YUSHUO XIAO

HYPERWANDS

PRACTICAL WORK PROJECT REPORT

Supervised by Thibault Dardinier

Contents

1 *Introduction* 3

2 *Preliminaries* 7

3 *Desirable Properties* 11

4 *Hyperwands for Rust* 15

5 *Going Beyond Rust* 24

6 *Conclusion & Future Work* 36

A *Discussion of Prusti* 38

Index 42

1 Introduction

Proving that a program is correct stands as one of the most important and compelling tasks in computer science and software development. This procedure, referred to as program verification, usually employs various program logics to reason about the behavior of a program. Hoare logic [1] is one of the earliest program logics trying to achieve this goal and has become the basis of many verification methods. Separation logic [2] is an extension of Hoare logic to reason about heap-manipulating programs and has gained increasing importance because of its effectiveness in expressing memory locations as resources. Separation logic features two main connectives, the separating conjunction $*$, and the separating implication $-\!\!*$ (also called the *magic wand*). A heap σ (*e. g.*, a partial function from heap locations to values) satisfying $A * B$ is defined as

We present formal definitions in Chapter 2.

$$\sigma \models A * B \quad \triangleq \quad \exists \sigma_A\ \sigma_B,\ \sigma_A \bullet \sigma_B = \sigma \wedge \sigma_A \models A \wedge \sigma_B \models B$$

where $\bullet$ is the join of two heaps. The definition of $*$ means that the heap σ can be divided into two disjoint heaps σ_A and σ_B which satisfy A and B, respectively. The separating implication (magic wand) operator $A -\!\!* B$, is defined as

$$\sigma \models A -\!\!* X \quad \triangleq \quad \forall \sigma_A,\ \sigma \perp \sigma_A \wedge \sigma_A \models A \Longrightarrow \sigma \bullet \sigma_A \models X,$$

where $\sigma \perp \sigma_A$ denotes the disjointness of σ and σ_A. This definition intuitively says that if a heap σ is extended with another disjoint heap σ_A that satisfies A, then the resultant heap $(\sigma \bullet \sigma_A)$ satisfies X. The magic wand has been shown to be useful for example when traversing recursive data structures (*e. g.*, a linked list), where it can be used to specify properties of partial data structures.

While magic wands are powerful tools to verify programs, they also have limitations that could be overcome with a more generalized version of magic wand, as we illustrate with the following use case.

Using magic wand to verify Rust programs. For Rust programs, magic wands can accurately describe the behavior of a function

that returns a mutable reference [3]. Consider the following Rust program that contains the definition of a struct `Point` and a function `foo` that takes a mutable reference of a `Point`, changes the y-coordinate to 5, and returns a mutable reference to the x-coordinate.

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

// ensures i32(result) ∗ (i32(result) −∗ Point(p) ∧ p.y = 5)
fn foo<'a>(p: &'a mut Point) -> &'a mut i32 {
    p.y = 5;
    &mut p.x
}
```

Then, consider the following Rust code that calls `foo`.

```
fn main() {
    let mut p = Point { x: 1, y: 2 };
    let x = foo(&mut p);  // p is borrowed into x.
    *x = 10;
    // At this point, we are done using the borrow x,
    // and p can be used freely from now.
    assert!(p.y == 5);  // Will this assertion succeed?
}
```

By the reference semantics of Rust, `p` will only be accessible after the last use of the reference `x`. So after we get the borrowed reference `x`, we can change the value of the x-coordinate of `p` but not of the y-coordinate, and the value of `p.y` will remain 5 until further assignments. In other words, we can exchange the validity of reference `x` for the permission of the `Point` object stored in `p` and the fact that `p.y` is equal to 5. This is exactly what a magic wand can express, as shown in the post-condition of the function `foo`. The wand is later applied (giving up the reference `x` and getting back the ownership of the `Point` object `p`) immediately after the borrow `x` expires. Prusti [3], a verifier for Rust, uses this encoding for functions that return a reference. It encodes the magic wand as a post-condition for the function and applies the wand when the returned reference expires.

However, for Rust programs that involve more complex borrowing situations, an ordinary magic wand is not expressive enough to prove fine-grained properties. Consider the following example.

```
fn foo<'a, 'b, 'c, 'x:'a+'b, 'y:'b+'c>
    (x: &'x mut Point, y: &'y mut Point)
    -> (&'a mut i32, &'b mut i32, &'c mut i32)
{
    // result.0 borrows from x.
    // result.2 borrows from y.
```

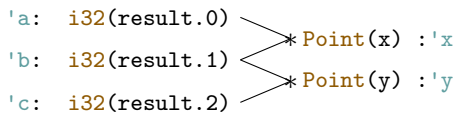
```
    // result.1 can borrow from both x and y.
}
```

The following code calls this function.

```
fn main() {
    let mut x = Point { x: 1, y: 2 };
    let mut y = Point { x: 3, y: 4 };
    let (a, b, c) = foo(&mut x, &mut y);
    *a = 1;
    *b = 2;
    // At this point, we are done using a and b.
    // x can be accessed from now on.
    // -- What if we want properties of x guaranteed
    // by foo at this point? --
    x.x = 3;
    x.y = 4;
    *c = 5;
    // At this point, we are done using both b and c.
    // y can be accessed from now on.
    y.x = 6;
    y.y = 7;
}
```

While the lifetime system in Rust is expressive enough to accept the above program, magic wands are not powerful enough to express properties like "as soon as the borrows `a` and `b` expire, some property X holds, *and*, as soon as the borrows `b` and `c` expire, some property Y holds". The fact that `b` appears in both statements prevents us from expressing the property with two magic wands, since separation logic resources are not duplicatable. The best we can do with a magic wand is something like "after the borrows `a`, `b`, and `c` all expire, both properties X and Y hold", which can only be applied after the statement `*c = 5`. This conservativeness makes assertions that should hold fail. This verification issue arises in Prusti as it uses the latter encoding. In this project, we want to explore a separation logic construct that is expressive enough to specify the former property.

Intuitively, the example above can be specified with a "wand" with richer structure.



We call this type of assertions *hyperwands*. Instead of having only one left-hand side and one right-hand side as in ordinary magic wands, a hyperwand is represented as a bipartite graph, with possibly multiple left-hand sides and right-hand sides and arbitrary connection between the two parts. On both sides are separation logic predicates. One

should be able to apply the hyperwand partially by only providing a subset of left-hand-side states. Specifically in this example, the hyperwand is specified as the postcondition of the function `foo`, and passed to the caller `main`. The `main` function will partially apply this wand as soon as one left-hand side expires, and when a right-hand side is "freed", it can reclaim the resources and start using them. This precisely captures the semantics of the lifetime annotations in Rust.

The capabilities of hyperwands look promising. Many aspects of the hyperwand were unknown to us at the beginning of this project, such as a definition of the hyperwand in terms of heaps. Furthermore, implementing the hyperwand in an automated program verifier will undergo some design choices, such as how the graph above should be represented in a textual language. In this practical work project, we explored some of these aspects and this report documents the progress that has been made.

Contributions. In this project, we made the following contributions.

- We motivate a new separation logic connective, the hyperwand, and give desired properties that hyperwands should have.
- We define hyperwands in several different ways with varied flexibility.
- We discuss the advantages and drawbacks of various definitions of hyperwands. Especially, we show that one of the definition is suitable for the verification of Rust programs.

Outline of the report. In Chapter 2, we give basic separation logic notions and hyperwand notations. In Chapter 3, we introduce the desired properties of hyperwands which serve as a guideline for finding the right definition. In Chapter 4, we give the first few definitions which are suitable for the verification of Rust programs. In Chapter 5, we try to give a more generalized definition of hyperwands. Chapter 6 concludes the report.

2 Preliminaries

This section introduces existing concepts in previous works such as separation algebra and abstract separation logic, which will be useful in later chapters to give formal general definitions of hyperwands. This section also defines notations that will be used later. The definitions on separation algebra and abstract separation logic are mainly taken from Calcagno et al. [4].

2.1 Separation Algebra

Definition 2.1 (*separation algebra*)**.** A separation algebra is a partial commutative monoid (PCM)[1] $(\Sigma, \bullet, u)$.

The definition of a separation algebra is given in an abstract sense. We explain the concept with a concrete model—the heap model. In the definition, Σ is the set of all possible states[2]. Two disjoint states can be added together, with the $\bullet$ operation. u is the unit of the PCM, *i. e.* the empty heap.

From the definition of a separation algebra, some related notations are induced.

Notation 1 ($\perp$)**.** $a \perp b \iff a \bullet b$ is defined. We say that a and b are *compatible* if $a \perp b$.

Notation 2 ($\preceq$)**.** $a \preceq b \iff \exists c,\ a \bullet c = b$. We say that a is a *substate* of b if $a \preceq b$.

Notation 3 ($\succeq$)**.** $a \succeq b \iff b \preceq a$. We say that a is a *superstate* of b if $a \succeq b$.

Notation 4 ($\ominus$)**.** If $a \preceq b$, the *difference* between b and a, $b \ominus a$, is defined by the state[3] x that satisfies $a \bullet x = b$.

Notation 5 ($\bigoplus$)**.**

$$\bigoplus_{i=1}^{n} a_i = a_1 \bullet a_2 \bullet \cdots \bullet a_n.$$

If any of the addition in the summation sequence is undefined, the final result is undefined.

[1] See Calcagno et al. [4] for the formal definition of a PCM. We explain the intuition of a PCM below.

[2] The word "*state*" here refers to any concrete model for memory. *e. g.* RAM model, fractional permission model, etc. From now on, when ever we use the word "state", we refer to some element in the abstract PCM (Σ). We refrain from using terms such as "heap state" and "heap" to avoid being concrete about the underlying model.

[3] Such a state is unique because of cancellativity.

Notation 6 (*split*)**.** Let $\overline{a_i}$ be a finite list of n states. If

$$\bigoplus_{i=1}^{n} a_i = a,$$

then we say that $\overline{a_i}$ *splits* the state a.

Predicates are subsets of the set of all possible states Σ.

Definition 2.2 (*predicate*)**.** A predicate P is an element from the powerset $\mathcal{P}(\Sigma)$, *i. e.*, $P \in \mathcal{P}(\Sigma)$.

Notation 7. We use italic uppercase Latin letters (A, B, C, P, Q, X, Y, Z, ...) to represent predicates appearing in magic wands and hyperwands, and use lowercase Latin letters a, b, c, ... to denote states. A lowercase letter is used to represent a state that satisfies the predicate written with the uppercase counterpart. For example, a is a state that satisfies A, etc.

Notation 8. A predicate is a set of states. So we can use $\sigma \in P$ to denote the state σ is in the set P. For clarity, we also use the "$\models$" symbol to denote the same concept, *i. e.*

$$\sigma \models P \iff \sigma \in P.$$

2.2 Abstract Separation Logic

In our development of hyperwands, we do not use any specific memory model, rather we use an abstract model to describe resources. Magic (hyper-) wands talk about resources, so it is a natural choice to use separation algebra to describe resources. In this report, the left-hand sides and right-hand sides of a hyperwand are predicates over some state model $(\Sigma, \bullet, u)$.

Before starting exploring hyperwands, we first introduce the basic standard separation logic connectives—separating conjunction and separating implication (*i. e.*, magic wands).

Definition 2.3 (*separating conjunction*)**.**

$$\sigma \models A * B \iff \exists\, a\ b,\ a \bullet b = \sigma, a \models A, b \models B.$$

Remark 2.1. An alternative way of describing Definition 2.3 is

$$A * B = \{a \bullet b \mid a \in A, b \in B\}.$$

We refrain from using it since it could be cumbersome and hard to read.

Definition 2.4.

$$\mathop{\Large *}_{1\le i\le n} A_i \quad \Leftrightarrow \quad A_1 * A_2 * \cdots * A_n.$$

In this report, the operator $\mathop{\Large *}$ must range over a *finite* set of predicates.

Definition 2.5 (*separating implication*)**.**

$$\sigma \models A \mathrel{-\!\!*} X \iff \forall a,\ a \perp \sigma \wedge a \models A \implies a \bullet \sigma \models X.$$

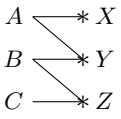
Definition 2.6 (*footprint*)**.** If $\sigma \models A \mathrel{-\!\!*} X$, σ is called a *footprint* of the wand $A \mathrel{-\!\!*} X$. The same terminology applies for hyperwands.

Notation 9. The precedence of $*$ is higher than that of $\mathrel{-\!\!*}$. *i. e.*

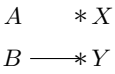
$$A \mathrel{-\!\!*} X * Y \iff A \mathrel{-\!\!*} (X * Y).$$

2.3 Notations for Hyperwands

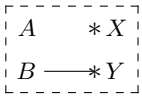
Hyperwands are *bipartite graphs*. Left-hand sides and right-hand sides are the two parts of the bipartite graphs. Each right-hand side is written with a star next to it. Below is an example of a hyperwand.



Sometimes, a right-hand side may be dangling, *i. e.*, it is not connected to any left-hand side. If the right-hand side is still written with a star, it is still consider part of the hyperwand and not a separate part, as shown below.



When necessary, we also use a dashed frame box to indicate the boundary of the hyperwand, as shown below.



Notation 10. In this report, left-hand sides are represented by *upper-case italic* Latin letters: $A, B, C, \cdots$, and right-hand sides are represented by $X, Y, Z, \cdots$.

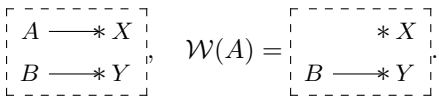
$\mathcal{W}, \mathcal{V}, \cdots$ are used for shorthand for hyperwands.

Notation 11. If not mentioned otherwise, hyperwands in this report have n left-hand sides and m right-hand sides. The index variable i ranges over $1 \le i \le n$, and j ranges over $1 \le j \le m$.

Notation 12. If A is a left-hand side, $\mathrm{conn}(A)$ represents the set of right-hand sides that is connected to A with an edge. If X is a right-hand side, $\mathrm{conn}(X)$ represents the set of left-hand sides that is connected to X with an edge.

Hyperwands are meant to be applied one left-hand side at a time. When a hyperwand is applied with a left-hand side, that left-hand side and all edges connected to it are removed; dangling right-hand sides are *not* removed or separated[4].

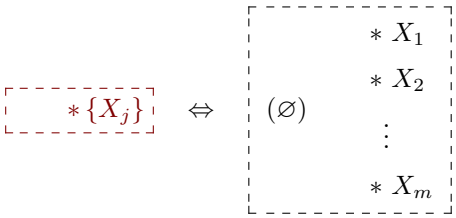
Notation 13 (*applying hyperwand*)**.** We write $\mathcal{W}(A)$ for the hyperwand $\mathcal{W}$ with one left-hand side A applied. For example,



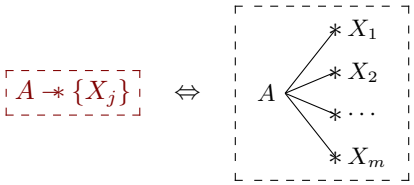
[4] It is possible that dangling right-hand sides can be separated with separating conjunction under some definitions. However, it is considered as a property of dangling right-hand sides in that specific definition rather than part of the application.

Notation 14. Sometimes, two or more left-hand sides are identical. In that case, the left-hand side predicate A in $\mathcal{W}(A)$ still refers to one *single* left-hand side, rather than all left-hand sides which are identical to A. One can assume every left-hand side and every right-hand side is implicitly labeled. Here, the notation is slightly informal, but it does not negatively affect understanding of the theorems and proofs in this report.

Notation 15.

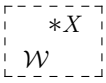


Notation 16.



Notation 17. The notation $\{A_i\} \cdots\!* \{X_j\}$ represents a hyperwand with left-hand side set $\{A_i\}$ and right-hand side set $\{X_j\}$, and *arbitrary* connections.

Notation 18 (*dangling right-hand side*)**.**



represents a single hyperwand, which is formed by adding a dangling right-hand side to the hyperwand $\mathcal{W}$.

Notation 19 (*separate hyperwands*)**.**

represents a single hyperwand, which is formed by taking the disjoint union of the left-hand-side set and the right-hand-side set of $\mathcal{V}$ and $\mathcal{W}$, respectively, without adding any edges between them.

3 Desirable Properties

As motivated in Chapter 1, a hyperwand should satisfy some properties to make them useful in practice. In this section, we state more formally what these properties are. Not all properties are equally important. One could still package and apply useful hyperwands with only a subset of these properties. Below, we state every property that is useful in some way, including why they could be useful and why they could be potentially irrelevant.

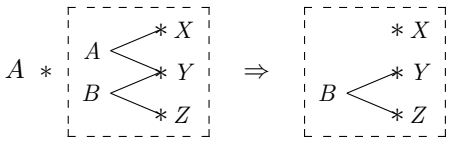
3.1 Inductive Pattern

Definition 3.1. A definition is said to satisfy the *inductive pattern*, if

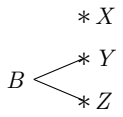
$$A * \mathcal{W} \Rightarrow \mathcal{W}(A),$$

where A is a left-hand side of $\mathcal{W}$.

All definitions should satisfy the inductive pattern since it is the "skeleton" of all useful properties, in the sense that other properties are stated on a partially applied hyperwand. For example, we expect the following separation logic formula to hold.



It is particularly important to note that the shape



represents a hyperwand by itself. The fact that X is not connected to any left-hand side does not make it separate in any sense.

3.2 Degenerate Cases

We want hyperwands to generalize magic wands. When a hyperwand is in certain shapes, it should be equivalent to a magic wand, or merely separating conjunction. However, it is not clear which cases should be equivalent to what magic wands and there might not be a single best solution.

First, if a hyperwand has no left-hand sides, it should be equivalent to a separation conjunction of all right-hand sides.

Definition 3.2. A hyperwand definition satisfies the *0–n equivalence*, if the following holds.

$$\boxed{\quad *_i \{X_i\} \quad} \Leftrightarrow \mathop{\Large *}_i X_i$$

If a hyperwand has exactly one left-hand side and one right-hand side, it should be equivalent to a normal magic wand.

Definition 3.3. A hyperwand definition satisfies the *1–1 equivalence*, if the following holds.

$$\boxed{A \mathrel{-\!\!*} X} \Leftrightarrow A \mathrel{-\!\!*} X$$

These two degenerate cases are clear. In other words, every hyperwand definition should satisfy these two properties. However, the situation is less clear for the following two cases.

Definition 3.4. A hyperwand definition satisfies the *1–n equivalence*, if the following holds.

$$\boxed{A \mathrel{-\!\!*_i} \{X_i\}} \Leftrightarrow A \mathrel{-\!\!*} \mathop{\Large *}_i X_i$$

Note that this equivalence might not be strictly necessary, because it can be expressed with a normal magic wand (the right-hand side is the separation conjunction of all X_i). However, we will also describe in the next two chapters how it affects the expressiveness of hyperwands.

3.3 Full Release

A hyperwand that only satisfies inductive pattern is useless because one gets nothing but another (useless) hyperwand by applying the hyperwand with a left-hand side. A basic property every useful hyperwand must have is *full release*.

Definition 3.5. A hyperwand definition is said to satisfy *full release*, iff

$$\boxed{\begin{array}{c} *X \\ \mathcal{W} \end{array}} \Rightarrow X * \mathcal{W}.$$

Note that full release implies 0–n equivalence.

This property is inspired by Rust and is one of the most important motivations of hyperwands. The reason we need hyperwands is that we want to get back some right-hand sides after only partially applying the hyperwand. If a right-hand side is "fully released" from all left-hand sides, that right-hand side predicate is expected to be "freed". Full release is necessary for hyperwands to be useful in practical scenarios.

3.4 Wand Separation

Another important property to have is *wand separation*, which is defined below.

Definition 3.6. A hyperwand definition is said to satisfy *wand separation*, iff

$$\boxed{\begin{array}{c}\mathcal{V}\\ \mathcal{W}\end{array}} \Rightarrow \mathcal{V} * \mathcal{W}.$$

Intuitively, wand separation means if a hyperwand consists of two disconnected parts, they can be split with a separating conjunction. It could be useful, for example, by splitting a disconnected hyperwand and giving the resources represented by two separate wands to two different threads (although a concrete use case that requires this property is yet to be found).

Followed directly by the definitions, wand separation is stronger than full release. In other words, any definition that satisfies wand separation satisfies full release.

While the property is good to have, it is not useful for the Rust reborrow scenario, because a reborrow never expires inside another function, albeit we pass the borrow to that function. Consider the abstract Rust code below.

```
1  let (x, y) = (some data)
2  let (a, b, c) = f(&mut x, &mut y)
   a ─┐
   b ─┼─* x
   c ─┘─* y   (the post-condition of the function)
3  [b expires]
   a ──* x
   c ──* y    (the new hyperwand after b has been applied)
4  let p = borrow(a)
5  [a expires here]
   (note that it is not possible that a expires inside the function
   borrow)
6  let q = borrow(c)
7  [c expires here]
   (similarly, it is not possible that c expires inside the function
   borrow)
```

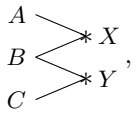
In this code snippet, the function `f` borrows `x` and `y` as mutable references, and returns three references `a`, `b`, and `c`, which creates a reborrow relationship represented by the hyperwand after line 2. *These three references must expire in the same scope the code snippet is in.* It is neither possible to pass it to another function and make it expire there, nor to leak it to outside the current scope[1].

A possible use case of wand separation is that after a hyperwand is partially applied, we want to pass two separate remaining parts to two

[1] As we can see in the following non-compiling Rust code snippet, it is not possible to leak a reference to the outside of the scope that owns the value.

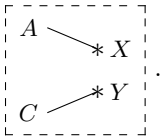
```
let r: &mut i32
{
    let x = 1
    r = &mut x
}
*r = 2
```

concurrent threads and they need to apply the wands individually. For example, if we hold the wand



,

and apply the second left-hand side, then we hold a new hyperwand



.

We might want to separate $A \mathrel{-\!\!*} X$ and $C \mathrel{-\!\!*} Y$ and pass them to two concurrent threads, which can then use resources from A and C and then apply them to get X and Y, respectively.

4 Hyperwands for Rust

Rust verification is the main motivation for this project. The verification of a type of function in Rust, called reborrow function, has particular interest in the use of wands.

We have already shown how normal magic wands can be used to verify simple reborrow functions in Rust and also why hyperwands are desirable to verify more complex reborrow functions in Chapter 1. In this chapter, we describe the restrictive subset of Rust that we consider, propose a working definition for this subset and then extend the definition to (hopefully) cover the complete Rust language.

4.1 A Restrictive Subset of Rust

In this section, we consider the subset of Rust where `struct`s do not have references as fields and build a hyperwand definition for it. This is the same restriction as in Prusti [3]. By imposing this restriction, we do not allow a `struct` to borrow from two values *simultaneously*[1]. Therefore, every left-hand side of the hyperwand packaged by the reborrow function connects to exactly one right-hand side on every individual execution, which makes hyperwands very restrictive (but enough for the restrictive Rust subset). This property is better understood with the following example.

[1] It is still possible to borrow from different values on different executions, as we will show later.

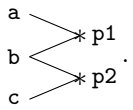
```
struct Point {
    x: i32,
    y: i32
}

fn foo<'a, 'b, 'c, 'x:'a+'b, 'y:'b+'c>
    (p1: &'x mut Point, p2: &'y mut Point)
    -> (&'a mut i32, &'b mut i32, &'c mut i32)
{
    (&mut p1.x,
     if rng.gen::<bool>() { &mut p1.y } else { &mut p2.y },
     &mut p2.x)
}
```

The function `foo` takes two mutable references of `Point` as argu-

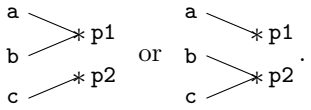
ments, and returns three mutable references of `i32`. The first and third returned references are just `p1.x` and `p2.y`, respectively. The second return value, depending on the value returned by a random number generator, is either `p1.y` or `p2.x`. If we consider all possible execution traces, the second returned reference may borrow from either arguments. However, if we only consider one concrete execution, it can only borrow from exactly one argument.

In this particular example, we might want to package the hyperwand



.

However, considering only a single execution, the actual shape of the hyperwand is either



.

In the Rust setting (or in general every setting that satisfies wand separation), they are equivalent to

$$^{(a)}\ (\mathtt{a} * \mathtt{b} \mathrel{-\!\!*} \mathtt{p1}) * (\mathtt{c} \mathrel{-\!\!*} \mathtt{p2}) \quad \text{and} \quad ^{(b)}\ (\mathtt{a} \mathrel{-\!\!*} \mathtt{p1}) * (\mathtt{b} * \mathtt{c} \mathrel{-\!\!*} \mathtt{p2}),$$

respectively. A workaround to express the post-condition if we do not have hyperwands is to make `foo` return an additional Boolean value, indicating if the post-condition is (a) or (b). By this way, we support the function `foo` with existing Viper facilities. However, this method violates information hiding enforced by the Rust lifetime annotation. To abstract this information away, we want to use a hyperwand to overapproximate all possible shapes derived from the function body. We thus propose the following definition of the hyperwand for Rust.

Definition 4.1 (*fixed-fixed hyperwand*)**.** $\sigma \models \{A_i\} \cdots\!\!* \{X_j\}$ iff $\exists\, \overline{\sigma_j}$ splits σ, $\exists\, \overline{g_i \in \mathrm{conn}(A_i)}$, such that

$$\forall\, \overline{a_i},\ \forall\, j,\ (\forall\, i,\ i \in \mathrm{conn}(X_j) \implies a_i \models A_i) \implies$$
$$\sigma_j \bullet \bigoplus_{g_i = j} a_i \text{ is defined} \implies \sigma_j \bullet \bigoplus_{g_i = j} a_i \models X_j.$$

While seemingly complex, this definition builds on a simple and intuitive idea. This definition says that, to package a hyperwand under the *fixed-fixed definition*, one needs to assign to each left-hand side a right-hand side (the $\overline{g_i}$ array in the definition), to which the state supplied for the left-hand side is given. Even if a left-hand side might be connected to multiple right-hand sides, the left-hand-side states do not split for a given valid footprint (a_i is never split in this definition, which is not the case in the next chapter).

For sake of simplicity, we use the name of the variable (*e. g.* `a`, `b`, `p1`) to denote the predicate of the corresponding resource.

Note that we require σ_j only to be compatible with the left-hand-side states assigned to X_j. This is important to make full release and separation hold for this definition. We also explored another weaker definition. Specifically, we require the left-hand-side states $\overline{a_i}$ to be compatible with the whole footprint σ. This restriction makes the definition not satisfy full release and wand separation. We provide a more detailed discussion in Chapter 5, where we also formalize this weaker version, but for a definition called *lhs-self*. The thing to notice is that there also exists a weaker *fixed-fixed* definition (or all definition based on split of the states in general) which we left out here.

We now see how this definition works in action using the Rust example above. Suppose the first branch is taken. Then the footprint of the hyperwand σ is the borrow of `p2.x`. There exists a split of σ:

$$\exists\, \overline{\sigma_j} : \quad \sigma_1 = \varnothing,\ \sigma_2 = \{\texttt{p2.x}\}.$$

Also, we can assign a right-hand side to each left-hand side.

$$\exists\, \overline{g_i \in \mathrm{conn}(A_i)} : \quad g_1 = 1,\ g_2 = 1,\ g_3 = 2.$$

Then, we need to show that given any compatible $\overline{a_i}$, for each right-hand side X_j, if all connected left-hand sides are satisfied by corresponding a_i, then the sum of all assigned a_i plus the split of the footprint σ_j satisfies X_j. The assignment relation and the footprint split are demonstrated in Figure 4.1. In this example, the property above is trivial to show.

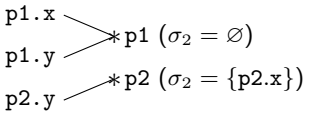


Figure 4.1: The assignment relation when the first branch is taken.

In fact, if we do not allow references to appear in `struct` declarations, this definition is sufficient to capture all Rust reborrow functions. Indeed, a Rust type can be seen as a tree, and a reborrow is merely a subtree of the lender type. The packaged magic wand is very similar to the magic wands used to express ownership of partial data structures.

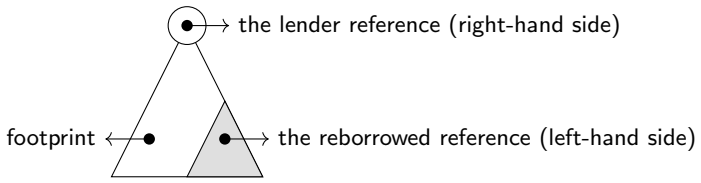


Figure 4.2: Demonstration of a reborrow relation in Rust.

For a Rust function that takes references as arguments and returns (reborrowed) references, the arguments are represented by the entire tree in Figure 4.2 and the returned reference is colored in gray. Each returned reference may potentially borrow from different "trees" but for any execution trace it can only borrow from one "tree" (the situation illustrated in Figure 4.3 cannot occur). The right-hand sides of the hyperwand specified by a reborrow function are the "trees", and the left-hand sides are the gray subtrees. The footprint of the hyperwand is just the part which is not covered in gray. It satisfies Definition 4.1 because the split of the footprint is fixed, and we can assign each gray area to the "tree" where it is borrowed from.

4.1.1 Translating reborrow functions into Viper

The support of hyperwands has not been implemented in Viper yet. However, to support only the Rust use case, we can make a

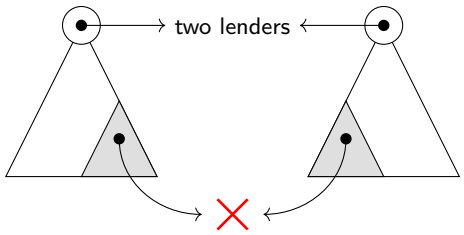


Figure 4.3: The situation that is not possible in the subset of Rust in question. One cannot create a reference out of two borrowed values simultaneously.

lightweight extension to Viper using existing magic wand facilities. We notice that in the Rust use case, the shape of the reborrow relation is fixed under the same code path. Therefore, we can specify which (normal) magic wands to package on *every* possible code path, each of which must be a subshape[2] of the final hyperwand. In this hypothetical extension of Viper, we could do something like the following for the function `foo`[3].

[2] A subshape of a hyperwand is just the hyperwand where we have removed some edges.

[3] We modify the code to an equivalent version so that every possible code path is clear.

```
1   fn foo<'a, 'b, 'c, 'x:'a+'b, 'y:'b+'c>
        (p1: &'x mut Point, p2: &'y mut Point)
        -> (&'a mut i32, &'b mut i32, &'c mut i32)
        [ post-condition: a, b, c ⟶ p1, p2 ]
2   {
3       if rng.gen::<bool>() {
4           (&mut p1.x, &mut p1.y, &mut p2.y)
            [ package the hyperwand: a, b ⟶ p1; b, c ⟶ p2 ]
5       }
6       else {
7           (&mut p1.x, &mut p2.x, &mut p2.y)
            [ package the hyperwand: a ⟶ p1; b, c ⟶ p2 ]
8       }
        (Every path packages a subshape of the hyperwand. The original
        hyperwand is packaged.)
9   }
```

In this Rust function (or the Viper equivalence), the user first specifies the hyperwand they want to package as a post-condition. Then, on every path, a subshape of the specified hyperwand must be packaged. Since on each branch, the packaged hyperwand is just a conjunction of multiple normal magic wands, we could reuse Viper facilities for magic wands and package normal magic wands instead. At the end

of the function, the "package" of the hyperwand can be seen as an "overapproximation" of all magic wands packaged on each path, and no additional work needs to be done to claim the post-condition. The overapproximation is correct because we can always add edges to a hyperwand and the footprint still remains valid under *fixed-fixed* definition.

This package method shows the essence of the use of hyperwand in Rust—we deliberately forget some information to abstract the information away. As a result, we cannot express the property

```
p1.y == b || p2.x == b
```

after all three left-hand sides are applied. This is a fundamental limit of our hyperwand model. Indeed, every right-hand side in our hyperwand is a predicate, whose validity must be verified only by a single state. Appendix A.2 gives another example explaining why this kind of specification might be desirable.

4.1.2 Properties of fixed-fixed Definition

The fixed-fixed definition, being restrictive, enjoys several useful and important properties.

Theorem 4.1. *Fixed-fixed hyperwands satisfy the inductive pattern.*

Theorem 4.2. *Fixed-fixed hyperwands satisfy full release.*

Theorem 4.3. *Fixed-fixed hyperwands satisfy wand separation.*

These results have not been formally mechanized yet. However, we are fairly confident about these claims and we have a proof on paper for a more permissive definition (Section 4.2).

4.1.3 Automation

Our final discussion on this definition is about the difficulty of automation. The ultimate goal of automation includes automatically inferring a proper footprint for the hyperwand. For now, we only consider checking if a *given* footprint satisfies the hyperwand. It turns out that checking this definition is straightforward. If the split of the footprint and the assignments of left-hand sides ($\overline{g_i}$) are all given, it is trivial to check the satisfiability of each right-hand side. The fact that it is easy to automate makes the fixed-fixed definition potentially useful.

Besides, in the Rust scenario, we are even able to implicitly infer the footprint automatically, using the method introduced in Section 4.1.1 to package normal magic wands instead.

4.2 Complete Rust

In the previous section we provided a hyperwand solution for reborrow functions in a restrictive subset of Rust. Specifically, we do not allow references to appear in `struct` definitions. This restriction makes sure every returned reference borrows from exactly one argument in any reborrow function. However, this property does not hold for all Rust programs. In this section, we try to lift this restriction and support a more complete subset of the Rust language.

In Definition 4.1, we state that each left-hand side must be assigned to a right-hand side, and the state provided to the left-hand side is "allocated" entirely to the assigned right-hand side. In this definition, the state given to satisfy one left-hand side will not be truly split even if the left-hand side is connected to more than one right-hand side. Instead, it represents possible choices the left-hand side is assigned to. This definition does not faithfully reflect the semantics of Rust if we allow references in `struct`. In the simplest form, a `struct` can store two mutable references to `i32`, and we define a function where two mutable references are taken as arguments and one *owned* `struct` is returned, storing both references, as shown in the following code snippet.

```
1  struct Data<'a>
2  {
3      x: &'a mut i32,
4      y: &'a mut i32,
5  }
6
7  fn pack_mut<'a> (x: &mut i32, y: &mut i32) -> Data<'a>
8  {
9      Data { x: x, y: y }
10 }
```

The function `pack_mut` specifies a hyperwand shape shown in Figure 4.4. The only left-hand side, represents the `struct` that is going to expire when the lifetime `'a` ends. The left-hand side contains the resources (ownership) of both arguments, and each right-hand side represent one input argument. Definition 4.1 is not sufficient for this use case anymore, since the state provided to the left-hand- side needs to be split and allocated to each right-hand side.

Figure 4.4: The hyperwand shape of the Rust function `pack_mut`.

To capture this scenario, we introduce another more flexible definition of hyperwand.

Definition 4.2 (*fixed-self hyperwand*)**.** $\sigma \models \{A_i\} \cdots\!* \{X_j\}$ iff. $\exists\, \overline{\sigma_j}$ splits σ, $\exists$ *split functions* $\overline{f_i}$, such that

1. $\bigoplus_j f_i(a, j) = a$;

2. $\forall\, j,\ j \notin \mathrm{conn}(A_i) \implies f_i(a, j) = u$;

Recall that u is the unit of the PCM.

3. $\forall\, \overline{a_i},\ \forall\, j,\ (\forall\, i,\ i \in \mathrm{conn}(X_j) \implies a_i \models A_i) \implies$
$\sigma_j \bullet \bigoplus_i f_i(a_i, j)$ is defined $\implies \sigma_j \bullet \bigoplus_i f_i(a_i, j) \models X_j$.

Intuitively, the difference between Definition 4.1 and Definition 4.2 is simply that the latter allows splitting of left-hand states. To satisfy the definition, one needs to come up with a split function for each left-hand side, and when the left-hand side is applied, the function is used to determine which part of the state supplied is distributed to connected right-hand sides. For the i-th left-hand side A_i, f_i is its split function. When the i-th left-hand side is applied with a state a_i, the state is supplied to f_i for each right-hand side, and we obtain a series of states: $f_i(a_i, 1)$, $f_i(a_i, 2)$, $\ldots$, $f_i(a_i, m)$. Property 2 states that among all these states, if some right-hand side X_j is not connected to the left-hand side A_i, then the split must be empty state (the unit u).

We believe that Definition 4.2 is sufficient to capture all Rust reborrow functions in theory. However, in practice, it is not clear how we should encode the Rust references as predicates (the left-hand sides and right-hand sides), especially if we allow references fields in `struct` definitions. See Appendix A for a discussion about the current limitation of Prusti on this issue.

4.2.1 Properties

Definition 4.2, despite being more flexible than Definition 4.1, still enjoys the same properties.

Theorem 4.4. *Fixed-self hyperwands satisfy the inductive pattern.*

Theorem 4.5. *Fixed-self hyperwands satisfy full release.*

When proving *full release* for some definition, it is very important that one not only verifies that the dangling right-hand side is satisfied by some state, but also verifies that the remaining part satisfy the rest of the hyperwand. Forgetting to verify this is one of the most common reasons in this project that a wrong conclusion is drawn.

If a right-hand side is dangling, then no left-hand sides can contribute to it, and its share of the split of the footprint satisfies it. The remaining part of the hyperwand, is satisfied by the original footprint σ, minus the part of the footprint that the dangling right-hand side takes.

Theorem 4.6. *Fixed-self hyperwands satisfy wand separation.*

Proof. Suppose we have the following hyperwand (denoted as $\mathcal{W}$) under the fixed-self definition.

$$\sigma \models \left[\begin{array}{l} \{A_i\} \cdots\!* \{X_j\} \\ \{B_i\} \cdots\!* \{Y_j\} \end{array}\right]$$

Then we have the split of the footprint σ_* and the split functions for left-hand sides f_*. The split of the footprint naturally sorts into two

groups. The first group contains the split of the footprint for X_j, and the second group contains the split of the footprint for Y_j. We claim that the sum of the first group is a valid footprint of the hyperwand $\{A_i\} \dashrightarrow\!\!* \{X_j\}$, and the sum of the second group is a valid footprint of the hyperwand $\{B_i\} \dashrightarrow\!\!* \{Y_j\}$. Due to symmetry, we only prove the first claim.

Having the footprint for the hyperwand $\{A_i\} \dashrightarrow\!\!* \{X_j\}$, now we need to give split functions for the left-hand sides. The split functions are identical to those of the hyperwand $\mathcal{W}$. Now we prove the validity of the footprint and the split functions.

First, it is clear that point 1 and 2 in Definition 4.2 hold. Now we prove point 3. Fix a list of states $\overline{a_i}$ provided to $\overline{A_i}$. Point 3 is an implication with two antecedents

$$\forall\, i,\ i \in \mathrm{conn}(X_j) \implies a_i \models A_i,\ \text{and}$$
$$\sigma_j \bullet \bigoplus_i f_i(a_i, j) \text{ is defined}$$

If they hold for the new footprint and the split functions we constructed for the hyperwand $\{A_i\} \dashrightarrow\!\!* \{X_j\}$, they also hold for the original footprint σ and the split functions for $\mathcal{W}$, since the split functions are exactly the same. Hence, we can derive the consequence using the definition on $\mathcal{W}$, getting

$$\sigma_j \bullet \bigoplus_i f_i(a_i, j) \models_{\text{fixed-self}} \{A_i\} \dashrightarrow\!\!* \{X_j\}.$$

□

Theorem 4.6 also implies that fixed-self hyperwands satisfy full release.

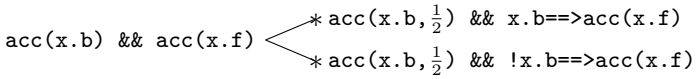
4.2.2 Automation

Checking whether a given state is a valid footprint of a hyperwand under Definition 4.2 in the most general setting is not trivial. It is not clear how to express a split function in an automated verifier. However, in some use cases of Rust, if we are able to precisely describe the resources on the left-hand sides, it is not necessarily difficult to make the split. For example, in the hyperwand in Figure 4.4, the resources described by the left-hand side are precisely the integers `x` and `y`, which correspond to the two right-hand sides. In this case, there is only one state that satisfies the left-hand side, and the split is fixed. Expressing this split is a much easier task to do.

Another idea[4] is to express the split with predicates. For each left-hand side, we add a list of predicates that add up to that left-hand

[4] Felix Wolf proposed this idea in the final presentation of this project.

side predicate, and each of the predicates describes a split from this left-hand side to a right-hand side it is connected to. Consider the following hyperwand.



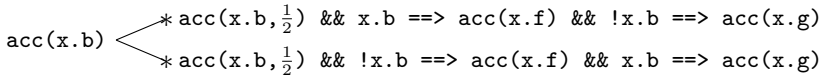
under the empty footprint. We can express the split using the two right-hand sides. So, the check becomes verifying if the separation conjunction of the two right-hand sides is equivalent to the left-hand side. This example is a special case where the footprint is empty and the wand is of 1-n shape. How to express the split using this approach in more complex hyperwands still needs to be explored.

5 Going Beyond Rust

In the previous chapter, we presented a hyperwand definition that potentially covers all reborrow scenarios in Rust in theory. However, as we will see below, the definition surprisingly does not satisfy some properties that one might expect to satisfy. On the other hand, it also remains unclear if these properties should hold. Also, there exist plenty of hyperwand examples that satisfy the inductive pattern and wand separation, but are not covered by previous definitions. These wands are also potentially useful. In this chapter, we will have a look at several more definitions of hyperwand.

5.1 Splitting the Footprint Depending on Left-Hand States

Definition 4.1 and 4.2 are very restrictive. In Definition 4.1, each left-hand side needs to be assigned to a single right-hand side, and does not allow true split of states. In both Definition 4.1 and 4.2, the split of the footprint is fixed before applying any left-hand-side states. This feature makes these two definitions violate *1–n equivalence*, which we demonstrate with the following example.



To satisfy the 1–n equivalence property, this hyperwand should be equivalent to $A \mathrel{-\!\!*} X * Y$.

For simplicity, we denote the left-hand side as A, and the two right-hand sides as X and Y, respectively.

From the definition of separating conjunction, we can easily see that the wand $A \mathrel{-\!\!*} X * Y$ is satisfied by the footprint `acc(x.f) && acc(x.g)`. However, the split of the footprint depends on the state supplied for the left-hand side. Indeed, if the value of `x.b` is `true`, X will get the permission to `x.f` and Y will get the permission to `x.g`. On the other hand, if the value of `x.b` is `false`, then X will get the permission to `x.g` and Y will get the permission to `x.f` instead. As a result, we are unable to split the footprint before we know the left-hand state. Therefore, the hyperwand $A \mathrel{-\!\!\!-\!\!*} \{X, Y\}$ does not hold for this footprint under Definition 4.1 and 4.2.

Should the normal magic wand $A \mathrel{-\!\!*} X * Y$ and the hyperwand $A \mathrel{-\!\!\!-\!\!*}$

$\{X, Y\}$ be equivalent? In other words, should 1–n equivalence hold for the hyperwand definition? That is a design choice we have to make. On the one hand, one could argue that letting them be equivalent demonstrates a nice generalization of normal magic wand. On the other hand, why would we need another construct to express the same thing when $A \mathrel{-\!\!*} X * Y$ can already be expressed with a normal magic wand?

Nevertheless, allowing the split of the footprint to depend on left-hand-side states makes the definition strictly more expressive since the split can just ignore the left-hand-side information and make a fixed split as in previous definitions. In this section, we are going to explore a potential definition for hyperwands that satisfies the 1–n equivalence, and discuss problems with this new definition[1].

[1] We mentioned in Section 4.1 that for every definition we give, there exists a weaker definition that does not satisfy the good properties. We are going to see how the definition is formulated and why it is the case

Definition 5.1 (*lhs(weak)–self (restrictive compatibility) hyperwand*)**.** $\sigma \models \{A_i\} \cdots\!\!* \{X_j\}$ iff. $\exists$ *footprint split functions* $\overline{\sigma_j}$, $\exists$ *split functions* $\overline{f_i}$, such that

1. $\forall \overline{a_i}$ compatible with σ, $\bigoplus_j \sigma_j(\overline{a_i}) \preceq \sigma \wedge$

$$\left((\forall\, i,\ a_i \models A_i) \implies \bigoplus_j \sigma_j(\overline{a_i}) = \sigma \right);$$

2. $i \notin \mathrm{conn}(X_j) \implies \sigma_j(\cdot)$ is constant with respect to a_i;

3. $\bigoplus_j f_i(a, j) = a$;

4. $\forall\, j,\ j \notin \mathrm{conn}(A_i) \implies f_i(a, j) = u$;

5. $\forall \overline{a_i}$ compatible with σ, $\forall\, j$,

$$(\forall\, i,\ i \in \mathrm{conn}(X_j) \implies a_i \models A_i) \implies \sigma_j(\overline{a_i}) \bullet \bigoplus_i f_i(a_i, j) \models X_j.$$

Intuitively, the definition above says that a footprint σ satisfies a hyperwand, if and only if there exist footprint split functions for σ, such that $^{(i)}$ σ_j depends on only the left-hand sides that X_j is connected to (point 2), $^{(ii)}$ the split is always valid provided that the given $\overline{a_i}$ is compatible with σ (point 1), and $^{(iii)}$ the split satisfies similar properties as in Definition 4.2 (point 3, 4, and 5).

Point 5 is slightly different than in previous definitions. In this definition, we require the left-hand-side states $\overline{a_i}$ to be compatible with σ, which is a stronger requirement. Definitions that have this pattern have *restrictive compatibility* in their names.

5.1.1 Properties

1–n equivalence

Theorem 5.1.

$$\sigma \models_{\text{lhs(weak)}-\text{self}^{\text{R}}} A \mathrel{-\!\!*} \{X_j\} \implies \sigma \models A \mathrel{-\!\!*} \mathop{\Large *}_j X_j.$$

We write lhs(weak)–self^R as the shorthand for the name "lhs(weak)-self (restrictive compatibility)". Similar shorthands also apply later.

Proof. Fix a, such that $a \perp \sigma$ and $a \models A$. By properties 1 and 3, there exist split functions σ_* and f, such that

$$\bigoplus_j \sigma_j(a) = \sigma \quad \text{and} \quad \bigoplus_j f(a,j) = a. \tag{5.1}$$

Let $x_j = \sigma_j(a) \bullet f(a,j)$. From (5.1), it follows that $\bigoplus_j x_j = \sigma \bullet a$. Finally, from property 5,

$$x_j = \sigma_j(a) \bullet f(a,j) \models X_j.$$

Hence,

$$\sigma \bullet a \models \mathop{\Large *}_j X_j.$$

□

The converse of Theorem 5.1

$$\sigma \models A \mathrel{-\!\!*} \mathop{\Large *}_j X_j \implies \sigma \models_{\text{lhs(weak)}-\text{self}^{\text{R}}} A \mathrel{-\!\!*} \{X_j\}.$$

holds under some conditions.

We now try to prove the above proposition. From the left-hand side of the implication, we know that for any compatible state a such that $a \models A$,

$$\sigma \bullet a \models \mathop{\Large *}_j X_j.$$

From the definition of separating conjunction, we know that we can split the state $(\sigma \bullet a)$ into $\overline{x_j}$, with $x_j \models X_j$. However, it is important to note that $(\sigma \bullet a)$ is a single state in the PCM $(\Sigma, \bullet, u)$, thus there is no way to convert a split of the entire state into two splits of two substates. In other words, we do not know if there exists a split of σ and a split of a such that the addition of these two split is the split $\overline{x_j}$.

However, we observe that such splits usually exist. Or more generally, the many useful PCMs satisfy *cross-split*.

Definition 5.2 (*cross-split*)**.** $a \bullet b = z \land c \bullet d = z \implies \exists\, ac, ad, bc, bd,$

$$ac \bullet ad = a \land bc \bullet bd = b \land ac \bullet bc = c \land ad \bullet bd = d.$$

Under the assumption that the underlying PCM satisfies cross-split, the converse of Theorem 5.1 holds.

The definition takes from the paper by Dockins et al. [5]. The authors mention that, "*There are reasonable separation logics that have models where disjointness, **cross-split**, and/or infinite splittability are false.*" but do not provide examples. However, based on our observation, such separation algebras are not needed in our use cases.

Theorem 5.2. *If the PCM* $(\Sigma, \bullet, u)$ *satisfies cross-split,*

$$\sigma \models A \mathrel{-\!\!*} \mathop{\Large *}_{j} X_j \implies \sigma \models_{\text{lhs(weak)}-\text{self}^{\text{R}}} A \longrightarrow\!\!* \{X_j\}.$$

Wand separation

Definition 5.1 gives us 1–n equivalence. However, it violates the property of wand separation.

Example 5.1. Let

$A \triangleq$ `acc(x.b,`$\frac{1}{2}$`)`,
$X \triangleq$ `acc(x.b,`$\frac{1}{2}$`) && (x.b==>acc(x.f)) && (!x.b==>acc(x.g))`,
$Y \triangleq$ `acc(x.b,`$\frac{1}{2}$`) && (!x.b==>acc(x.f)) && (x.b==>acc(x.g))`.

Then, the following footprint of the hyperwand is valid:

$$\texttt{acc(x.f) \&\& acc(x.g)} \models_{\text{lhs(weak)}-\text{self}^{\text{R}}} \boxed{\begin{array}{l} A \longrightarrow\!\!* X \\ A \longrightarrow\!\!* Y \end{array}}.$$

However,

$$\texttt{acc(x.f) \&\& acc(x.g)} \not\models (A \mathrel{-\!\!*} X) * (A \mathrel{-\!\!*} Y).$$

The issue[2] arises from the fact that the two left-hand sides of this hyperwand are highly entangled. When we consider the hyperwand as a whole, because of the compatibility restriction in the definition, we have to provide compatible states for left-hand sides. As a result, we can reason about the hyperwand with all the global information, and say that the value of `x.b` must be either *true* or *false*. Therefore, we can construct split functions for the footprint, and split the footprint depending on the left-hand-side states.

However, when we consider the two separate wands as two unrelated entities, problems arise. Most importantly, we lose the information that the two left-hand sides are actually completely related, and we cannot perform the split before knowing the left-hand-side states. This prevents us from getting two separate wands with their own dedicated footprints.

[2] When we say "issue", we do not mean it is necessarily a negative problem of this definition, but some important properties that one needs to consider when dealing with it.

Full release

Worse, Definition 5.1 does not even satisfy full release, which is an essential property to have. It arises from the same reason as wand separation. The only difference is that now the left-hand sides are

entangled with the footprint instead of other left-hand sides. Notice that in Definition 5.1, the validity of the split

$$\bigoplus_j \sigma_j(\overline{a_i}) \preceq \sigma$$

needs to hold only if the supplied $\overline{a_i}$ is compatible with the footprint σ. We can construct a hyperwand whose left-hand-side states are fixed by the footprint, but are unconstrained if the footprint gets split and the constraint does not exist for the remaining wand anymore.

Example 5.2. The example is constructed by "applying" one left-hand side of the hyperwand in Example 5.1.

Let

$A \triangleq \texttt{acc(x.b,}\tfrac{1}{2}\texttt{)}$,
$X \triangleq \texttt{acc(x.b,}\tfrac{1}{2}\texttt{) \&\& (x.b==>acc(x.f)) \&\& (!x.b==>acc(x.g))}$,
$Y \triangleq \texttt{acc(x.b,}\tfrac{1}{2}\texttt{) \&\& (!x.b==>acc(x.f)) \&\& (x.b==>acc(x.g))}$.

the following footprint of the hyperwand is valid:

$$\texttt{acc(x.f) \&\& acc(x.g) \&\& acc(x.b,}\tfrac{1}{2}\texttt{) \&\& x.b == true} \models_{\text{lhs(weak)}-\text{self}^{\text{R}}} \begin{array}{|c|}\hline \quad * X \\ A \mathrel{-\!\!*} Y \\ \hline \end{array}$$

The allocation of `acc(x.f)` and `acc(x.g)` to the right-hand side X or Y is fixed due to the value of `x.b` being `true`. However,

$$\texttt{acc(x.f) \&\& acc(x.g) \&\& acc(x.b,}\tfrac{1}{2}\texttt{) \&\& x.b == true} \not\models X * (A \mathrel{-\!\!*} Y).$$

This is because the permission to `x.f` and `x.b` is allocated to X, but nothing about the value of `x.b` is given to $A \mathrel{-\!\!*} Y$. Therefore, `acc(x.g)` is not a valid footprint of the wand $A \mathrel{-\!\!*} Y$.

The fact that lhs(weak)–self$^{\text{R}}$ hyperwands do not satisfy full release makes this definition useless, since one cannot get anything useful from the definition of the hyperwand.

5.2 Improving the Definition

Definition 5.1 does not satisfy wand separation and full release. However, we can tweak the definition a bit to obtain these good properties. The key is to make sure that the split functions of the footprint are always a "strict split" whatever states are provided to them, even if the states do not satisfy the corresponding left-hand sides. The formal definition is given below.

Definition 5.3 (*lhs(strict)-self (restrictive compatibility) hyperwand*)**.** $\sigma \models \{A_i\} \dashrightarrow\!\!* \{X_j\}$ iff. $\exists$ *footprint split functions* $\overline{\sigma_j}$, $\exists$ *split functions* $\overline{f_i}$, such that

1. $\forall \overline{a_i},\ \bigoplus_j \sigma_j(\overline{a_i}) = \sigma$;
2. $i \notin \mathrm{conn}(X_j) \implies \sigma_j(\cdot)$ is constant with respect to a_i;
3. $\bigoplus_j f_i(a, j) = a$;
4. $\forall j,\ j \notin \mathrm{conn}(A_i) \implies f_i(a, j) = u$;
5. $\forall \overline{a_i}$ compatible with σ, $\forall j$,
$$(\forall i,\ i \in \mathrm{conn}(X_j) \implies a_i \models A_i) \implies \sigma_j(\overline{a_i}) \bullet \bigoplus_i f_i(a_i, j) \models X_j.$$

The only thing changed from Definition 5.1 is point 1. For reference, we put point 1 of Definition 5.1 below.

$$\forall \overline{a_i} \text{ compatible with } \sigma,\ \bigoplus_j \sigma_j(\overline{a_i}) \preceq \sigma \text{ and}$$
$$(\forall i,\ a_i \models A_i) \implies \bigoplus_j \sigma_j(\overline{a_i}) = \sigma$$

The difference is that, Definition 5.1 only requires that the split functions $\overline{\sigma_i}$ to be a real split if provided $\overline{a_i}$ are compatible with each other and the footprint σ. However, Definition 5.3 requires the split functions to give a strict split for any provided states. This forces the split functions to give a "natural split" by providing some unsatisfied states (*e.g.*, empty states), and thus enables a fixed split if the hyperwand shape consists of two unconnected parts.

This improvement is a solid step towards the goal of satisfying full release and wand separation. However, this definition still does not satisfy these two properties. This is a consequence of point 5 of Definition 5.3. This requirement states that, *only if* the left-hand-side states $\overline{a_i}$ are compatible with the footprint σ, then the sums of the corresponding states satisfy the right-hand sides. However, if we separate a disconnected hyperwand into two parts, the set of compatible states for the left-hand sides expands, and this definition is not strong enough to guarantee the satisfiability of the right-hand sides.

To illustrate this issue, we use the following example.

Example 5.3. Consider the following hyperwand

$$\begin{array}{c} * \texttt{acc(x.b,}\tfrac{1}{2}\texttt{)} \\ \texttt{acc(x.b,}\tfrac{1}{2}\texttt{)} \dashrightarrow\!\!* \texttt{acc(x.b,}\tfrac{1}{2}\texttt{) \&\& x.b==>acc(x.f) \&\& !x.b==>acc(x.g)} \end{array}$$

with the footprint

$$\texttt{acc(x.b,}\tfrac{1}{2}\texttt{) \&\& x.b == true \&\& acc(x.f).}$$

We do not need the permission to `x.g` in the footprint because the bottom right-hand side only needs to be satisfied if the left-hand side is compatible with the (whole) footprint, which means that we do not need to consider the case where the value of `x.b` is `false`. However, if we want to prove wand separation (full release) for this hyperwand, we need to prove that the magic wand

$$\texttt{acc(x.b,}\tfrac{1}{2}\texttt{)} \mathrel{-\!\!*} \texttt{acc(x.b,}\tfrac{1}{2}\texttt{) \&\& x.b==>acc(x.f) \&\& !x.b==>acc(x.g)}$$

is satisfied by the footprint `acc(x.f)` which is clearly untrue.

The core of this problem is that when we separate a (hyper-)wand out of a hyperwand, we only consider the part of the footprint which is allocated to the separate part, and more left-hand-side states become compatible as a result. These states are not covered by the original definition for the hyperwand.

5.3 *Improving the Definition Again*

To address the problem mentioned in the previous section, we need to modify Point 5 as follows.

$$\begin{gathered}\forall \overline{a_i},\ \forall j,\ (\forall i,\ i \in \mathrm{conn}(X_j) \implies a_i \models A_i) \implies \\ \sigma_j(\overline{a_i}) \bullet \bigoplus_i f_i(a_i, j) \text{ is defined} \implies \sigma_j(\overline{a_i}) \bullet \bigoplus_i f_i(a_i, j) \models X_j\end{gathered} \tag{5.2}$$

For reference, point 5 of Definition 5.3 is as follows.

$$\begin{gathered}\forall \overline{a_i} \text{ compatible with } \sigma,\ \forall j, \\ (\forall i,\ i \in \mathrm{conn}(X_j) \implies a_i \models A_i) \implies \sigma_j(\overline{a_i}) \bullet \bigoplus_i f_i(a_i, j) \models X_j.\end{gathered}$$

The change gives us the following *permissive compatibility* definition.

Definition 5.4 (*lhs(strict)-self (permissive compatibility) hyperwand*)**.** $\sigma \models \{A_i\} \cdots\!\!* \{X_j\}$ iff. $\exists$ *footprint split functions* $\overline{\sigma_j}$, $\exists$ *split functions* $\overline{f_i}$, such that

1. $\forall \overline{a_i},\ \bigoplus_j \sigma_j(\overline{a_i}) = \sigma$;
2. $i \notin \mathrm{conn}(X_j) \implies \sigma_j(\cdot)$ is constant with respect to a_i;
3. $\bigoplus_j f_i(a, j) = a$;

4. $\forall j,\ j \notin \mathrm{conn}(A_i) \implies f_i(a, j) = \varnothing$;
5. $\forall \overline{a_i},\ \forall j,\ (\forall i,\ i \in \mathrm{conn}(X_j) \implies a_i \models A_i) \implies$
   $\sigma_j(\overline{a_i}) \bullet \bigoplus_i f_i(a_i, j)$ is defined $\implies \sigma_j(\overline{a_i}) \bullet \bigoplus_i f_i(a_i, j) \models X_j$.

In Point 5, we only require the *relevant part* of the provided left-hand-side states $\overline{a_i}$ to be compatible. Specifically, for each right-hand side X_j, only the split of the footprint $\sigma_j(\overline{a_i})$ and the part of the states that is given to X_j are considered. If they are compatible, then the sum of these states must satisfy X_j.

This definition satisfies both full release and wand separation. It is one of the most important discoveries of this project, so we state it formally and prove it below. Since wand separation is stronger than full release, we only need to prove the former. To prevent too much formal argument in the proof, we only prove a special case, but the proof can be generalized to arbitrary hyperwands.

Theorem 5.3. *If*

$$\sigma \models_{\mathrm{lhs(strict)-self^P}} \boxed{\begin{array}{l} A \longrightarrow\!\!* X \\ B \longrightarrow\!\!* Y \end{array}},$$

then

$$\sigma \models (A \mathrel{-\!\!*} X) * (B \mathrel{-\!\!*} Y).$$

Proof. From the premise, we obtain the split functions

$$\sigma_X(a), \sigma_Y(b), f(a), g(b).$$

Since $\forall a, b,\ \sigma_X(a) \bullet \sigma_Y(b) = \sigma$, σ_X and σ_Y are constant functions. Let their values be σ_X and σ_Y, respectively. From Point 3 of Definition 5.4, the sum of the split functions for each left-hand side must add up to the state itself. In this particular hyperwand, we have $f(a) = a$ and $g(b) = b$. Now we prove $\sigma_X \models A \mathrel{-\!\!*} X$, and $\sigma_Y \models B \mathrel{-\!\!*} Y$ follows by symmetry.

Fix a such that $a \perp \sigma_X$ and $a \models A$. From Point 5, because $a \bullet \sigma_X$ is defined, $a \bullet \sigma_X = a \bullet \sigma_X(a) \models X$. □

To make the definition useful, inductive pattern must also be satisfied. We prove the inductive pattern on Definition 5.4 on a specific case, but the proof can be generalized.

Theorem 5.4. *If*

$$\sigma \models_{\mathrm{lhs(strict)-self^P}} \boxed{\begin{array}{ll} & -\!\!* X \\ A & \\ & -\!\!* Y \\ B & \\ & -\!\!* Z \end{array}},$$

then

$$\sigma * A \models_{\mathrm{lhs(strict)-self^P}} \boxed{\begin{array}{c} * X \\ B \lessdot \\ * Y \\ * Z \end{array}}.$$

Proof. Fix some state a such that $a \perp \sigma$ and $a \models A$. Now we prove that $a \bullet \sigma$ is a valid footprint of the partially applied hyperwand.

From the definition, we obtain the split functions

$$\sigma_X(a), \sigma_Y(a,b), \sigma_Z(b), f(a), g(b)$$

where $f(a)$ represent the part of the state given by A to X, and $g(b)$ represents the part of the state given by Y to Z. The split functions from A and B to Y are just represented by $a \ominus f(a)$ and $b \ominus g(b)$, respectively.

We construct the split functions for the partially-applied hyperwand as follows.

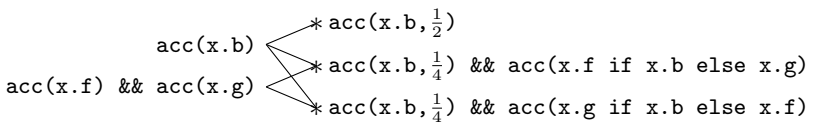
$$\begin{aligned} \sigma'_X &\triangleq \sigma_X(a) \bullet f(a), \\ \sigma'_Y(b) &\triangleq \sigma_Y(a,b) \bullet (a \ominus f(a)), \\ \sigma'_Z(b) &\triangleq \sigma_Z(b), \\ g'(b) &\triangleq g(b). \end{aligned}$$

These split functions are derived from those for the original hyperwand and are restrictions of those functions. As a result, they sum up to the corresponding states or the footprint and the sum satisfies the right-hand sides. □

5.4 LHS-LHS Definition

There still exist some hyperwand definitions that satisfy the desirable properties but are not covered by previous definitions. These examples are constructed by moving the footprint to new left-hand sides, and making the splits of the left-hand-side states depend on these left-hand sides.

Example 5.4. Consider the following hyperwand.



We use `acc(x.f if x.b else x.g)` as a shorthand of `x.b ==> acc(x.f) && !x.b ==> acc(x.g)`.

The left-hand side `acc(x.b)` provides permission to all three right-hand sides and information to the second and the third right-hand sides. The second left-hand side provides permission to the second and the third right-hand sides. Specifically, the split of the second

left-hand-side state cannot be determined before knowing the value of `x.b`. As a result, the split function for the second left-hand side should depend on both left-hand sides.

To cover this use case, we extend Definition 5.4 to give split functions of left-hand sides the information about other (connected) right-hand-side states.

Definition 5.5 (*lhs(strict)-lhs hyperwand*)**.** $\sigma \models \{A_i\} \cdots\!* \{X_j\}$ iff. $\exists$ *footprint split functions* $\overline{\sigma_j}$, $\exists$ *split functions* $\overline{f_i}$, such that

1. $\forall \overline{a_i},\ \bigoplus_j \sigma_j(\overline{a_i}) = \sigma$;
2. $i \notin \mathrm{conn}(X_j) \implies \sigma_j(\cdot)$ is constant with respect to a_i;
3. $\bigoplus_j f_i(\overline{a_i}, j) = a_i$;
4. $i \notin \mathrm{conn}(X_j) \implies f_i(\overline{a_i}, j)$ is constant with respect to a_i;
5. $\forall j,\ j \notin \mathrm{conn}(A_i) \implies f_i(\overline{a_i}, j) = \varnothing$;
6. $\forall \overline{a_i},\ \forall j,\ (\forall i,\ i \in \mathrm{conn}(X_j) \implies a_i \models A_i) \implies$
   $\sigma_j(\overline{a_i}) \bullet \bigoplus_i f_i(\overline{a_i}, j)$ is defined $\implies \sigma_j(\overline{a_i}) \bullet \bigoplus_i f_i(\overline{a_i}, j) \models X_j$.

From now on, we omit the *restrictive compatibility* and *permissive compatibility* description in the definition names. Permissive compatibility is the default.

Theorem 5.5. *Definition 5.5 satisfies the inductive pattern, full separation, wand separation, and 1–n equivalence.*

5.5 All-All Definition and Restrictions on Predicates

We have already shown that the split functions can depend on different set of states, which gives them varied expressivity. The most extreme choice we can make is to make all split functions depend on all left-hand sides.

Definition 5.6 (*all–all (restrictive compatibility) hyperwand*)**.** $\sigma \models \{A_i\} \cdots\!* \{X_j\}$ iff. $\exists$ *footprint split functions* $\overline{\sigma_j}$, $\exists$ *split functions* $\overline{f_i}$, such that

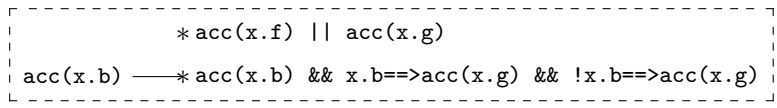
1. $\forall \overline{a_i},\ \bigoplus_j \sigma_j(\overline{a_i}) = \sigma$;
2. $\bigoplus_j f_i(\overline{a_i}, j) = a_i$;
3. $\forall j,\ j \notin \mathrm{conn}(A_i) \implies f_i(\overline{a_i}, j) = \varnothing$;
4. $\forall \overline{a_i}$ compatible with σ, $\forall j$,
   $(\forall i,\ i \in \mathrm{conn}(X_j) \implies a_i \models A_i) \implies \sigma_j(\overline{a_i}) \bullet \bigoplus_i f_i(\overline{a_i}, j) \models X_j$.

Compared with Definition 5.4, point 2 and 4 are removed because now the functions can take every parameter into consideration.

Unsurprisingly, this definition does not give us a lot of good properties, as we demonstrate with the following example[3].

[3] This example is extremely similar to the unsound packaging example presented by Dardinier et al. [6]. Although not directly related, readers might be interested in this example to see the unsoundness issue.

Example 5.5. Consider the hyperwand



with the footprint

`acc(x.f) && acc(x.g).`

The state of the second right-hand side depends on the value of `x.b`, which is only known after applying the second left-hand side. As a result, the split of the footprint cannot be fixed without further information.

The example above shows that Definition 5.6 does not satisfy full release and wand separation. It still supports the inductive pattern.

One advantage of this definition is that it is potentially easy to automate[4]. So it is worth exploring this definition and imposing realistic restrictions on the predicates to see if we could achieve some usefulness.

[4] One possible way to automate is to inhale every possible subset of left-hand sides and exhale any right-hand side that becomes dangling if the subset of left-hand sides are applied. However, the correctness of this approach remains to be formally proven.

5.5.1 Making Right-Hand Sides Precise

The first step to restrict *all–all* hyperwands is to make all right-hand sides precise.

Definition 5.7 (*precise*)**.** A predicate P is precise, iff. $\forall\, x$, there exists at most one substate of x, x', such that $x' \models P$.

Preciseness rules out Example 5.5, because the state `acc(x.f) && acc(x.g)` has two substates, namely `acc(x.f)` and `acc(x.g)`, that both satisfy the first right-hand-side assertion.

With preciseness, we can derive a weaker version of full release. In this weaker version, we only require some states to satisfy the dangling right-hand sides, but do not care about the remaining hyperwand. We could prove the following theorem.

Theorem 5.6. *If*

$$\sigma \models_{\text{all-all}} \boxed{\begin{array}{c} *X \\ \mathcal{W} \end{array}},$$

then

$$\exists\, \textit{unique}\ x \preceq \sigma,\ x \models X.$$

Proof. Follows directly from the definition of all-all hyperwands and the preciseness of X. □

However, we are not able to prove the remaining footprint $\sigma \ominus x$ satisfies the hyperwand $\mathcal{W}$. The reason is exactly the same as Definition 5.3: giving out part of the footprint to X opens up new possibilities for the left-hand-side states of $\mathcal{W}$, and the definition does not cover these cases.

5.5.2 Making Left-Hand Sides Compatible

To address the problem mentioned above, we impose a direct and rather rough restriction—we require that for any states that satisfy the left-hand sides, they must be compatible together with the footprint. The formal definition of compatible predicates is given below.

Definition 5.8 (*compatible predicates*)**.** A set of predicates

$$A_1, A_2, \cdots, A_n$$

are considered *compatible*, iff.

$$\forall\, a_1 \models A_1, a_2 \models A_2, \cdots, a_n \models A_n,\ \bigoplus_{i=1}^{n} a_i \text{ is defined.}$$

Remark 5.1. When we say all left-hand sides are compatible, *together with the footprint*, it means

$$\forall\, a_1 \models A_1, a_2 \models A_2, \cdots, a_n \models A_n,\ \sigma \bullet \bigoplus_{i=1}^{n} a_i \text{ is defined.}$$

This restriction is powerful but at the same time restrictive. For example, the assertion `List(x)`[5] and `List(y)` are not compatible, because they might share same nodes if considered separately. If this restriction is assumed, we do not need to worry about the compatibility issues any more, which essentially solves the problem we have been facing all the time. First, we do not need the *strict* definitions any more. Even the *weak* definitions will give a fixed split in case of wand separation. Second, we do not need *permissive compatibility* any more. Under the assumption of predicate compatibility, it is equivalent to *restrictive compatibility*. This equivalence might make it easier to automate.

[5] `List(x)` represents a linked list starting at `x`.

6 Conclusion & Future Work

Motivated by the verification of reborrow functions in Rust, we explored a novel separation logic connective, *hyperwand*. We mainly explored a class of definition which is based on split functions of individual states (the footprint and left-hand-side states). We presented the definitions from the least to the most expressive, which are also observed to have increasing difficulty of automation and diminishing good properties (full release and wand separation). We especially focus on two definitions that cover the Rust use case and also simple enough so that they might potentially be automated without too much effort. The more powerful definitions serve a theoretical interest but are much less clear how they could be used or automated. See Table 6.1 and 6.2 for an overview of all the hyperwand definitions explored and their properties.

Inductive Pattern

LHS \ footprint	fixed	LHS (strict)	LHS (weak)	all
fixed	✓	✓	✓	✓
self	✓	✓	✓	✓
LHS (strict)	✗	✓	✓	✓
LHS (weak)	✗	✗	✓	✓
all	✗	✗	✗	✓

Table 6.1: An overview of all hyperwand definitions explored and if they satisfy *inductive pattern*. The columns indicate what the split functions for the footprint depend on, and the rows indicate what the split functions for the left-hand states depend on. All definitions have *restrictive compatibility*. The table includes all combinations, including the definitions that did not appear in the main text.

Full Release and Wand Separation

LHS \ footprint	fixed	LHS (strict)	LHS (weak)	all
fixed	✓	✓	✗	✗
self	✓	✓	✗	✗
LHS (strict)	✓	✓	✗	✗
LHS (weak)	✓	✓	✗	✗
all	✓	✓	✗	✗

Table 6.2: An overview of all hyperwand definitions explored and if they satisfy *full release* and *wand separation*. The rows and columns have the same meaning as in Table 6.1.

During the project, we also explored a number of other ideas, but

they are not mature enough to cover in the main body of the report and we leave them as future work. We briefly list them below.

Adjointness. The only desired properties are the inductive pattern, full release and wand separation. So, what is the most expressive definition that *only* gives these three properties and nothing else? In other words, does there exist a definition that can be derived directly from these three properties? We have invested a large amount of time to explore this in this project. However, the result is not satisfying, in the sense that we are neither able to prove the adjointness property for any definition, nor able to find any counterexample for the most expressive definition with all three properties (*lhs–lhs* definition).

Splitting the footprint and left-hand-side states together. Instead of giving multiple split functions for the footprint and left-hand-side states, we can have just one split function for each right-hand-side, which gives the sum of everything needed to satisfy the right-hand-side. It has the potential to get rid of cross-split and yet provide adjointness.

Splitting the footprint to left-hand-sides. Instead of splitting the footprint to each right-hand-side, we can first have a fixed split and allocate the split to each left-hand-side, and let it split its share when the left-hand-side states are provided. This makes the proof of wand separation easier and might also be easier to automate. However, its relationship with existing definitions is unclear.

A Discussion of Prusti

A.1 Packing References into `struct`

Assume that we allow references inside `struct` declaration, and a reborrow function stores reborrowed references inside a struct. If we try to specify this reborrow function with a magic wand, the left-hand-side will be a description of the struct, and the right-hand-side will be the packed resources. The question is, how to correctly specify the left-hand-side? We cannot use the usual "tree" representation, since holding a tree predicate enables us to freely modify what is inside the tree. One thing that is perfectly fine to do is replace the packed reference inside the tree with something else. After that, we can still fold the tree to get the "right" predicate but the content has been changed and no longer contains the right-hand-side resources.

For a more elaborate account of this issue, see the Master's Thesis by Gorse [7].

A.2 Functional Properties

Using magic wands (or hyperwands) to specify reborrow function also gives rise to the problem of describing functional properties. Wands are a perfect tool for describing resource relations, but are not suitable for describing functional behaviors. For example, consider the following Rust code.

```
1  fn foo<'a, 'b>(p1: &'a mut Point, p2: &'b mut Point)
2      -> (&'a mut i32, &'b mut i32)
3  {
4      p1.y = p2.y
5      (&mut p1.x, &mut p2.x)
6  }
```

In this example, the shape of the hyperwand is

$$\texttt{a} \mathrel{-\!\!*} \texttt{p1}$$
$$\texttt{b} \mathrel{-\!\!*} \texttt{p2}$$

In addition, we might want to express the property `p1.y = p2.y`.

However, the fact that `p1` and `p2` are two separate right-hand-sides makes this property impossible to express. One must have access to both states of the right-hand-sides to decide if this property holds. To support this property is beyond the scope of this project. However, there is another ongoing project trying to explore the concept of *IDF wands* to separate permission from functional specification in magic wands [8], which we believe is what Rust needs to precisely capture its real semantics.

Bibliography

[1] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL https://doi.org/10.1145/363235.363259.

[2] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, page 55–74, USA, 2002. IEEE Computer Society. ISBN 0769514839.

[3] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):147:1–147:30, October 2019. doi: 10.1145/3360573. URL https://dl.acm.org/doi/10.1145/3360573.

[4] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local Action and Abstract Separation Logic. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 366–378, July 2007. doi: 10.1109/LICS.2007.30. URL https://ieeexplore.ieee.org/abstract/document/4276580.

[5] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A Fresh Look at Separation Algebras and Share Accounting. In Zhenjiang Hu, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 161–177, Berlin, Heidelberg, 2009. Springer. ISBN 978-3-642-10672-9. doi: 10.1007/978-3-642-10672-9_13.

[6] Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. Sound Automation of Magic Wands. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 130–151, Cham, 2022. Springer International Publishing. ISBN 978-3-031-13188-2. doi: 10.1007/978-3-031-13188-2_7.

[7] Lorenz Gorse. Extended Support for Borrowing and Lifetimes in Prusti. Master's thesis.

[8] Manuel Dublanc. Separating Magic Wands and Functional Properties (IDF Wands) — project description.

Index

$\bigoplus$, 7
$\mathcal{W}(\cdot)$, 9
$\cdot \mathrel{-\!\!*} \{\cdot\}$, 10
$\cdots\!\!*$, 10
$\bullet$, 7
$\ominus$, 7
$\bot$, 7
$\preceq$, 7
$\succeq$, 7
$*$, 8
$\llcorner\!\!\llcorner * \{\cdot\}$, 10
$\mathrel{-\!\!*}$, 9

abstract separation logic, 8

bipartite graph, 9

compatible, 7
compatible predicates, 35
cross-split, 26

dangling right-hand side, 10
definitions
  all–all (restrictive compatibility), 33
  fixed-fixed, 16
  fixed-self, 20
  lhs(strict)-lhs, 33
  lhs(strict)-self (permissive compatibility), 30
  lhs(strict)-self (restrictive compatibility), 29
  lhs(weak)–self (restrictive compatibility), 25
degenerate cases, 11
  0–n, 12
  1–n, 12
  1–1, 12
difference, 7

footprint, 9
footprint split functions, 25, 29, 30, 33
full release, 12

hyperwand, 5
  apply, 9

inductive pattern, 11

permissive compatibility, 30, 35
precise, 34
predicate, 8

restrictive compatibility, 25, 35

separate hyperwands, 10
separating conjunction, 8
separating implication, 9
separation algebra, 7
split, 8
split functions, 20, 25, 29, 30, 33
state, 7
substate, 7
superstate, 7

wand separation, 13