Hyperwand: Extending the Magic Wand Operator in Separation Logic

Practical Work Project Supervised by Thibault Dardinier

Yushuo Xiao

November 6, 2023

1 Introduction

Separation logic [6] is an extension of Hoare logic [4] to reason about heap-manipulating programs. Separation logic features two main connectives, separation conjunction $*$, and separation implication $-\!\!*$. A heap σ satisfying $A * B$ is defined as

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

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

$$\sigma \models A -\!\!* B \quad \triangleq \quad \forall \sigma_A. \, \sigma \perp \sigma_A \wedge \sigma_A \models A \Longrightarrow \sigma \oplus \sigma_A \models B,$$

where $\sigma \perp \sigma_A$ denotes the disjointness of σ and σ_A. This definition intuitively means if a heap σ is extended with another disjoint heap σ_A that satisfies A, then the resultant heap $(\sigma \oplus \sigma_A)$ satisfies B. The magic wand has been shown to be useful in recursive data structure traversal, where one needs to get back to the assertion of the entire data structure from assertion to partial 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 two use cases.

Using magic wand to verify Rust programs. In the programming language Rust, magic wands can accurately describe the behavior of a function that returns a mutable reference. 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 accessibility 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 a 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 [2], a verifier for Rust, uses this encoding for functions that return a mutable reference. It encodes the magic wand as a post-condition for the function and applies the wand when the returned reference expires.

However, in 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 program 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 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 otherwise hold fail. This verification issue arose in Prusti as it uses the latter encoding. In this practical work project, we want to explore a separation logic construct that is expressive enough to specify the former property.

Verifying a Treap operation. Consider the Treap data structure [1]. A Treap is a randomized binary search tree. All of Treap's operations rely on two fundamental transformations on the tree, which are *split* and *join*. Given a value v, the split operation splits a Treap into two Treaps. All keys in the first Treap are less than or equal to v, while all keys in the second Treap are greater than v. The join operation joins two Treaps, where all the keys of the first Treap are smaller than any key of the second Treap.

Consider the following operation, where we have two Treaps given by a split operation on value b, and we want to further split these two Treaps into three Treaps, separated by values a and c ($a < b < c$). In other words, we want the keys of the first Treap less than or equal to a, the keys of the second Treap of range $(a, c]$, and the keys of the third Treap greater than c.

Consider a predicate we might use to describe a Treap:

$$\text{Treap}(t, l, r) = \ldots \quad (\textit{formal definition omitted})$$

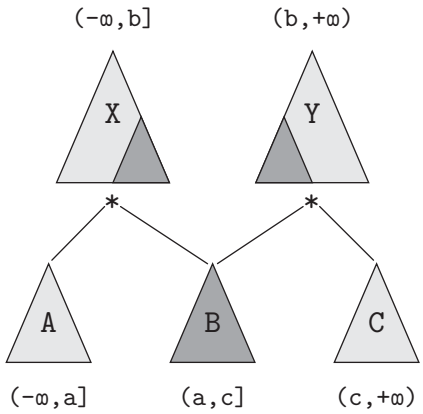


Figure 1: An operation of Treaps that could be specified with hyperwands.

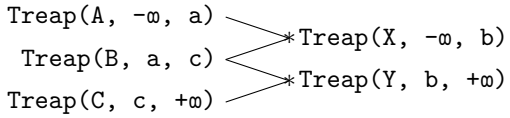
which means the tree t satisfies the Treap data structure requirements, and its keys fall into $(l, r]$. Suppose the procedure we described above is called $\text{Split}(X, Y, a, c)$ and returns a tuple (A, B, C), where X and Y are Treaps satisfying the above property and A, B, C are separated by keys a and c. In Viper [5], a permission based verification language, the pre- and post-condition can be expressed as follows.

```
requires Treap(X, -ϖ, b) && Treap(Y, b, +ϖ)
ensures  Treap(A, -ϖ, a) && Treap(B, a, c) && Treap(C, c, +ϖ)
method   split(X, Y, a, c) returns (A, B, C) { ... }
```

In a verifier based separation logic, we cannot have $\text{Treap}(X, -\infty, b)$ and $\text{Treap}(Y, b, +\infty)$ as the method's post-condition, since A, B, and C share resources with X and Y. A possible way to get back to these predicates after we are done using the returned Treaps is to conjunct the following to the post-condition.

```
ensures  Treap(A, -ϖ, a) && Treap(B, a, c) && Treap(C, c, +ϖ)
         --* Treap(X, -ϖ, b) && Treap(Y, b, +ϖ)
```

With this post-condition, we must give up all of A, B, and C before we are able to get `Treap(X, -ϖ, b)` and `Treap(Y, b, +ϖ)` back. However, considering that X only shares resources with A and B, and Y only shares resources with B and C, we could define more fine-grained assertions to precisely capture this property. Therefore, a generalized version of the magic wand, which we call *hyperwand*, could be used to express it.



With the hyperwand, we are able to first give up `Treap(A, -ϖ, a)` and `Treap(B, a, c)`, and immediately get `Treap(X, -ϖ, b)` and the remaining magic wand `Treap(C, c, +ϖ)` $-\!\!*$ `Treap(Y, b, +ϖ)`. Symmetrically, we can first apply the hyperwand with `Treap(B, a, c)` and `Treap(C, c, +ϖ)`, immediately gaining access to `Treap(Y, b, +ϖ)` and the remaining magic wand `Treap(A, -ϖ, a)` $-\!\!*$ `Treap(X, -ϖ, b)`.

The capabilities of hyperwands look promising. However, many aspects of the hyperwand are still unknown to us, 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. This practical work consists of exploring the theory side of the hyperwand as well as a basic implementation of the hyperwand in Viper.

2 Goals

2.1 Core Goals

The following points constitute the core goals of this project.

- **Give a formal and generalized definition of the hyperwand.** We only gave an informal definition and intuition in the previous section, which is not enough for automation. The first goal of this project is to give a definition of a hyperwand based on separation algebra [3].
- **Prove key properties of the hyperwand.** We might want to derive useful properties of hyperwands, which may be used to simplify the packaging of hyperwands, applying hyperwands, and transforming hyperwands, etc. These properties, together with the definition, should be mechanized in Isabelle/HOL.
- **Design a syntax for hyperwands in Viper.** We want to add support for hyperwands in Viper. The first step towards this goal is to design the syntax for the Viper language. Representing rich structures such as hyperwand might pose some challenges in a textual language like Viper.
- **Implement hyperwands in Viper, including (simple) packaging and applying, in the Carbon backend.** This step contributes towards a usable hyperwand implementation in Viper, which may help verification of Rust code in the Prusti frontend.

2.2 Extension Goals

Depending on the time spent on the core goals, the following extension goals can improve usability of hyperwands and demonstrate its use cases.

- **Find more use cases of hyperwand and encode them in (extended) Viper.**
- **Make package more automated.**

References

[1] Cecilia Aragon and Raimund Seidel. Randomized search trees. In *30th Annual Symposium on Foundations of Computer Science*, pages 540–545, 1989.

[2] 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.

[3] 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, 2007.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969.

[5] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[6] 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.