P(y) \land \text{old[a:rhs]}(x.f) = y.f\Big).$$

First of all, the magic wands are named, there is one called a (denoted by $-\!\!*_a$) and another one called b (denoted by $-\!\!*_b$). Putting the permissions aside, the inner magic wand contains the assertion

$$\text{old[a:rhs]}(x.f) = y.f,$$

which is to be read as "the value of $x.f$, evaluated directly after applying the magic wand a, equals the value of $y.f$". If we imagine Viper code that is using this magic wand, we could observe something like this:

$$\begin{aligned}
&\textbf{apply } P(x') \mathrel{-\!\!*_a} P(x) \land (P(y') \mathrel{-\!\!*_b} P(y) \land \text{old}[a : \text{rhs}](x.f) = y.f);\\
&\textbf{label } a : \text{rhs};\\
&\ldots\\
&\textbf{apply } P(y') \mathrel{-\!\!*_b} P(y) \land \text{old}[a : \text{rhs}](x.f) = y.f;\\
&\textbf{assert } \text{old}[a : \text{rhs}](x.f) = y.f;
\end{aligned}$$

Or this example:

$$P(x') \mathrel{-\!\!*_a} P(x) \land \Big(P(y') \mathrel{-\!\!*_b} P(y) \land (\text{old[a:lhs]}(x'.g) = y.g \lor \text{old[b:lhs]}(y'.g) = y.g)\Big).$$

Here, the inner magic wand contains the assertion

$$\text{old[a:lhs]}(x'.g) = y.g \lor \text{old[b:lhs]}(y'.g) = y.g,$$

which is to be read as "$y.g$ equals either $x'.g$, evaluated immediately before applying the magic wand a, or $y'.g$, evaluated immediately before applying the magic wand b". This could lead to the following usage:

$$
\begin{array}{l}
\textbf{label}\ a : \text{lhs};\\
\textbf{apply}\ P(x') -\!\!*_a P(x) \wedge (\\
\quad P(y') -\!\!*_b P(y) \wedge (\text{old}[a : \text{lhs}](x'.g) = y.g \vee \text{old}[b : \text{lhs}](y'.g) = y.g)\\
);\\
\ldots\\
\textbf{label}\ b : \text{lhs};\\
\textbf{apply}\ P(y') -\!\!*_b P(y) \wedge (\text{old}[a : \text{lhs}](x'.g) = y.g \vee \text{old}[b : \text{lhs}](y'.g) = y.g);\\
\textbf{assert}\ \text{old}[a : \text{lhs}](x'.g) = y.g \vee \text{old}[b : \text{lhs}](y'.g) = y.g;
\end{array}
$$

Having $\text{old}[x : \text{lhs}](\ldots)$ and $\text{old}[x : \text{rhs}](\ldots)$ expressions, we can now encode pledges. To start with an example, take the pledge p' from before

$$
\begin{array}{l}
\#[\textbf{pledge}(\\
\quad (\text{before_expiry}(*e) = \text{after_unblocked}(b.x)) \vee\\
\quad (\text{before_expiry}(*e) = \text{after_unblocked}(c.x))\\
)]
\end{array}
$$

and consider a single position where the encoded pledge is to be inserted and simplify the expiration tool using the expiration order corresponding to that position, for example:

$$\text{Point}(e) -\!\!*_e \Big(\text{Point}(d) -\!\!*_d \text{Point}(a) \wedge \text{Point}(b) \wedge \Big[\text{Point}(f) -\!\!*_f \text{Point}(c) \wedge (\bullet)\Big]\Big).$$

The magic wands are named according to the output reference on the left-hand side. The sub-expression

$$\text{before_expiry}(*e)$$

should be evaluated immediately before e expires, which corresponds to the left-hand side of the magic wand e. It is therefore encoded as

$$\text{old[e:lhs]}(*e).$$

The sub-expression

$$\text{after_unblocked}(b.x)$$

should be evaluated immediately after b is unblocked, which happens when the magic wand d is applied. It is therefore encoded as

$$\text{old[d:rhs]}(b.x).$$

Finally, the sub-expression

$$\text{after_unblocked}(c.x)$$

should be evaluated immediately after c is unblocked, which happens when the magic wand f is applied. It is therefore encoded as

$$\text{old[f:rhs]}(c.x).$$

The complete encoded pledge that (•) will be replaced by therefore looks like this:

$$\Big[\text{old[e:lhs]}(*e) = \text{old[d:rhs]}(b.x)\Big] \vee \Big[\text{old[e:lhs]}(*e) = \text{old[f:rhs]}(c.x)\Big].$$

To make this procedure formal, we begin with a function $EP_{WC}(P)$ that takes a pledge P and turns it into a Viper expression, under a context WC. The context contains information about the names of magic wands that expire outputs and unblock inputs. For an input reference r_i, $WC(r_i)$ gives the name of the magic wand that unblocks r_i and for an output reference r_o, $WC(r_o)$ gives the name of the magic wand that expires r_o. The definition of $EP_{WC}(P)$ then looks as follows:

$$\begin{aligned}
\text{EP}_{\text{WC}}(\text{before_expiry(e)}) &= \text{old}[\text{WC}(r) : \text{lhs}](\text{EP}_{\text{WC}}(e)),\\
\text{EP}_{\text{WC}}(\text{after_unblocked(e)}) &= \text{old}[\text{WC}(r) : \text{rhs}](\text{EP}_{\text{WC}}(e)),\\
\text{EP}_{\text{WC}}(e_1 \wedge e_2) &= \text{EP}_{\text{WC}}(e_1) \wedge \text{EP}_{\text{WC}}(e_2),\\
&\dots
\end{aligned}$$

Next, EP_{WC} must be integrated with the ET/CET algorithms. The adjusted definition of the ET algorithm is as follows:

$$\begin{aligned}
\text{ET}_{\text{WC}}(I, O, P) &:= \bigwedge_{i=1}^{n} \text{ET}'_{\text{WC}}(I_i, O_i, P),\\
\text{ET}'_{\text{WC}}(I, O, P) &:= \bigwedge_{o \in O} \text{expires_first}(o, O \setminus o) \to W_{\text{WC}}(I, O, P, o),\\
W_{\text{WC}}(I, O, P, o) &:= Q(o) -\!\!*_l \left(\bigwedge_{i \in I'} Q(i)\right) \wedge EP_{\text{WC}'}(P') \wedge ET_{\text{WC}'}(I \setminus I', O \setminus o, P \setminus P'),\\
I' &:= \text{accessible}(I, O \setminus o),\\
P' &:= \{p \in P : ((I \setminus I') \cup (O \setminus o)) \cap (I(p) \cup O(o)) = \emptyset\},\\
\text{WC}' &:= \text{WC}[o \to l, i \in I' \to l].
\end{aligned}$$

This is quite a bit to digest. The ET algorithm now receives two additional parameters. First there is P, which is the set of all pledges that still need to be embedded in the expiration tool – once a pledge is embedded, the P parameter of subsequent recursive calls doesn't contain it anymore. Then there is the magic wand context WC. Whenever a new magic wand is created, WC', an updated version of WC, is created such that the output o on the left-hand side of the magic wand and the inputs I' on the right-hand side of the magic wands are mapped to l, the fresh label of the newly created magic

```
1| fn f15(rx : &mut u32) → &mut u32 {&mut *rx}
2| fn f16() {
3|     let mut x = 0;
4|     let a = if cond() { f15(&mut x) } else { &mut 1 };
5|     // a expires.
6| }
```

Figure 3.10.: A re-borrowing function is invoked inside a branch.

wand. This update reflects the intention behind WC, since the output o and inputs I' are expired/unblocked by the magic wand l. The two new parameters WC and P are finally used to embed pledges in the right-hand side of magic wands, via the call $EP_{WC'}(P')$. The argument P' is the set of pledges in P that are ready. A pledge p is ready if the inputs and outputs that it mentions, formally $I(p) \cup O(p)$, overlap neither with the inputs that haven't been unblocked yet, formally $I \setminus I'$, nor the outputs that haven't expired yet, formally $O \setminus o$. Formulated equivalently, a pledge is ready if all inputs and outputs it mentions have been unblocked resp. expired.

3.5. Clients

Finally, the question remains how clients can make use of the expiration tools returned by the re-borrowing functions they call. Before the method is described in detail in Section 3.5.1 and Section 3.5.2, we begin with two examples that illustrate the involved challenges. This section often talks about loans created by function calls and their expiration, instead of references returned from function calls and their expiration. Recall from Section 2.2 that there is a one-to-one correspondence between loans created by and references returned from function calls, which means both points of view are equally valid and easily translated into each other.

One difficulty is posed by branches, as illustrated in Fig. 3.10. The function f_{15} takes a mutable reference and returns a re-borrow of it. The function f_{16} assigns either the result of a call to f_{15} or a reference to a literal 1 to a variable a, depending on the expression cond(). In line 5, a expires and unblocks x. The appropriate action on the Viper level in response to this expiration depends on which arm of the earlier branch was taken. In one case, line 5 is encoded by applying the expiration tool returned by the call to f_{15}. In the other case, no further action is needed.

Another example, shown in Fig. 3.11, calls a function that takes two references rx, ry

```
 1| fn f_17<'x, 'y : 'x>(
 2|     rx : &'x mut u32,
 3|     ry : &'y mut u32
 4| ) → (&'x mut u32, &'y mut u32) {(&mut *rx, &mut *ry)}

 5| fn f_18() {
 6|     let mut x = 0;
 7|     let mut y = 0;
 8|     let rx = &mut x;
 9|     let ry = &mut y;
10|     let (a, b) = f_17(rx, ry);
11|     *a = 1;
12|     // a expires
13|     *b = 2;
14|     // b expires
15| }
```

Figure 3.11.: A re-borrowing function is called.

as arguments and returns two re-borrows. The lifetimes tell us that the first output reference can be a re-borrow of both rx and ry, while the second output reference is a re-borrow of ry. After the call to f_{17} we hold some expiration tool that allows us to recover permissions for rx and ry in exchange for permissions for a and b. If we fix the expiration order a, b, this expiration tool looks as follows:

$$Q(a) \mathrel{-\!\!*} Q(rx) \wedge (Q(b) \mathrel{-\!\!*} Q(ry)).$$

To expire a in line 12, we apply the top-level magic wand from the expiration tool that has permissions for a on the left-hand side. Recall that the right-hand side of this magic wand will contain multiple resources. First, there are permissions for rx, since this input parameter is unblocked by the expiration. Second, there is the expiration tool that we can use to expire the next reference, ie, b:

$$Q(b) \mathrel{-\!\!*} Q(ry).$$

To expire b, we don't use the expiration tool returned by the call to f_{17}, since this has been consumed by the expiration of a. Instead, we take the expiration tool that we obtained from the expiration of a, and apply the top-level magic wand from this expiration tool that has permissions for b on the left-hand side. The right-hand side of this magic wand will contain permissions for ry. Because b is the last output reference to expire, there will be no further nested expiration tool. This example illustrates the need to keep track of the expiration tool that we currently hold. Because a expired before b, the expiration tool we can use to expire b is not the whole expiration tool we got from the call to f_{17}, but instead whatever is left over after expiring a.

Superficially, the expiration of loans at a given program location is split into two parts. Section 3.5.1 describes a function that generates statements used to expire a single loan. This function generates different statements depending on the kind of loan it is given. Important for us is the case where the loan was created by a call to a re-borrowing function, in which case the statements will include the application of the appropriate magic wand that has permissions for the reference corresponding to the expired loan on the left-hand side. Section 3.5.2 describes a function that takes the statements generated in the first step and assembles them into a CFG that orchestrates their execution. This is not entirely trivial for two reasons. One is that loans can be conditional, as we saw before in Fig. 3.10. This necessitates measures that make sure the statements to expire a loan are only executed if this loan has actually been created. The second reason is that loans can have dependencies between each other, in the sense that the statements to expire one loan must come after the statements to expire another loan. These dependencies must be reflected in the generated expiration CFG.

```
1| let (x, y) = f(...);
2| if cond() {
3|     *x = 1
4|     *y = 2
5| } else {
6|     *y = 1
7|     *x = 2
8| };
```

Figure 3.12.: The order in which the references x, y returned by a re-borrowing function expire is different in the two arms of the if-statement. In one arm, x expires after line 3 followed by y expiring after line 4. In the other arm, the expiration order is different. There, y expires after line 6 followed by x expiring after line 7.

3.5.1. Generating the Expiration Statements

We begin by looking at the algorithm that generates the statements to expire a single loan L. Besides L, it receives the program location P (in the form of basic block index and statement index) at which the loan expires. This is necessary because the same loan can expire at multiple different (mutually exclusive) program locations, and the generated statements depend on which loans have already expired before. As Fig. 3.12 shows, this depends on the program location.

First of all, we determine the call C that created L. Given C, we determine the set of all loans created by C, called CL. Note that $L \in \text{CL}$, since L was also created by C. Knowing CL is important, because these loans are managed by the same expiration tool as L. We partition CL into two subsets, the set of loans that expired prior to P, called CLE, and the set of loans that will expire later, called $\overline{\text{CLE}}$. This step requires knowing P, because the loans that have already expired can be different at different locations in the program. Note that CLE does not contain L, while $\overline{\text{CLE}}$ does.

Next, we retrieve the initial expiration tool E for C. This expiration tool makes it possible to expire all the loans from CL and obtain the blocked inputs in return. But we have to be careful, because after using E to expire one loan from CL, we cannot use E again to expire the next loan from CL. Instead, the first expiration dismantles E and leaves behind a smaller expiration tool, E', which we must use for the next expiration. This process is described in detail on page 53.

Because all the loans from CLE already expired before L, the expiration tool EL we can use to expire L is whatever is left over of E after expiring the loans from CLE. This expiration tool is obtained as follows. We start with E and perform one dismantling step for every loan from CLE. The input for one step is the output of the previous step. The output of the last step is the expiration tool EL. Note that the order in which the loans from CLE are used to dismantle E is irrelevant. The expiration tool is constructed such that any order yields the same result. This is important, because in general there is no single correct order in which the loans from CLE expired. Instead, as we saw in Fig. 3.12, every path from the entry point to the expiration point can expire the loans from CLE in a different order.

Having the right expiration tool EL to expire L, we can use it to perform the actual expiration. The first step is to identify the branch B_L of EL responsible for L. Recall that this branch has the following shape, where P contains the pledges made available by the expiration and the permissions for input references that are unblocked by the expiration, and EL' is the expiration tool that must be used for the next expiration after L:

$$B_L = \text{expires_first}(L, \overline{\mathrm{CLE}}) \to Q(L) \mathrel{-\!\!*} P \wedge \mathrm{EL}'.$$

Note that this branch is a resource we own in Viper (ie, asserting B_L at this point will succeed). The right-hand side of the implication contains the magic wand to expire L. The left-hand side is the branch condition that must be true before we can use the magic wand. Since Polonius assures us that L really is the next loan to expire out of $\overline{\mathrm{CLE}}$, the branch condition would be true according to the definition of expires_first. However, Viper does not know this because the semantics of expires_first are not written down as Viper axioms.[1] Instead, we just assume the branch condition:

$$\textbf{assume } \text{expires_first}(L, \overline{\mathrm{CLE}})$$

and apply the magic wand:

$$\textbf{apply } Q(L) \mathrel{-\!\!*} P \wedge \mathrm{EL}'.$$

After the application, permissions for the unblocked references and the pledges that are made available by this expiration are accessible through P.

Finally, there is one more thing to take care of. The pledges in the expiration tool EL' can refer to the left- and right-hand side of the magic wand we just applied, via $\mathrm{old}[L : lhs](\dots)$ and $\mathrm{old}[L : rhs](\dots)$ expressions. But because the next expiration will use EL' as a top-level expiration tool (instead of it being nested under the magic wand), these references will be dangling. To keep them available, the expressions in these

[1] We could enhance Viper's understanding of loans by appropriately axiomatizing expires_first and inserting information about which loans expire where. For example, a boolean flag L_{expired} for every loan L that is false initially and set to true as soon as L expires would already provide enough information. A definition of expires_first could then state that $\text{expires_first}(L, \overline{LE})$ is true if and only if there is a single loan $L' \in \overline{LE}$ such that L'_{expired} is true and $L = L'$.

dangling references must be evaluated and saved to local variables now. Later usages of EL′ will then replace every old$[L : lhs](\dots)$ and old$[L : rhs](\dots)$ expression with the name of the corresponding local variable, thereby eliminating dangling references.

Dismantling the Expiration Tool

This process works as follows. We start with an expiration tool E_i (which may or may not be the initial expiration tool). Our goal is to construct the expiration tool E_{i+1} reflecting the state after we used E_i to expire a loan L_i. (The indices are meant to indicate the expiration we're doing. We first expire L_1 using E_1, which leaves us with E_2, and so on.) Recall that E_i is just a collection of partial expiration tools $E_{i,1}, \dots, E_{i,n}$ for the n connected components of the re-borrowing graph. One of these partial expiration tools, call it $E_{i,j}$, allows us to expire L_i:

$$E_i = E_{i,1} \wedge \cdots \wedge E_{i,j} \wedge \cdots \wedge E_{i,n}.$$

Further recall that a partial expiration tool is just a collection of branches, one for every loan covered by it. We can dig into $E_{i,j}$ to identify the branch B_{L_i} responsible for L_i:

$$E_{i,j} = \cdots \wedge B_{L_i} \wedge \dots.$$

A branch is simply an implication with the expires_first condition on the left-hand side and a magic wand on the right-hand side. This magic wand will contain permissions for the expired place on the left-hand side and permissions for the unblocked places, pledges, and a nested expiration tool $E'_{i,j}$ on the right-hand side:

$$B_{L_i} = \text{expires_first}(L_i, \overline{\text{CLE}}) \rightarrow Q(L_i) \mathrel{-\!\!*} P \wedge E'_{i,j}.$$

The expiration tool E_{i+1} is constructed by taking E_i and replacing $E_{i,j}$ with $E'_{i,j}$:

$$E_{i+1} = E_{i,1} \wedge \cdots \wedge E_{i,j-1} \wedge E'_{i,j} \wedge E_{i,j+1} \wedge \cdots \wedge E_{i,n}.$$

A brief justification of why this is the right thing to do is that to expire L_i, we apply the magic wand from B_{L_i}, which gives us $E'_{i,j}$. Since the branches of $E_{i,j}$ are mutually exclusive, choosing B_{L_i} voids all the other branches. This means the only part of $E_{i,j}$ that is left after the expiration is $E'_{i,j}$.

3.5.2. Assembling the Expiration CFG

Every program location can have multiple loans expiring simultaneously. Because Prusti tracks the permissions held for every place and loans transfer permissions from one place to another (the expiration undoes this transfer), the expiration of simultaneously expiring

loans must follow some order. For example, if loan $L1$ moves permissions from A to B and loan $L2$ moves permissions from B to C, $L2$ must be expired before $L1$. Otherwise the expiration of $L1$ notices that the permissions it wants to move from B to A are missing. In other words, the loans can have dependencies on each other. This applies to all loans created in a program, not just loans corresponding to references returned from function calls. But for the latter case, the requirement is not just due to Prusti's permission tracking. Consider a program

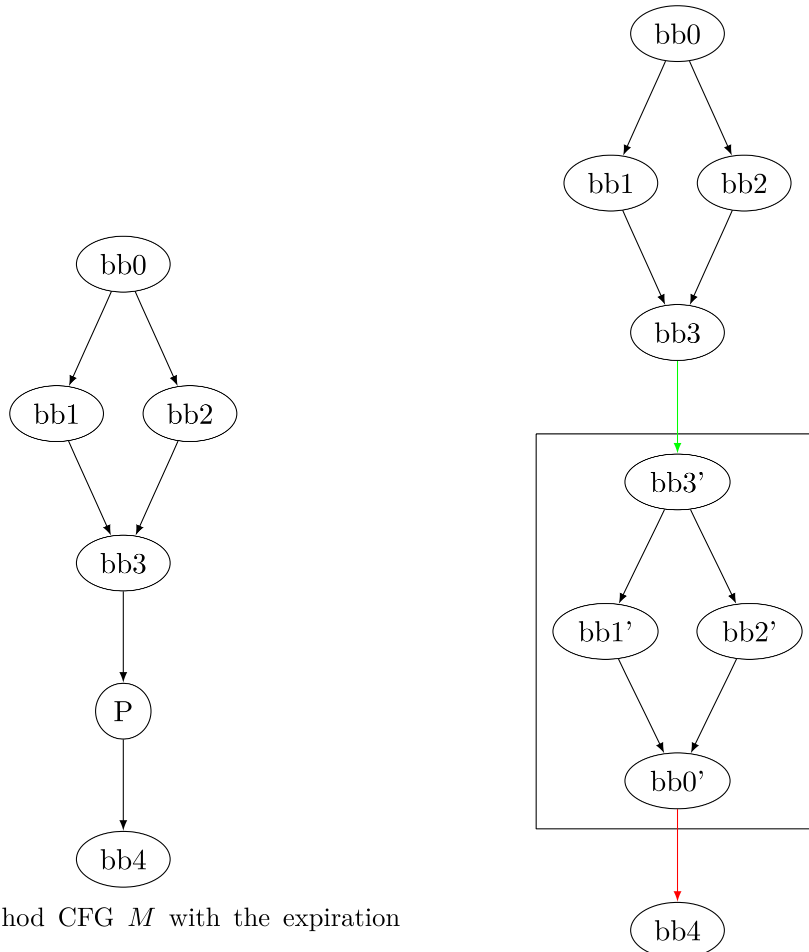
$$\text{let } B \stackrel{L1}{=} f(A);\ \text{ let } C \stackrel{L2}{=} f(B),$$

where A, B, C are mutable references. The loans $L1$ and $L2$ require a magic wand application to instruct Viper to move the permissions from B to A and from C to B. If $L1$ and $L2$ expire at the same time and we try to order the expiration of $L1$ before the expiration of $L2$, Viper will produce an error that the magic wand for $L1$ cannot be applied, because the permissions for the left-hand side are missing. We obtain these permissions by applying the magic wand for $L2$ first.

Besides the order in which expirations must be performed, the expiration CFG also ensures that loan conditions are honored. The example from Fig. 3.10 showed why this is necessary. There are two loans, one for every branch of the if statement. When one of these loans expires, the expiration statements should only be executed if the branch in which the loan was created was taken earlier in the execution.

The general idea behind the expiration CFG is to undo every statement that led to the expiration point P in order, starting with the most recently executed statement and finishing with the first statement of the method. Intuitively this works by taking the method CFG M, flipping all the edges, and reversing the statement order within the basic blocks. Then every statement is either deleted (if it does not create a loan) or replaced with the expiration statements for the loan it created. The result M' is embedded into M with two edges. The first edge connects the predecessor of P with the point corresponding to the predecessor of P in M'. The second edge connects the point in M' that corresponds to M's entry point with the successors of P. One example of this process is shown in Fig. 3.13.

With the structure of the expiration CFG fixed, it remains to equip the edges with the appropriate conditions. First consider a method without loops. Because the goal is to retrace the steps leading to P in reverse order, every edge of the expiration CFG should be followed if the corresponding edge of the method CFG was taken to reach P. To achieve this, we maintain a boolean variable $\text{visited}(e)$ for every edge e of the method CFG that indicates whether the edge was taken during the current execution. Then we say an edge e' of the expiration CFG should be followed if the $\text{visited}(e)$ variable for the corresponding edge e of the method CFG is true. What changes when we allow loops? This is left as an exercise for the reader.



(a) The method CFG M with the expiration point P.

(b) The expiration CFG M' (within the box) embedded into the method CFG M. The predecessor of P in M is connected to its corresponding point in M' (green edge). The point corresponding to the entry point of M is connected to the successor of P (red edge).

Figure 3.13.: The method CFG M before and after embedding the expiration CFG M'.

3.6. Implementation

We start with a general overview of the implementation and then revisit the individual components in more detail in Sections 3.6.1 to 3.6.6. Afterwards, Section 3.6.7 highlights some notable differences between the method that has been described so far and what is actually implemented at the moment. The following outlines the relevant steps during the verification of a method M.

Like pre- and postconditions, pledges are added to M via a procedural macro called *pledge*, described in more detail in Section 3.6.1. The entry point of this macro is the method of the same name in the main module of the *prusti-contracts-internal* sub-project. During the verification of M, these pledges will be accessible through the *ProcedureSpecification* struct as a vector containing instances of the *Pledge* struct. The *rhs* field of the *Pledge* struct wraps the assertion that is the pledge. This assertion is represented as an *Assertion* struct, just like regular pre- and postconditions.

Soon after beginning to verify M, its contract is constructed in the *encoder::borrows::compute_procedure_contract* method of the *prusti-viper* sub-project. The contract contains information pertaining to M's signature. One important part of the contract is the re-borrowing graph R from Section 3.3, represented by the *InterfaceReborrowingGraph* struct. This data structure is described in more detail in Section 3.6.2.

As the verification of M proceeds, three methods are relevant in particular:

encode_postcondition_expiration_tool constructs the expiration tool for a method as a Viper expression. It is called to construct the expiration tool for M itself (if M is a re-borrowing function) and also to construct the expiration tool for other methods that M is calling.

encode_package_end_of_method generates the statements that package M's expiration tool. It is called once during the verification of M and the generated statements are inserted close to the end of the encoded method.

construct_vir_reborrowing_node_for_call generates the statements that expire a loan created by a function call within M. It is called once for every expiration of a call loan. This is described in more detail in Section 3.6.6.

All three methods rely heavily on one central data structure, the *ExpirationTool* struct, located in the *encoder::expiration_tool* module of the *prusti-viper* sub-project. It accurately implements (together with its descendants) the structure of the expiration tool described in Section 3.3 and Section 3.4. An instance of this struct simply wraps a list

of partial expiration tools:

$$\begin{array}{l}\textbf{struct}\ \text{ExpirationTool}\langle{}'c, {}'\text{tcx}\rangle\ \{\\ \quad \text{partial_expiration_tools} : \text{Vec}\langle \&'c\ \text{PartialExpirationTool}\langle{}'c, {}'\text{tcx}\rangle\rangle\\ \}\end{array}$$

A partial expiration tool is represented by the *PartialExpirationTool* struct and contains, besides the sets of places that block something or are blocked, a list of magic wands. Each magic wand enables the expiration of one place from the set of blocking places:

$$\begin{array}{l}\textbf{struct}\ \text{PartialExpirationTool}\langle{}'c, {}'\text{tcx}\rangle\ \{\\ \quad \text{blocking} : \text{HashSet}\langle\text{Place}\langle{}'tcx\rangle\rangle,\\ \quad \text{blocked} : \text{HashSet}\langle\text{Place}\langle{}'tcx\rangle\rangle,\\ \quad \text{magic_wands} : \text{Vec}\langle\&'c\ \text{MagicWand}\langle{}'c, {}'\text{tcx}\rangle\rangle\\ \}\end{array}$$

A magic wand is represented by the *MagicWand* struct. It specifies the place that is expired by this magic wand, the places unblocked by this magic wand, the pledges included on the right-hand side of this magic wand, as well as the nested expiration tool:

$$\begin{array}{l}\textbf{struct}\ \text{MagicWand}\langle{}'c, {}'\text{tcx}\rangle\ \{\\ \quad \text{expired} : \text{places::Place}\langle{}'tcx\rangle,\\ \quad \text{unblocked} : \text{HashSet}\langle\text{Place}\langle{}'tcx\rangle\rangle,\\ \quad \text{pledges} : \text{Vec}\langle\&'c\ \text{typed::Assertion}\langle{}'tcx\rangle\rangle,\\ \quad \text{expiration_tool} : \&'c\ \text{ExpirationTool}\langle{}'c, {}'\text{tcx}\rangle,\\ \}\end{array}$$

The structs use two lifetimes. The first, $'c$, is the lifetime of the whole expiration tool itself. This allows the structs to contain references to their nested components, instead of the nested components themselves. It avoids unnecessary copying, though the performance improvement is negligible. The second, $'\text{tcx}$, is the lifetime of the compiler's typing context. This is allows the expiration tool to reference compiler data, specifically places.

There are two operations defined for expiration tools. The first is construction, implemented in the *encoder::expiration_tool::construct* module and described in more detail in Section 3.6.3. The construction takes the previously computed *InterfaceReborrowing-Graph* and a list of pledges as input and returns an instance of the *ExpirationTool* struct. The second operation is encoding, implemented in the *encoder::expiration_tool::encode*

module and described in more detail in Section 3.6.4. The encoding takes an instance of the *ExpirationTool* struct as input and either returns a Viper expression (implemented in the *expression* sub-module) or a list of Viper statements that package the expiration tool (implemented in the *package* sub-module).

3.6.1. Pledge Parsing

The parsing of pledges is mostly implemented in the *specifications::untyped* module of the *prusti-specs* sub-project. It follows the existing structure of the parsing code, with one exception. Recall that pledges can use the special before_expiry and after_unblocked environments. These two forms come in two variations each. The reference that needs to expire or be unblocked can be mentioned explicitly, written as before_expiry[ref](e) and after_unblocked[ref](e), where ref is the output reference that needs to expire or be unblocked. Alternatively, the reference can be inferred automatically, allowing the shorter forms before_expiry(e) and after_unblocked(e).

Pledge bodies are ultimately compiled as regular Rust expressions. While before_expiry(e) is a proper Rust expression (the function before_expiry called with the parameter e), before_expiry[ref](e) has no equivalent in Rust syntax. To solve this, the token stream passed to the macro is subjected to a pre-processing step in which the following replacements are performed:

$$\begin{aligned} \text{before_expiry}(e) &\mapsto \text{before_expiry}(\&0, e) \\ \text{before_expiry[ref]}(e) &\mapsto \text{before_expiry}(\&\text{ref}, e). \end{aligned}$$

The result of compiling the pre-processed token streams is an expression where both before_expiry(e) and before_expiry[ref](e) are represented as function calls to the same 2-ary before_expiry function. The first parameter is either the reference that needs to expire (if specified in the original pledge) or just 0. The after_unblocked environment is handled symmetrically.

3.6.2. Re-Borrow Relationships

The *InterfaceReborrowingGraph* data structure found in the *encoder::reborrow_signature* module of the *prusti-viper* sub-project implements the re-borrowing graph R introduced in Section 3.3. It is constructed by the *InterfaceReborrowingGraph::construct* method and maps every blocking output reference to a list of input references that it blocks.

Instances of the *InterfaceReborrowingGraph* provide a method *expire_output* to determine the effect of an expiration. This method takes a blocking output reference (which will be expired) and returns a new instance of the *InterfaceReborrowingGraph* together

with a list of input references unblocked by the expiration. The returned instance of the *InterfaceReborrowingGraph* reflects the re-borrowing situation after the expiration – essentially, entries with the expired reference on the left-hand side are removed from the mapping.

3.6.3. Expiration Tool Construction

Every instance of the *ExpirationTool* struct is constructed by three mutually recursive methods: *ExpirationTool::construct*, *PartialExpirationTool::construct*, and *MagicWand::construct*.

All three methods take the re-borrowing graph R as input, which allows the construction process to determine the input references that are unblocked in response to any given expiration. This input changes as the recursion descends to reflect the current re-borrowing situation.

The second input is a list of pledges that are to be embedded in the expiration tool. Recall from Section 3.4 that every pledge p has dependencies $I(p)$ and $O(p)$, the input references that need to be unblocked and output references that need to expire for the pledge to make sense. These dependencies determine where in the expiration tool the pledge is embedded. An initial pre-processing step, described in detail in Section 3.6.5, computes the dependencies for every pledge such that the three construction methods can readily access this information. Note that the re-borrowing graph R together with the list of pledges and their dependencies encodes the re-borrowing–with–pledges graph RP of the method that we introduced in Section 3.4.

The *ExpirationTool::construct* method constructs instances of the *ExpirationTool* struct. It determines the connected components of the re-borrowing–with–pledges graph RP (by calling the *split_reborrows* method from the *encoder::expiration_tool::split_reborrows* module) and turns every one of them into a partial expiration tool (by calling the *PartialExpirationTool::construct* method). Every connected component of RP is represented in the same way as RP itself, ie, by a re-borrowing graph together with a list of pledges and their dependencies.

The *PartialExpirationTool::construct* method constructs instances of the *PartialExpirationTool* struct. It simply calls the *MagicWand::construct* method for every remaining output reference and forwards the re-borrowing graph and list of pledges unchanged.

The *MagicWand::construct* method constructs instances of the *MagicWand* struct. Besides the re-borrowing graph and list of pledges, it receives the output reference whose expiration it is responsible for as another input. First of all, it produces an updated version of the re-borrowing graph reflecting the state after the output reference expired.

This also yields the input references unblocked by this expiration as a by-product. Next, it identifies the *ripe* pledges, ie, the pledges that should be included immediately on the right-hand side of the magic wand because their dependencies are satisfied after the current expiration. Finally, it constructs the nested expiration tool by calling *ExpirationTool::construct* with the updated re-borrowing graph and the list of pledges that are still pending.

3.6.4. Expiration Tool Encoding

After an expiration tool has been constructed, it can be encoded, either as a Viper expression or as a list of Viper statements that package it. The result of encoding an expiration tool as package statements is used by the *encode_package_end_of_method* method, which includes them at the end of a re-borrowing function to prove to Viper that the expiration tool is justified. The result of encoding an expiration tool as a Viper expression is used in a variety of places, including the *encode_postcondition_expiration_tool* method and *construct_vir_reborrowing_node_for_call*.

Viper expressions are produced by three functions (one for expiration tools, one for partial expiration tools, and for magic wands) that recurse on the data they are called to encode. Besides that there is no algorithmic complexity, because the whole structure of the expiration tool has been determined during construction.

Viper statements for packaging are also produced by three functions that again recurse on the data they are called to encode. Because packaging a magic wand requires a representation of the magic wand as an expression, the function responsible for packaging calls out to the function responsible for encoding as an expression.

The call graph of the six encoding functions is shown in Fig. 3.14.

One aspect of the encoding that warrants a more detailed description is the expires_first function that is used in the left-hand sides of the implications that make up the expiration tool branches. Recall that this function takes two arguments. The first is a reference-typed place x returned by the function call, the second is a set of reference-typed places ys also returned by the function call. The function is defined to return true if x expires before all the places from ys, and false otherwise.

In the final encoding, the arguments of this functions are encoded as integers. Thus instead of the expression $\text{expires_first}(x, ys)$, the encoding will contain an expression $\text{expires_first}(x', ys')$, where x' and ys' are integer representations of x and ys. We find x' by fixing an arbitrary mapping from output references to integers, and sending x through this mapping. For ys', we simply take the number of elements in ys. Though this is not very accurate – two sets are represented by the same integer if they contain

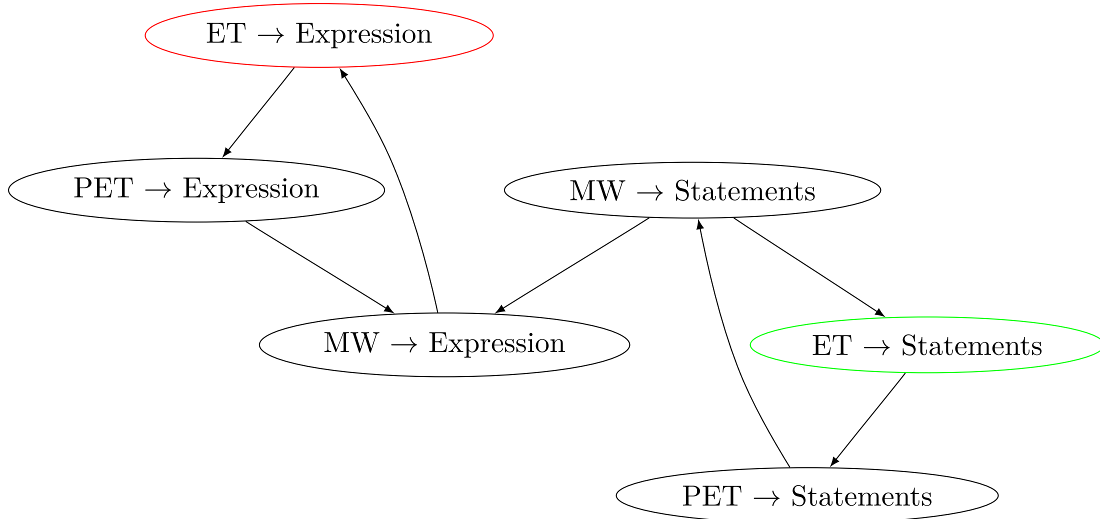


Figure 3.14.: The encoding call graph, where each node represents a function and each edge represents a function call. The entry point for encoding expiration tools as expressions is outlined in red, the entry point for encoding expiration tools as statements is outlined in green. ET = expiration tool; PET = partial expiration tool; MW = magic wand.

the same number of elements – it is enough for our purposes. Because these conflicting encodings of the *ys* parameter always appear in mutually exclusive branches of the expiration tool, a user can never depend on them being different.

3.6.5. Pledge Dependencies

Before embedding a pledge p in the expiration tool, its dependencies – the input references $I(p)$ appearing inside an after_unblocked environment and the output references $O(p)$ appearing inside a before_expiry environment – must be identified. A dependency is an output reference that needs to expire or an input reference that needs to be unblocked before the pledge can be embedded in the expiration tool. This analysis happens in the *encoder::expiration_tool::pledges::analyze* module of the *prusti-viper* sub-project. The entry point of this module is the *identify_dependencies* function, which takes a pledge to its list of dependencies. A single pledge dependency is represented by the *PledgeDependency* type, instances of which contain a context (indicates whether the dependency is due to a before_expiry or due to an after_unblocked expression) and a place (the output place that needs to expire if the dependency is due to a before_expiry expression and the input place that needs to be unblocked if the dependency is due to an after_unblocked expression).

Users can write syntactically incorrect pledges and these errors are caught as a natural byproduct of the dependency analysis. All errors that can occur are defined in the *encoder::expiration_tool::pledges::errors* module. In each case, an appropriate error message is shown to the user. A quick summary follows:

unsupported_assertion The pledge uses a kind of assertion that is not supported in pledges. Right now, only quantifiers are not supported. The reason is explained in Section 3.6.7.

unsupported_expression The pledge uses a kind of expression that is not supported in pledges. Rust has a versatile expression language, defined in the *rustc_hir::ExprKind* struct [3]. The dependency analysis is only implemented for the following subset of this language: *Binary* (a binary operation like $x + y$), *Unary* (a unary operation like $-x$), *Call* (a function call like $f(x)$), *MethodCall* (a method call like $x.f()$), *Field* (a field access like $x.y$), *Match* (a match expression like **match** x $\{\dots\}$), *Block* (a block like $\{\dots\}$), *Path* (a path like $X :: Y$), *Lit* (a literal like 12), *DropTemps* (an internal expression that influences when values are dropped). It should be straightforward to extend this list.

before_expiry_contains_inputs The pledge contains a before_expiry(e) sub-expression where e contains an input reference.

after_unblocked_contains_outputs The pledge contains an after_unblocked(e) sub-expression where e contains an output reference.

ctxt_no_dependencies The pledge contains a before_expiry(e) sub-expression where e does not contain any output reference or it contains an after_unblocked(e) where e does not contain any input reference.

ctxt_multiple_dependencies The pledge contains a before_expiry(e) sub-expression where e contains multiple output references or it contains an after_unblocked(e) sub-expression where e contains multiple input reference.

ctxt_wrong_dependencies The pledge contains a *before_expiry* or *after_unblocked* sub-expression that contains a reference that does not match the reference explicitly specified in square brackets.

ctxt_wrong_expected_dependency The pledge contains a *before_expiry* or *after_unblocked* sub-expression where the explicitly specified reference does not denote an input or output reference.

nested_environments The pledge contains a before_expiry or after_unblocked sub-expression nested inside of another before_expiry or after_unblocked sub-expression.

3.6.6. Client-Side Expiration

Two methods implement the expiration of re-borrows created by function calls. The first one, *construct_vir_reborrowing_node_for_call*, generates the statements to expire a single loan. The second one, *process_expire_borrows*, assembles these statement lists for multiple loans expiring at the same time into a larger structure that encodes the expiration of all these loans. This method handles loans of all kinds, not just re-borrows created by calls.

3.6.7. Differences

Some parts of the implementation, described in more detail below, deviate from the method that we described so far, for various reasons. Either Viper does not support some required features, or there are incompatibilities with the existing encoding, or time constraints made the implementation of the ideal solution impossible. All of these differences can be resolved eventually and once they are, the implementation should match the described method exactly.

Simulating Labelled Magic Wands with Let-Expressions

We use labelled magic wands to allow nested magic wands to refer back to the left- and right-hand side of their ancestors. This is very convenient, but Viper allows right-hand sides of magic wands only to refer to the left-hand side of the same magic wand, and not also to the left- and right-hand sides of magic wands they are nested in. An imperfect work-around uses let-bindings to evaluate expressions in one magic wand and use the result in another (nested) magic wand. For example, we may want to write this magic wand:

$$A \mathrel{-\!\!*_x} (B \mathrel{-\!\!*_y} \mathrm{old}[x : \mathrm{lhs}](e)).$$

To turn this into a form that Viper understands, we insert a let-binding to evaluate the $\mathrm{old}[x : \mathrm{lhs}](e)$ expression at a position where this is possible and use the let-bound variable at a later point:

$$A \mathrel{-\!\!*} \textbf{let } v = \mathrm{old}[\mathrm{lhs}](e) \textbf{ in } (B \mathrel{-\!\!*} v).$$

This approach cannot handle old-expressions under quantifiers, for example:

$$A \mathrel{-\!\!*_x} (B \mathrel{-\!\!*_y} \forall n : \mathrm{old}[x : \mathrm{lhs}](e)).$$

Because n could be free in e, evaluating the old-expression in a let-binding outside of the quantifier does not work. Viewed from another angle, the quantifier (which is just a big

conjunction) contains many old-expressions, one for each value of n. This means we need as many let-bindings, one for each old-expression. But the number of old-expressions is variable, while we have to fix the number of let-expressions during the encoding.

Another problem becomes apparent when e requires some precondition (imposed by a pure function, for example). Take the following example:

$$A \mathrel{-\!\!*}_x (B \mathrel{-\!\!*}_y n > 0 \rightarrow \mathrm{old}[x : \mathrm{lhs}](f(n))),$$

where f requires a positive argument. The old-expression (and thus f) won't be evaluated under the implication, which means the pre-condition of f may not be satisfied.

The implementation can work in two ways. First, we could work with an extended version of Viper's AST that supports labelled magic wands and compile it down to Viper's actual AST eventually. This is very flexible – any part of Prusti can just use labelled magic wands without being aware that Viper doesn't actually support the feature. Second, we can let the code that would use labelled magic wands (if they were supported) simply generate the let-bindings directly instead. This is clearly inferior (because it conflates two easily separable concerns), but much faster to implement in the short term. It is also what was implemented in the end.

Encoding Expiration Tools as Expressions Recall from Section 3.6.4 that the encoding of expiration tools is carried out by three functions that call each other recursively and all return Viper expressions. One encodes expiration tools, one encodes partial expiration tools, and one encodes magic wands. Consider the magic wand

$$A \mathrel{-\!\!*}_x (B \mathrel{-\!\!*}_y \mathrm{old}[x : \mathrm{lhs}](e))$$

from before and the nested magic wand

$$B \mathrel{-\!\!*}_y \mathrm{old}[x : \mathrm{lhs}](e)$$

that it contains in particular. Because the old-expression will be realized with a let binding, the nested magic wand is encoded as

$$B \mathrel{-\!\!*} v,$$

where v is the let-bound variable. To make this expression meaningful, the function that encodes this magic wand must provide enough information to allow one of its (indirect) callers – the one that is encoding the magic wand x – to generate the correct let-binding for v. This information, which we call an *open binding*, comprises three elements. The first element is the name v of the binding. The second element states whether the evaluation should happen on the left- or right-hand side of the magic wand. The third element is the expression e that should be evaluated. Note that the name of the magic

wand that should evaluate e is not explicitly included, because it is implicitly determined by e

Consequently, we update the function that encodes magic wands to return, besides the encoded expression, a set of open bindings. This set contains the open bindings that were created by the magic wand itself and also the open bindings of the nested expiration tool. We update the functions that encode partial expiration tools and expiration tools in the same way, because they can have open bindings too. Partial expiration tools inherit them from the magic wands they contain, and expiration tools inherit them from the partial expiration tools they contain.

Open bindings are not only created. During the encoding of a magic wand, we inspect all current open bindings. The ones that are meant for the current magic wand (call them *ripe* bindings) are materialized as let-bindings. The open bindings that are not ripe are passed on to the caller.

Encoding Expiration Tools as Package Statements We adapt the packaging of expiration tools similarly. The three functions that package expiration tools, partial expiration tools, and magic wands, respectively, are again modified to also return the set of open bindings. As before, the two functions that package expiration tools and partial expiration tools simply return the aggregated open bindings of their children. The function that packages magic wands again materializes ripe open bindings. Because the bindings are open in statements (instead of expressions), they are materialized as local variables (instead of let-bindings).

Inhalation Point of Expiration Tools

The life of a re-borrow returned by a function is defined by two main events. The first one is the creation by the function call (p and q model reference-typed variables):

$$\begin{aligned}&\textbf{label}\ \text{pre}\\&q = f(p)\\&\textbf{label}\ \text{post}\end{aligned}$$

Directly after the call, the expiration tool to expire the re-borrow q is inhaled. Assume that q points to a value of type T_1 and p points to a value of type T_2. For this example, the expiration tool is essentially just a magic wand (ignoring the expires_first condition). The magic wand can contain an encoded pledge pledge R that relates the state of the memory included on the left-hand side with the state of the memory included on the right-hand side:

$$\textbf{inhale}\ P_{T_1}(\text{old}[\text{post}](q.p)) \mathrel{-\!\!*} P_{T_2}(\text{old}[\text{pre}](p.p)) \wedge R.$$

Arbitrary statements can follow, and at some point the re-borrow expires – the second main event. This is modeled by the application of the magic wand:

$$\textbf{apply } P_{T_1}(\text{old[post]}(q.p)) \mathrel{-\!\!*} P_{T_2}(\text{old[pre]}(p.p)) \wedge R.$$

This works as long as $q.p$ is not changed for the lifetime of the re-borrow q. In other words, we need $q.p = \text{old[post]}(q.p)$ at the point where q expires. However, Prusti's model of certain statements, such as $*q = *q + 1$ (assuming q points to an integer), changes $q.p$ even though q still points to the same memory in Rust. This is problematic, because pledges read the state of $\text{old[post]}(q.p)$, but not $q.p$. For example, the increment of q and any subsequent modification would be lost on the pledge.

There are two ways to solve this. The first one is to change the way statements like $*q = *q+1$ are modeled such that they do not change $q.p$. Essentially, $q.p$ should change only if the address saved in q is changed. This is harder than it sounds, but there are ideas that can achieve this. The second way – the way it is actually implemented – is to first move the inhalation point of the magic wand from after the function call to immediately before the magic wand application and then replace every dereference of a returned reference, like $\text{old[post]}(q.p)$, with the current value of its p field, like $q.p$.

Assembling the Expiration CFG

We discussed in Section 3.5.2 how the expiration CFG expires loans in the reverse order they are created. Though necessary, this approach is significantly different from the current implementation and would require changes throughout many different parts of Prusti. The main obstacle is the current assumption that expirations are performed by a list of statements, not an arbitrary CFG. Due to time constraints, our implementation still expires loans using a list of statements. The consequence is that Prusti crashes on certain inputs, but many other examples are verified successfully.

3.7. Evaluation

This section serves two purposes. In the first part, we show some self-contained examples that Prusti can verify with the proposed changes. The second part takes a closer look at the performance of the implementation.

```
#[pledge(after_unblocked(p.x) = before_expiry(*result)))]
#[pledge(after_unblocked(p.y) = 0))]
fn f19(p : &mut Point) → &mut u32 {
    p.y = 0;
    &mut p.x
}
```

Figure 3.15.: The function f_{19}.

```
#[pledge(after_unblocked(p.x) = before_expiry(*result.0)))]
#[pledge(after_unblocked(q.x) = before_expiry(*result.1)))]
#[pledge(after_unblocked(p.y) = after_unblocked(q.y)))]
fn f20⟨'p,'q⟩(p : &'p mut Point, q : &'q mut Point) → (&'p mut u32, &'q mut u32) {
    p.y = q.y;
    (&mut p.x, &mut q.x)
}
```

Figure 3.16.: The function f_{20}.

3.7.1. Examples

We first present examples to demonstrate what is possible to verify and qualitatively evaluate the expressive power of pledges.

The first example, which Prusti was already able to verify before, involves a single input and output reference, shown in Fig. 3.15. Though not necessary, the specification of this function is distributed over two pledges.

The next function, shown in Fig. 3.16, involves two input and output references with unrelated lifetimes. This means the expiration tool will actually have two levels of nesting. Multiple pledges are required to capture the different properties.

What is borrowed can depend on a condition, as Fig. 3.17 illustrates. The pledges state that it depends on the boolean parameter b what result.1 is aliasing.

The old value of inputs can also be used in pledges, as shown in Fig. 3.18.

```
#[pledge(b → after_unblocked(q.x) = before_expiry(*result.1)))]
#[pledge(!b → after_unblocked(p.y) = before_expiry(*result.1)))]
#[pledge(after_unblocked(p.x) = before_expiry(*result.0)))]
fn f21⟨'a : 'b, 'b⟩(
    p : &'a mut Point,
    q : &'b mut Point,
    b : bool
) → (&'a mut u32, &'b mut u32) {
    let x1 = &mut p.x;
    let x2 = if b { &mut q.x } else { &mut p.y };
    (x1, x2)
}
```

Figure 3.17.: The function f_{21}.

```
#[pledge(old(a.x) > 0 → after_unblocked(a.y) = before_expiry(*result.0))]
#[pledge(old(a.x) ≤ 0 → after_unblocked(b.y) = before_expiry(*result.0))]
#[pledge(after_unblocked(b.x) = before_expiry(*result.1))]
#[pledge(after_unblocked(c.x) = before_expiry(*result.2))]
fn f22⟨'a : 'd, 'b : 'd + 'e, 'c : 'f, 'd, 'e, 'f⟩(
    a : &'a mut Point,
    b : &'b mut Point,
    c : &'c mut Point
) → (&'d mut u32, &'e mut u32, &'f mut u32) {
    let t0 = if a.x > 0 { &mut a.y } else { &mut b.y };
    (t0, &mut b.x, &mut c.x)
}
```

Figure 3.18.: The function f_{22}.

```
fn f_23(b : bool) {
    let mut p = Point {x : 10, y : 20};
    let mut q = Point {x : 30, y : 40};
    let (x_1, x_2) = f_21(&mut p, &mut q, b);
    *x_1 = 11;
    *x_2 = 41;
    assert!(p.x = 11);
    if b {
        assert!(q.x = 41);
    } else {
        assert!(p.y = 41);
    }
}
```

Figure 3.19.: The function f_{23}.

```
fn f_24(b : bool) {
    let mut p = Point {x : 10, y : 20};
    let mut q = Point {x : 30, y : 40};
    let (x_1, x_2) = if p.x > 0 {
        f_21(&mut p, &mut q, b)
     } else {
        (&mut p.x, &mut q.y)
     };
}
```

Figure 3.20.: The function f_{24}.

Callers can use the pledges provided by the called function to verify properties about their data, which is shown in Fig. 3.19 for one possible caller of f_{21}. Callers can also call a re-borrowing function in a branch, shown in Fig. 3.20, and still recover permissions after the branch.

These examples show that the implementation is indeed able to verify re-borrowing functions with multiple input and output references.

3.7.2. Performance

This section analyzes the performance of the implementation and identifies its limits. The verification time is almost exclusively determined by the size of the expiration tool, which is exponential in the number of output references. For this reason, we expect there to be a point beyond which verification is infeasible. To determine this more exactly, we construct a sequence of functions. The n-th function f_n will have n input references pointing to a *Point* and n output references pointing to a *u32*. The function signature defines n unrelated lifetimes. The first input and output reference have the first lifetime, the second input and output reference have the second lifetime, and so on. Therefore, f_1 is defined as

$$\textbf{fn } f_1\langle 'a\rangle(p_1 : \&'a \textbf{ mut } \text{Point}) \rightarrow (\&'a \textbf{ mut } \text{u32}),$$

f_2 is defined as

$$\textbf{fn } f_2\langle 'a, 'b\rangle(p_1 : \&'a \textbf{ mut } \text{Point}, p_2 : \&'b \textbf{ mut } \text{Point}) \rightarrow (\&'a \textbf{ mut } \text{u32}, \&'b \textbf{ mut } \text{u32}),$$

and f_3 is defined as

$$\begin{aligned}
&\textbf{fn } f_3\langle 'a, 'b, 'c\rangle(\\
&\quad p_1 : \&'a \textbf{ mut } \text{Point}, \\
&\quad p_2 : \&'b \textbf{ mut } \text{Point}, \\
&\quad p_3 : \&'c \textbf{ mut } \text{Point} \\
&) \rightarrow (\&'a \textbf{ mut } \text{u32}, \&'b \textbf{ mut } \text{u32}, \&'c \textbf{ mut } \text{u32}).
\end{aligned}$$

These functions verify rather quickly. The unrelated lifetimes allow the expiration tool to be optimized, essentially eliminating all the nesting. To force the construction of a nested expiration tool, we add pledges that relate the inputs with each other. The expiration tool is maximally nested if there is one pledge for every pair of input references. Concretely, we add no pledge to f_1, the pledge

$$\#[\text{pledge}(\text{after_unblocked}(p_1.x) = \text{after_unblocked}(p_2.x))]$$

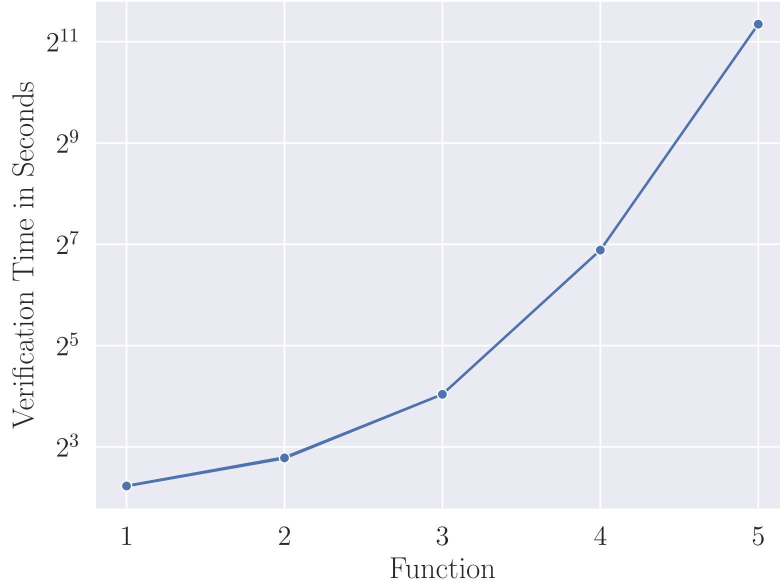


Figure 3.21.: The time in seconds it takes to verify the functions $f_1, \ldots, f_5$ (indicated by the x-label).

	f_1	f_2	f_3	f_4	f_5
mean [seconds]	4.69	6.88	16.43	118.30	2604.77
std	0.05	0.14	0.15	1.07	21.70

Table 3.1.: The mean time in seconds it takes to verify the functions $f_1, \ldots, f_5$, as well as the standard deviation.

to f_2, the pledges

$$\#[\text{pledge}(\text{after_unblocked}(p_1.x) = \text{after_unblocked}(p_2.x))]$$
$$\#[\text{pledge}(\text{after_unblocked}(p_1.x) = \text{after_unblocked}(p_3.x))]$$
$$\#[\text{pledge}(\text{after_unblocked}(p_2.x) = \text{after_unblocked}(p_3.x))]$$

to f_3, and so on.

We verify each function 10 times and report the mean verification time as well as the standard deviation. The results are shown Table 3.1 and plotted in Fig. 3.21 using a logarithmic y-axis. The data does not produce a line, because the expiration tool grows with the factorial function[2], which grows faster than the exponential function, and the time Viper takes to verify a function is not linear in the size of the expiration tool. The verification of f_5 takes roughly 40 minutes, which stretches the limits of practicality.

[2]The expiration tool covering n references contains n magic wands that each contain an expiration tool covering $n - 1$ references.

3.8. Future Work

The work presented so far can be improved or extended in several directions. I highlight the most important ones in the following sections. Section 3.8.1 outlines an alternative way to phrase expiration tools that focuses on lifetimes that expire (in contrast to the approach taken in Section 3.3, which focuses on references that expire). This makes expiration tools smaller, because there can never be more relevant lifetimes than there are relevant output references. Section 3.8.2 explores one important scenario unsupported by the described method – re-borrows in loops. We will see that loops are closely related to tail-recursive functions and pose essentially the same problems in the context of re-borrowing. Finally, Section 3.8.3 describes the second important scenario that is still unsupported – shared re-borrows of mutable references.

3.8.1. Phrasing the Expiration Tool in Terms of Lifetimes

The expiration tool described previously is phrased in terms of output references that expire. Essentially, every magic wand handles the expiration of a single output reference, with permissions for the expiring reference on the left-hand side and permissions for the input references unblocked by the expiration on the right-hand side. This is best illustrated with an example:

$$\textbf{fn}\ f\langle 'a\rangle(p : \&'a\ \textbf{mut}\ \text{Point},\ q : \&'a\ \textbf{mut}\ \text{Point}) \to (\&'a\ \textbf{mut}\ \text{u32}, \&'a\ \textbf{mut}\ \text{u32})\ \{\dots\}$$

Two output references can expire after a call to this function: result.0 and result.1. Either expiration does not unblock anything, because the other reference can still contain an alias of the inputs. The expiration tool provides us with the following magic wand for the expiration order result.0, result.1:

$$Q(\text{result.0}) \mathrel{-\!\!*} (Q(\text{result.1}) \mathrel{-\!\!*} Q(p) \land Q(q)).$$

For the other expiration order, it looks similar:

$$Q(\text{result.1}) \mathrel{-\!\!*} (Q(\text{result.0}) \mathrel{-\!\!*} Q(p) \land Q(q)).$$

The order in which the output references does not seem to matter. Only after both expirations do we get back permissions for both inputs. The reason is found in the type of p, ie., $\&'a$ **mut** Point. The lifetime $'a$ means that as long as any output reference with lifetime $'a$ is still alive, p is blocked. The same is true of q. More generally, expiring one output reference with lifetime $'a$ cannot unblock anything if there is still another output reference with the same lifetime around.

We can exploit this behavior and slightly change the role of magic wands. Instead of one magic wand being responsible for the expiration of a single output reference, we change

it to be responsible for the expiration of a single lifetime – involving multiple output references, potentially. With this change, the expiration tool contains a single magic wand:

$$Q(\text{result.0}) \wedge Q(\text{result.1}) \mathrel{-\!\!*} Q(p) \wedge Q(q).$$

Generally, the nested structure of the expiration tool is kept. The only difference is that the nesting is determined by lifetimes, not output references.

3.8.2. Re-Borrows in Loops

The introduction claimed that loops can show the same behavior that we already know from re-borrowing functions. Consider a linked list

$$\textbf{struct}\ L\langle A\rangle\ \{\ v : A,\ n : \text{Box}\langle L\langle A\rangle\rangle\ \}$$

and a loop that borrows the i-th node of a list instance xs:

$$\begin{array}{l}\textbf{let mut}\ n = \&\textbf{mut}\ \text{xs};\\ \textbf{while}\ i > 0\ \{\\ \quad n = \&\textbf{mut}\ n.n;\\ \quad i = i - 1;\\ \}\end{array}$$

Clearly, n aliases xs after the loop. Consequently, xs is blocked while n is still used. What happens after n expires? We should restore permissions for xs – but how? Viper is confused by the loop and does not understand why folding permissions for xs is possible. It demands proof, which we can provide in the form of a magic wand that has permissions for n on the left-hand side and permissions for xs on the right-hand side. This magic wand is included in the loop variant (and proved again at the end of every iteration using the magic wand from the previous iteration), which convinces Viper of it's validity.

Now consider the loop written as a tail-recursive function:

$$\begin{array}{l}\textbf{fn}\ f_{25}(n : \&\textbf{mut}\ L\langle A\rangle) \to \&\textbf{mut}\ L\langle A\rangle\ \{\\ \quad \textbf{if}\ i > 0\ \{\ f(\&\textbf{mut}\ n.n,\ i - 1)\ \}\ \textbf{else}\ \{\ n\ \}\\ \}\end{array}$$

This essentially requires the same magic wand. Instead of proving the magic wand at the end of every iteration using the magic wand from the previous iteration, we now prove it at the end of every call using the magic wand from the recursive invocation. The proof required in the base case is virtually equivalent to the proof required to establish

the loop invariant initially. We are looking at two incarnations of the same fundamental thing.

Of course, loops can also create multiple re-borrows. To show this, we need trees:

$$\textbf{struct}\ T\langle A\rangle\ \{\ v : A,\ l : \text{Box}\langle T\langle A\rangle\rangle,\ r : \text{Box}\langle T\langle A\rangle\rangle\ \}$$

This loop borrows a node of *xs* into n_1 and a node of either *xs* or *ys* into n_2:

$$\begin{aligned}
&\textbf{let mut}\ n_1 = \&\textbf{mut}\ \text{xs};\\
&\textbf{let mut}\ n_2 = \&\textbf{mut}\ \text{ys};\\
&\textbf{while}\ i > 0\ \{\\
&\quad n_1 = \&\textbf{mut}\ n_1.l;\\
&\quad n_2 = \textbf{if}\ \text{cond}()\ \{\ \&\textbf{mut}\ n_1.r\ \}\ \textbf{else}\ \{\ \&\textbf{mut}\ n_2.r\ \};\\
&\quad i = i - 1;\\
&\}
\end{aligned}$$

This results in two aliases n_1, n_2 for *xs* and *ys* that behave in the same way as the outputs of this re-borrowing function:

$$\begin{aligned}
&\textbf{fn}\ f_{26}\langle 'a : 'b, 'b\rangle(\\
&\quad \text{xs} : \&'a\ \textbf{mut}\ T\langle A\rangle,\\
&\quad \text{ys} : \&'b\ \textbf{mut}\ T\langle A\rangle\\
&) \to (\&'a\ \textbf{mut}\ T\langle A\rangle,\ \&'b\ \textbf{mut}\ T\langle A\rangle)
\end{aligned}$$

Expiring n_1 unblocks *xs*, but only if n_2, which can also borrow from *xs*, already expired before. Expiring n_2 unblocks *ys* and, if n_1 already expired before, *xs* as well. These are the same phenomena expiration tools are designed to handle, which likely means they will play an important role for such loops as well. Indeed, it is again possible to rewrite this example as a tail-recursive function.

The two examples demonstrate that loops can create re-borrows very much like re-borrowing functions. The possibility to translate re-borrowing loops to tail-recursive re-borrowing functions reinforces this similarity. It is therefore reasonable to assume that the expiration tool that we developed for re-borrowing functions can also support re-borrowing loops. However, the representation of loops in Prusti is distinct from functions. Consequently, just as there is a certain amount of work needed to integrate the expiration tool with functions, it is necessary to determine how loops are integrated with the expiration tool.

3.8.3. Shared Re-Borrows of Mutable References

The previous chapters never talked about shared references; they exclusively considered mutable references and their implications for the verification of Rust programs. It is true

that shared references are less exciting. For example, this function requires no special handling at all:

$$\textbf{fn}\ f_{27}\langle 'a : 'b, 'b\rangle(\text{xs} : \&'a\ T\langle A\rangle,\ \text{ys} : \&'b\ T\langle A\rangle) \to (\&'a\ T\langle A\rangle,\ \&'b\ T\langle A\rangle).$$

A call to this function takes no permissions away from the caller, a consequence of shared references being *copy*-types in Rust. The outputs keep the inputs alive, but the expiration of an output does not necessitate the application of a magic wand, because the inputs were never blocked.

Another function involving shared references is impossible to implement in Rust:

$$\textbf{fn}\ f_{28}\langle 'a : 'b, 'b\rangle(\text{xs} : \&'a\ T\langle A\rangle,\ \text{ys} : \&'b\ T\langle A\rangle) \to (\&'a\ \textbf{mut}\ T\langle A\rangle,\ \&'b\ \textbf{mut}\ T\langle A\rangle).$$

The output lifetimes suggest that the returned references borrow the input data or static memory. The first option is not possible, because the input data is behind a shared reference. The second option is not possible, because this creates mutable references to static memory – something that Rust forbids.[3]

Finally, one case remains – shared re-borrows of mutable references:

$$\textbf{fn}\ f_{29}\langle 'a : 'b, 'b\rangle(\text{xs} : \&'a\ \textbf{mut}\ T\langle A\rangle,\ \text{ys} : \&'b\ \textbf{mut}\ T\langle A\rangle) \to (\&'a\ T\langle A\rangle,\ \&'b\ T\langle A\rangle).$$

Though this does not create an interesting specification scenario (we cannot modify the inputs via the re-borrows), the permission handling requires attention. Rust guarantees that data behind shared references cannot change. For this reason, the shared references degrade the aliased data to be read-only until they expire. After the expiration, the previously aliased data is mutable again.

Prusti already provides methods to deal with shared aliases of mutable data. Most importantly, it can encode the temporary degradation of mutable data to read-only data while shared aliases are alive. This existing encoding must be integrated with the expiration tool, which is mostly implementation work.

[3]Both these points ignore a possible *unsafe* implementation of the function. Arenas, which are a type of allocator, have a method *alloc* with a similar signature. It takes a shared reference to the arena and produces a mutable reference to data owned by the arena. To make this safe, the arena guarantees that it never produces aliasing references – much like C's *malloc*.

4. References Inside Structs

Prusti supports only a subset of types. Imagine a type as a tree, and every node has to be either a primitive, tuple, struct, enum, or reference. Furthermore, reference nodes can only appear at the root. So far, we restricted re-borrowing functions to these supported types. But Rust's type system is far richer. In this chapter, we explore how to relax these restrictions: the requirement that every node must be a primitive, tuple, struct, enum, or reference stays, but we do not restrict reference nodes to the root anymore. This chapter does not lay out a finished solution. Instead, it tries to illustrate the challenges introduced by this generalization and possible directions an eventual solution could take.

4.1. Permission Management

We explore the permission management aspect of the problem using a simple linked list type as an example:

$$\begin{aligned}
&\textbf{enum}\ \text{Option}\langle T\rangle\ \{\ \text{None}, \text{Some}(T)\ \}\\
&\textbf{struct}\ \text{List}\langle 'a, T\rangle\ \{\\
&\quad v : \&'a\ \textbf{mut}\ T,\\
&\quad n : \text{Option}\langle \&'a\ \textbf{mut}\ \text{List}\langle 'a, T\rangle\rangle\\
&\}
\end{aligned}$$

The option type is encoded as the following Viper predicates:

$$\begin{aligned}
\text{none}(x : \text{Ref}) &= \emptyset\\
\text{some}_T(x : \text{Ref}) &= \text{acc}(x.v) \land P_T(x.v)\\
\text{option}_T(x : \text{Ref}) &= \text{acc}(x.d) \land \text{acc}(x.v) \land\\
&\quad\; x.d = 1 \rightarrow \text{none}(x.v) \land\\
&\quad\; x.d = 2 \rightarrow \text{some}_T(x.v).
\end{aligned}$$

The list type is encoded as the following Viper predicate:

$$\begin{aligned}\text{list}_T(x : \text{Ref}) = \text{acc}(x.v) \wedge \text{refmut}_T(x.v)\\ \text{acc}(x.n) \wedge \text{option}_{\text{List}[T]}(x.n).\end{aligned}$$

The memory owned by a linked list with two elements is shown in Fig. 4.1. Now consider the following function:

$$\textbf{fn } f_{30}\langle a\rangle(l : \text{List}\langle' a, T\rangle) \rightarrow \&' a \textbf{ mut } T \; \{\dots\}$$

It receives a linked list l of (mutable references to) T's and returns a single mutable reference to a T (call it r). The lifetimes reveal that the returned reference can be a re-borrow of references owned by the list. As a result of the same aliasing rules that motivated the previous chapter, the compiler must block access to anything borrowed by l while r is still alive:

$$\begin{aligned}&\textbf{let mut } v = 0;\\ &\textbf{let } l = \text{List}(\&\textbf{mut } v, \text{None});\\ &\textbf{let } r = f_{30}(l);\\ &// \textit{ v is blocked due to r}\\ &*r = 1;\\ &// \textit{ v is unblocked}\\ &\text{assert!}(v = \dots);\end{aligned}$$

Let's analyze this at the Viper level. Ignoring the part where permissions are returned to the caller after re-borrows expire, the method f_{30} is basically encoded like this:

$$\begin{aligned}&\textbf{method } f_{30}(l : \text{Ref}) \textbf{ returns } (r : \text{Ref})\\ &\quad \textbf{requires } \text{List}_T(l)\\ &\quad \textbf{ensures } \text{refmut}_T(r)\\ &\{...\}\end{aligned}$$

Naturally, f_{30}'s postcondition doesn't include permissions for the list l itself, because it is moved into the method call and therefore lost to the caller. But the postcondition also doesn't include permissions for the T-values borrowed by l, because these permissions may be included in the $\text{refmut}_T(r)$ predicate instance already. But this changes once r expires and the permissions in $\text{refmut}_T(r)$ are available again. It is possible to imagine the following magic wand:

$$P_T(\text{old[post]}(r.p)) \mathrel{-\!\!*} \text{permissions for all } T\text{-values borrowed by } l.$$

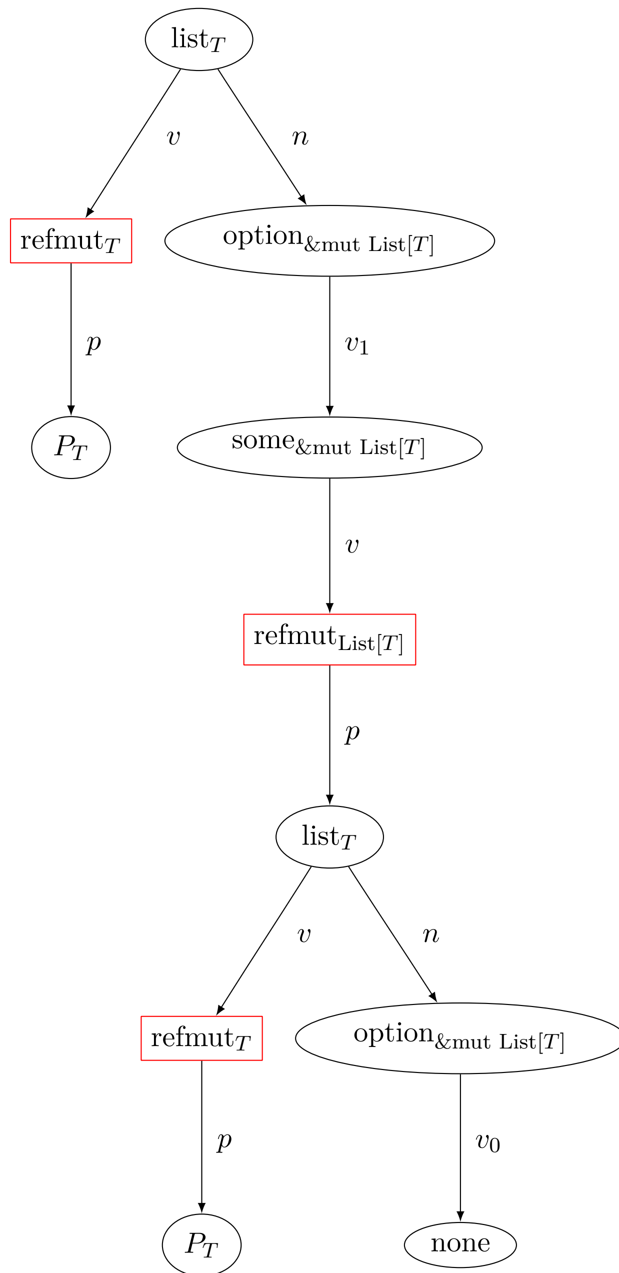


Figure 4.1.: The memory owned by a linked list with two elements. Nodes corresponding to references are drawn as a red square.

How can we formulate the right-hand side of this magic wand? It needs to refer to values below l evaluated in the pre-state of the method call. Writing this out for the first two nodes, we get the following:

$$P_T(\text{old[post]}(r.p)) \mathrel{-\!\!*} P_T(\text{old[pre]}(l.v.p)) \wedge P_T(\text{old[pre]}(l.n.v_1.v.p.v.p)) \wedge \dots$$

Once again, notice that the right-hand side does not contain permissions for $P_T(l.v.p)$, since the caller does not have permissions to access $l.v.p$ after the call. Instead, it contains permissions for $P_T(\text{old[pre]}(l.v.p))$. This is well defined, since the caller had permissions to access $l.v.p$ before the call. However, writing the right-hand side like this is not viable in practice where the length of the list is usually determined at runtime (and, more fundamentally, can vary between calls). What is needed is a statically constructable expression that does not depend on the exact shape of l and still captures permissions for all T-values borrowed by l.

Viper provides two tools for this: predicates and quantified permissions. The need for old-expressions disqualifies predicates. But quantified permissions provide a way out. Assume one function $\text{length}(l)$ that returns the length of l and one function $\text{get_nth}(l, n)$ that returns the n-th T-value borrowed by l:

$$\begin{aligned}
\text{get_nth}(l, 0) &= l.v.p \\
\text{get_nth}(l, 1) &= l.n.v_1.v.p.v.p \\
\text{get_nth}(l, 2) &= l.n.v_1.v.p.n.v_1.v.p.v.p \\
&\dots
\end{aligned}$$

Now we can write down the magic wand:

$$P_T(\text{old[post]}(r.p)) \mathrel{-\!\!*} \forall\, 0 \le i < \text{length}(l) : P_T(\text{old[pre]}(\text{get_nth}(l, i))).$$

The functions length and get_nth are very specific to the shape of the data structure. For a binary tree, we would need another set of functions that allow us to enumerate permissions for all values referenced by the tree nodes. Ultimately, we want to generate the magic wand for general data structures. We thus need a general method to enumerate the unblocked permissions.

Start with a *path expression* that describes a sequence of field accesses, but leaves the root open. Examples for path expressions are $_.v.p$ and $_.n.v_1.v.p.v.p$. Note that path expressions are runtime values, which means we can write Viper functions that operate on them. A path expression can be applied to a root to produce a value. For example, the path expression $_.v.p$ applied to the root l produces a value equivalent to the expression $l.v.p$. This operation is implemented by the $\text{apply}(r, p)$ function. In general, it allows a simple recursive definition:

$$\begin{aligned}
\text{apply}(r, p) &= \text{if empty}(p)\ \{r\} \text{ else } \{\text{apply}(r', \text{tail}(p))\} \\
r' &= \text{apply_one}(r, \text{first}(p)),
\end{aligned}$$

where the apply_one expressions applies the first element of p to r in a simple case distinction.

Path expressions allow a reformulation of get_nth:

$$\begin{aligned}\text{get_nth}(l, 0) &= \text{apply}(l, _.v.p)\\ \text{get_nth}(l, 1) &= \text{apply}(l, _.n.v_1.v.p.v.p)\\ \text{get_nth}(l, 2) &= \text{apply}(l, _.n.v_1.v.p.n.v_1.v.p.v.p)\\ &\dots\end{aligned}$$

The exact structure of data is not known statically due to enums. For example, lists are terminated after a statically unknown number of nodes when the field containing the next list node is a None value. This means not every possible path expression is actually valid for every instance of the data structure. Another function can test this property at runtime:

$$\begin{aligned}\text{valid}(r, p) &= \text{if empty}(p)\ \{\top\}\ \text{else}\ \{v \wedge \text{valid}(r', \text{tail}(p))\}\\ v &= \text{valid_one}(r, \text{first}(p))\\ r' &= \text{apply_one}(r, \text{first}(p)).\end{aligned}$$

The functions apply and valid are closely related in the sense that the latter is a precondition of the former: evaluating $\text{apply}(r, p)$ requires $\text{valid}(r, p)$ to be true.

The final element we need is a boolean-valued function over path expressions that is true if the argument corresponds to a reference that points to an unblocked value. For the list example, the expressions

$$\begin{gathered}\text{unblocked}(_.v)\\ \text{unblocked}(_.n.v_1.v.p.v)\\ \text{unblocked}(_.n.v_1.v.p.n.v_1.v.p.v)\\ \dots\end{gathered}$$

evaluate to true, while all other expressions evaluate to false. Now we can give another formulation of the magic wand:

$$P_T(\text{old[post]}(r.p)) \mathrel{-\!\!*} \forall p : \text{valid}(l, p) \rightarrow \text{unblocked}(p) \rightarrow P_T(\text{old[pre]}(\text{apply}(l, p).p))$$

This section demonstrated some challenges introduced by allowing structures with references – the statically unbounded number of references that are involved, most importantly – and outlined a general approach that an eventual solution could build upon. Many questions are left open, which future work will have to answer. For one, it is

unclear which magic wands to actually generate. For this, we need to better understand Rust's blocking rules, which we attempt in Section 4.2. Then there is the issue of packaging the magic wands. Since they contain quantified permissions, we likely need ghost code to assemble the resources on the left- and right-hand side. For example, this could involve a function that consumes permissions for a list and produces permissions for all T-values in the list. It is also not obvious how clients can use these magic wands to obtain permissions for unblocked references. It should be possible in theory (because the right-hand side includes permissions for everything that is unblocked by the expiration), but communicating this to Viper can prove tricky.

4.2. Re-Borrow Relationships

In Section 3.1 we described for re-borrowing functions that involve simple references which output references can borrow from which input references, which also helped to explain the blocking behavior. Now we want to adapt these rules to account for references that appear inside structs. We still exclusively consider mutable references.

Consider three structs A, B, C:

$$\begin{aligned}
&\textbf{struct}\ A\langle 'x, 'y\rangle\ \{\ b_1 : B\langle 'x\rangle,\ b_2 : \&'x\ \textbf{mut}\ B\langle 'y\rangle,\ c : \&'y\ \textbf{mut}\ C\langle 'y\rangle\ \}\\
&\textbf{struct}\ B\langle 'x\rangle\ \{\ f : \&'x\ \textbf{mut}\ \text{u32}\ \}\\
&\textbf{struct}\ C\langle 'x\rangle\ \{\ c : \&'x\ \textbf{mut}\ C\langle 'x\rangle,\ f : \&'x\ \textbf{mut}\ \text{u32}\ \}
\end{aligned}$$

We can draw any type as a tree with back-edges. Generally, the root node of the tree for a type T_1 has one sub-tree for every type T_2 obtainable from T_1 via a single projection step (field access, dereference, ...), generated in the same way. If possible, we replace a sub-tree for T_2 with a back-edge to an ancestor node that already represents T_2. The result of this construction for the type $A\langle 'x, 'y\rangle$ is shown in Fig. 4.2. A single projection step from $A\langle 'x, 'y\rangle$ can produce values of type $B\langle 'x\rangle$, $\&'x$ **mut** $B\langle 'x\rangle$, and $\&'y$ **mut** $C\langle 'y\rangle$, visible in the tree as the three children of the node for $A\langle 'x, 'y\rangle$. Instances of $C\langle 'y\rangle$ can produce values of type $\&'y$ **mut** $C\langle 'y\rangle$ and $\&'y$ **mut** u32 with a single projection step. The latter becomes a sub-tree of the node for $C\langle 'y\rangle$. The former is realized with a back-edge to the node for $\&'y$ **mut** $C\langle 'y\rangle$, since it is an ancestor.

Since a place is blocked after a function call if there is the possibility that some reference is aliasing it, we must answer two questions. First, Section 4.2.1 explains which data owned by the caller we can create references for (borrow sources). These are the places that are blocked after a call if the created references can be communicated to the caller. This leads to the second question, covered in Section 4.2.2. Here we explain where references can be stored such that they are accessible to the caller (borrow sinks). While these two sections fall short of explaining the blocking behavior completely, the thoughts they contain may be useful in an eventual formalization.

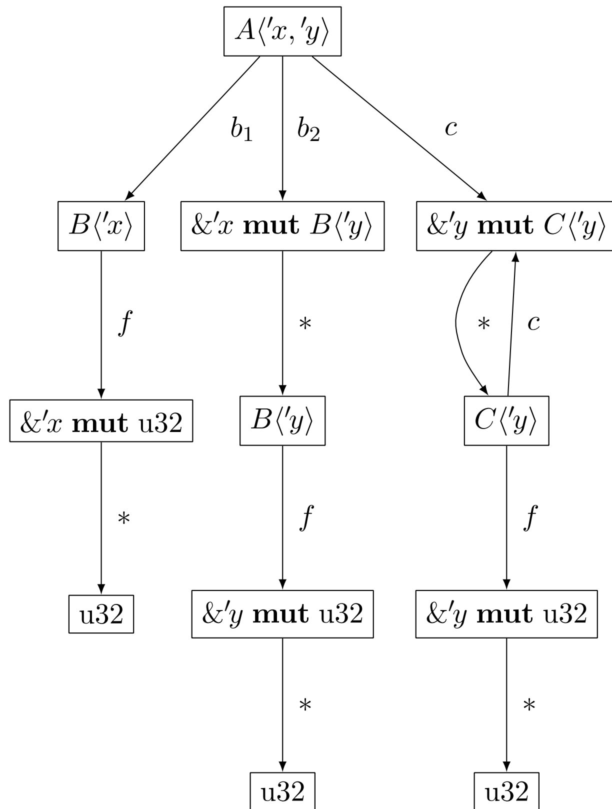


Figure 4.2.: The type $A\langle 'x, 'y\rangle$ represented as a tree with back-edges. The nodes are types and an edge from T_1 to T_2 indicates that a value of type T_1 gives access to a value of type T_2 via a single projection step (field access, dereference, . . .).

4.2.1. Borrow Sources

Take borrow sources first and consider a value of type $A\langle 'x, 'y\rangle$ with no relation between $'x$ and $'y$. The expression &**mut** $_.b_1$ (where _ indicates the argument of type A) creates a reference of type &$'x$ **mut** $B\langle 'x\rangle$, but this is uninteresting because the borrowed data is not owned by the caller. The expression &**mut** $*_.b_1.f$ creates a reference of type &$'x$ **mut** u32. The referenced u32 is owned by the caller, because we didn't receive the value itself, but a reference to it. Note that the expression &**mut** $*_.b_1.f$ cannot create a reference of type &$'y$ **mut** u32, because $_.b_1.f$ only lives for $'x$, and $'x$ is not known to outlive $'y$. The expression &**mut** $*_.b_2$ creates a reference of type &$'x$ **mut** $B\langle 'y\rangle$. Again, the referenced value is owned by the caller, because we followed at least one reference to obtain it. One might think that the expression &**mut** $*(*_.b_2).f$ allows us to create a reference of type &$'y$ **mut** u32, but this is not the case. Because $_.b_2$ has the lifetime $'x$, no reference obtained from $*_.b_2$ can outlive $'x$. In other words, the lifetime of the reference created by &**mut** $*(*_.b_2).f$ must be outlived by both $'x$ and $'y$. Because $'x$ and $'y$ are unrelated, the compiler conservatively assumes that the only lifetime that satisfies this requirement is the empty lifetime, making re-borrows impossible.

We now formulate rules that generalize the observations from the previous paragraph. Given some value, we want to determine for every projection p out of this value the most permissive lifetime that a reference to the corresponding place can have. For this, collect the lifetimes of all references dereferenced by p and compute their intersection:

$$\text{intersection}(\{'x\}) = 'x$$

$$\text{intersection}(\{'x, 'y, \dots\}) = \begin{cases} \text{intersection}(\{'y, \dots\}) & \text{if } 'x : 'y \\ \text{intersection}(\{'x, \dots\}) & \text{if } 'y : 'x \\ \emptyset & \text{otherwise} \end{cases}$$

We can try this for the value of type $A\langle 'x, 'y\rangle$. The place $*_.b_1.f$ dereferences a single reference, $_.b_1.f$, which has the lifetime $'x$. The intersection of the singleton set is the lifetime contained in it, ie $'x$. This means the longest-living reference to $*_.b_1.f$ that we can create has lifetime $'x$. The place $*(*_.b_2).f$ dereferences $_.b_2$ and $(*_.b_2).f$, with lifetimes $'x$ and $'y$, respectively. The intersection of these two lifetimes is empty, because they are unrelated. This means the longest-living reference to $*(*_.b_2).f$ that we can create has the empty lifetime. The place $*(*_.c).f$ also dereferences references with lifetimes $'x$ and $'y$, yielding the same result.

What about $*(*(*_.c).c).f$, $*(*(*(*_.c).c).c).f$, $\dots$? These places still just dereference references with the lifetime $'y$. We see in Fig. 4.2 that the projections performed by these places follow the cycle between &$'y$ **mut** $C\langle 'y\rangle$ and $C\langle 'y\rangle$ once, twice, or more often. Following the cycle does not visit new types, which means it cannot add dereferences of references with other lifetimes. This allows us to handle these places together. We need

place	borrow source lifetime	borrow sink lifetime
$_.b_1$	–	–
$_.b_1.f$	–	–
$_.b_1.f.*$	$'x$	–
$_.b_2$	–	–
$_.b_2.*$	$'x$	–
$_.b_2.*.f$	$'x$	$'y$
$_.b_2.*.f.*$	$\emptyset$	–
$_.c$	–	–
$_.c.*[.c.*]$	$'y$	–
$_.c.*[.c.*].c$	$'y$	$'y$
$_.c.*[.c.*].f$	$'y$	$'y$
$_.c.*[.c.*].f.*$	$'y$	–

Figure 4.3.: Places accessible from a value of type $A\langle 'x, 'y\rangle$ together with the maximum lifetime a reference to the place can have (borrow source lifetime) and the minimum lifetime a reference stored in the place must have (borrow sink lifetime). Places that are not owned by the caller or where references cannot be stored in are indicated with a "–" on the right side.

two notational extensions to refer to these places as a unit. First, we add the option to write dereferences as field accesses, turning $*(*(*(*_.c).c).c).f$ into $_.c.*.c.*.c.*.f.*$. Second, we add a square bracket notation to denote arbitrary repetition. This allows us to write $_.c.*[.c.*].f.*$ for all places that start with $_.c.*$, followed by arbitrarily many instances of $.c.*$, followed by $.f.*$. All of these places are equivalent with respect to the lifetimes of the references they dereference.

Finally, we build a list, given in Fig. 4.3, of all accessible places together with the most permissive lifetime that a reference to each such place can have. A place from this list is blocked after a call if we can also save the reference in a location that the caller can access after the call, which is the topic of the next subsection.

4.2.2. Borrow Sinks

Having obtained a reference with some lifetime to data owned by the caller, we now try to store it in a place that is accessible to the caller too. The rules for this are a bit simpler than for borrow sources. We still consider the list of all places from Fig. 4.3. To start, exclude the ones that are not owned by the caller (ie, are not behind at least one dereference). Next, also exclude all places that are not of a reference-type, because we naturally cannot store references in them. What's left are the reference-typed places that are owned by the caller. In such a place of type $\&'a$ **mut** T, for some $'a$ and T, we can store any reference that has a lifetime outliving $'a$. Interestingly, the lifetimes of

references that were dereferenced to get to this place are unimportant here.

4.3. Pledges

This section demonstrates some verification scenarios and pledges that one could write to capture suitable functional properties. It does not provide a comprehensive pledge syntax suitable for functions involving structures with references, but instead shows exemplary properties that such a syntax must be able to capture eventually.

4.3.1. Single Non-Recursive Struct

We begin with an example struct that contains a reference:

$$\textbf{struct}\ S\langle 'a, T\rangle\ \{\ x : \&'a\ \textbf{mut}\ T\ \}$$

Consider a function that takes a reference to the struct and changes the contained reference to be 0:

$$\begin{aligned}
&\#[\textbf{pledge}(\text{after_unblocked}(*\text{old}[\text{pre}]((*d).x)) = 0)] \\
&\text{fn}\ f_{31}\langle 'a\rangle(d : \&'a\ \textbf{mut}\ S\langle 'a, \text{u32}\rangle)\ \{ \\
&\quad *d.x = 0; \\
&\}
\end{aligned}$$

Because the contained reference is (briefly) blocked after the call, a pledge is required. This pledge needs to describe the following property: take the reference that was passed into the function via $d.x$ – formally written as $\text{old}[\text{pre}]((*d).x)$ – and dereference it after it is unblocked. It's value will be 0.

This function stores an alias for $d.x$ in the output parameter x:

$$\begin{aligned}
&\#[\textbf{pledge}(\text{after_unblocked}(*\text{old}[\text{pre}]((*d).x)) = \text{before_expiry}(*\text{old}[\text{pre}](*x)))] \\
&\textbf{fn}\ f_{32}\langle 'a\rangle(d : \&'a\ \textbf{mut}\ S\langle 'a, \text{u32}\rangle,\ x : \&\textbf{mut}\ \&'a\ \textbf{mut}\ \text{u32})\ \{ \\
&\quad *x = \&\textbf{mut}\ *d.x; \\
&\}
\end{aligned}$$

The pledge describes the following property. Take the reference that was passed into the function via $d.x$ – formally written as $\text{old}[\text{pre}]((*d).x)$ again – and dereference it after it is unblocked. Then take reference that was passed into the function via x – formally

written as old[pre]($*x$) – and dereference it immediately before it expires. These values turn out to be the same, ie, old[pre](($*d$).x) and old[pre]($*x$) are aliases.

The following function is a slight variation of f_{32}, it just returns the alias instead of saving it to an output parameter:

$$\#[\textbf{pledge}(\text{after_unblocked}(*\text{old[pre]}((*d).x)) = \text{before_expiry}(*\text{result}))]$$
$$\textbf{fn}\ f_{33}\langle' a\rangle(d : \&' a\ \textbf{mut}\ S\langle' a, \text{u32}\rangle) \to \&' a\ \textbf{mut}\ \text{u32}\ \{$$
$$\quad \&\textbf{mut}\ *d.x$$
$$\}$$

Another variation of f_{32} stores the caller-accessible alias for $d_1.x$ in another instance of S:

$$\#[\textbf{pledge}(\text{after_unblocked}(*\text{old[pre]}((*d_1).x)) = \text{before_expiry}(*\text{old[pre]}((*d_2).x)))]$$
$$\textbf{fn}\ f_{34}\langle' a\rangle(d_1 : \&' a\ \textbf{mut}\ S\langle' a, \text{u32}\rangle,\ d_2 : \&\textbf{mut}\ S\langle' a, \text{u32}\rangle)\ \{$$
$$\quad d_2.x = \&\textbf{mut}\ *d_1.x;$$
$$\}$$

4.3.2. Single Recursive Struct

Structs can contain instances of themselves. For example, we can define a linked list of mutable references:

$$\textbf{struct}\ L\langle' a, T\rangle\ \{\ v : \&' a\ \textbf{mut}\ T,\ n : \text{Box}\langle L\langle' a, T\rangle\rangle\ \}$$

This list is infinite, which makes the following examples easier to write – we don't have to worry about the length. A finite version is easily defined by using enums. Assume a pure *get* method takes a parameter i and returns a shared reference to the i-th list node.

We can write a function that returns a mutable reference to the i-th value of the list:

$$\#[\textbf{pledge}(\text{after_unblocked}(*\text{old[pre]}((*\text{self.get}(i)).v)) = \text{before_expiry}(*\text{result}))]$$
$$\textbf{fn}\ f_{35}\langle' a\rangle(\&' a\ \textbf{mut}\ \text{self},\ i : \text{u32}) \to \&' a\ \textbf{mut}\ T\ \{$$
$$\quad \textbf{if}\ i = 0\ \{$$
$$\quad\quad \&\textbf{mut}\ *\text{self}.v$$
$$\quad \}\ \textbf{else}\ \{$$
$$\quad\quad f_{35}(\text{self}.n, i - 1)$$
$$\quad \}$$
$$\}$$

The specification for this is similar to what we saw before. We use *after_unblocked* and *before_expiry* to communicate an alias relationship between a reference passed into the call – $\text{old}[\text{pre}]((*\text{self.get}(i)).v)$ – and a reference available after the call – result.

We can create many aliases, as the following function that returns a list containing every second value of the input list shows:

$$
\begin{aligned}
&\#[\textbf{pledge}(\forall i.\ \textbf{let}\\
&\qquad p = \text{old}[\text{pre}]((*\text{self.get}(2i)).v),\\
&\qquad q = \text{old}[\text{post}](\text{result.get}(i).v)\\
&\quad \textbf{in}\ \text{after_unblocked}(*p) = \text{before_expiry}(*q))]\\
&\textbf{fn}\ f_{36}\langle 'a\rangle(\&'a\ \textbf{mut}\ \text{self}) \to L\langle 'a, T\rangle\ \{\\
&\quad L\ \{\ v : \&\textbf{mut}\ *l.v,\ n : \text{Box::new}(f_{36}(\text{self}.n.n))\ \}\\
&\}
\end{aligned}
$$

Note that q is obtained in the post-state of the call. To see why this is necessary, observe two things. First, the pledge is activated once all references in the returned list expire. Second, a caller can change the values in the returned list, for example by pointing some of them to different memory locations. This breaks alias relationships guaranteed by f_{36}. Therefore, $\text{result.get}(i).v$ must be evaluated in the post-state of the call. Technically, this is also necessary in the previous examples.

This section explored possible specifications for some of the more advanced re-borrowing functions involving structs with references. We saw that the familiar before_expiry and after_unblocked environments still provide a powerful tool to formulate functional properties. Determining the correct state in which sub-expressions must be evaluated requires extra care. Often, a reference is obtained in the pre-state of the function call and only dereferenced later, when the place it references is unblocked. The pledge of f_{36} makes this especially clear. Eventually, pledges must be embedded in the encoded program. Once future work paints a clearer picture of the way permissions around re-borrowing functions involving structs with references are handled, the embedding of pledges can be approached. The examples given here will help to test ideas.

A. Terms and Notation

Re-Borrowing Function. A function that has reference-typed arguments and reference-typed return places. Crucially, a re-borrowing function can create aliases by returning references to data referenced by input references.

Input Reference. A reference-typed argument of a re-borrowing function.

Output Reference. A reference-typed return place of a re-borrowing function.

Pledge. A kind of specification that can relate the final state of output references with the initial state of input references after they are unblocked.

Re-Borrowing Graph R. A bipartite graph that indicates for a function f which input references any output reference can borrow from. It has one node for every input reference, one node for every output reference, and an edge between an input and an output node if the output can borrow from the input.

Re-Borrowing-With-Pledges Graph RP. The re-borrowing graph R extended with edges to indicate input references and output references that appear together in the same pledge. For every pledge p, edges are inserted such that the nodes corresponding to $I(p) \cup O(p)$ form a clique.

Expiration Tool. A Viper resource that allows the caller of a re-borrowing function to give up permissions for expired references and obtain permissions for unblocked places in return. It also provides the caller with the facts from pledges.

Partial Expiration Tool. An expiration tool that is responsible for one connected component of the re-borrowing-with-pledges graph RP. Many partial expiration tools make up one "whole" expiration tool.

expires_first. A boolean-valued function of two arguments that is defined to be true when the first argument, which is an output reference of a re-borrowing function, expires before all elements of the second argument, which is a set of output references of the same re-borrowing function. This is used in the expiration tool to provide the caller with the correct magic wand that can expire the next expiring output reference.

before_expiry. An environment that can be used in pledges to evaluate an expression immediately before a reference expired.

after_unblocked. An environment that can be used in pledges to evaluate an expression immediately after a place was unblocked.

$I(p)$**.** The input references mentioned by a pledge p within *after_unblocked* environments.

$O(p)$**.** The output references mentioned by a pledge p within *before_expiry* environments.

$Q(x)$**.** Shortcut notation to denote permissions passed into or returned from a function call. If x corresponds to an argument of type &mut T, then $Q(x)$ expands to $P_T(\text{old[pre]}(x.p))$, where P_T is the Viper predicate that encodes the type T. If x corresponds to a returned value of type &mut T, then $Q(x)$ expands to $P_T(\text{old[post]}(x.p))$.

Projection. A sequence of field accesses and dereferences. A single element of this sequence is called *projection step.*

Bibliography

[1] An alias-based formulation of the borrow checker. URL: https://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/.

[2] Rust bug tracker: Borrowed referent of a &T sometimes incorrectly allowed. URL: https://github.com/rust-lang/rust/issues/38899.

[3] The Rust compiler: ExprKind. URL: https://doc.rust-lang.org/nightly/nightly-rustc/rustc_hir/enum.ExprKind.html.

[4] The Rust programming language: Lifetimes. URL: https://doc.rust-lang.org/1.9.0/book/lifetimes.html.

[5] The Rust programming language: Ownership. URL: https://doc.rust-lang.org/1.9.0/book/ownership.html.

[6] The Rust programming language: References and Borrowing. URL: https://doc.rust-lang.org/1.9.0/book/references-and-borrowing.html.

[7] Underhanded Rust Contest: Results. URL: https://web.archive.org/web/20190511070359/http://blog.community.rs/underhanded/2017/09/27/underhanded-results.html.

[8] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 3, pages 147:1–147:30. ACM, 2019. doi:10.1145/3360573.

[9] Catalin Cimpanu. Microsoft: 70 percent of all security bugs are memory safety issues. URL: https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/.

[10] Synopsys Inc. The Heartbleed bug. URL: https://heartbleed.com/.

[11] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure

for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[12] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-Oriented Programming*, pages 148–172. Springer, 2009.

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):

Extended Support for Borrowing and Lifetimes in Prusti

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

Name(s):	**First name(s):**
Gorse	Lorenz

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the '<u>Citation etiquette</u>' 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	**Signature(s)**
Zürich, 13.10.2020	Gorse

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