Internship Report — Mpri M2
Idiomatic specification of magic wands in implicit dynamic frames

Rémi Germe supervised by Jonáš Fiala, Marco Eilers, Peter Müller

Programming Methodology Group, ETH Zürich

General context

Program verification (Section 1) aims to prove that a program satisfies its specification, which could be given as a set of preconditions assumed to hold upon entry and postconditions that must hold upon termination. Yet, heap-manipulating programs require careful reasoning, for example, issues arise from pointers aliasing to the same heap location (details in Section 2). Separation logic (SL) [1] (Section 2) is a popular framework to reason about such programs, natively featuring heaps and access permissions to heap locations. These permissions are required to read or write to a particular memory location and are unique, thus preventing aliasing issues.

Implicit dynamic frames (IDF) [2, 3] (Section 2) is a variant of SL which separates the permission reasoning from the functional specification. Both SL and IDF use predicates to represent (recursive) data structures, for example, `list(x)` might represent that `x` is the head of a well-formed linked list, encapsulating the permissions to access the fields of all its reachable nodes. However, unlike SL, IDF allows *heap-dependent expressions* such as `x.val` in specifications, provided that they are *well-framed*, *i.e.*, the current state holds the permissions to the accessed memory locations. Heap-dependent expressions enable developers to write functional specifications separately from permission reasoning, paving the way for incremental and modular specification.

Viper [4] (Section 3) is a verification infrastructure using IDF to verify programs in an automated fashion, leveraging satisfiability modulo theories (SMT) solvers. It provides an intermediate verification language, two verifiers (*i.e.*, backends) based on symbolic execution (SE) and verification condition generation (VCG) respectively (Section 1), as well as a collection of frontends to verify general-purpose programming languages such as Rust.

Research problem

The magic wand, denoted by A `--*` B, is a connective from SL and IDF used to specify incomplete data structures, such as the prefix of a linked list. Intuitively, a magic wand `list(cur) --* list(root)` is a promise: the wand can be *applied* to exchange permissions to a list starting from `cur` for permissions to a list starting from `root`. Both predicates and magic wands have a *footprint*, *i.e.*, a set of permissions they own. An example of such footprints is depicted in Figure 1.

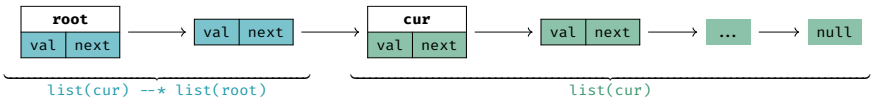


Figure 1: Memory footprints of the predicate `list(cur)` and the magic wand `list(cur) --* list(root)`.

However, unlike predicates whose footprints can be accessed to prove the well-framedness of heap-dependent expressions (*e.g.*, `cur.next.val` is well-framed in `list(cur)`), IDF lacks a mechanism to frame expressions using the footprint of a wand (*e.g.*, `root.val` is well-framed in `list(cur) --* list(root)` but IDF cannot currently prove this). Several challenges explain this absence. Indeed, unlike predicates whose footprints are explicitly (potentially recursively) defined, the footprint of a magic wand is not explicit and can be very unintuitive (see Section 2 and Section 5).

The goal of my internship is to design and implement a new IDF construct in Viper that enables access to the permissions contained within the footprint of a wand, analogous to the mechanism for predicates. When designing this new construct, I consider the following requirements. It should be:

- **sound**, *i.e.*, Viper successfully verifies a program only if it is indeed safe with respect to its specification. Soundness is the one property program verifiers must have, as they must never approve an incorrect program. However, the construct can be *incomplete*, *i.e.*, Viper might fail to successfully verify a correct program. Being incomplete might be annoying, but it is safe as it does not allow drawing a (potentially dangerous) conclusion.
- **expressive**: it should enable users to write specifications or code that currently cannot be expressed in Viper, that is, heap-dependent expressions whose required permissions are contained in the footprint of a wand. This would increase the expressiveness of the code, and pave the way for modular functional specifications of magic wands.
- **lightweight**: to be realistically usable, the proposed construct should be automatable in the existing Viper framework and induce only minimal annotation efforts from users.

My contributions

My contributions focus on the SE backend of Viper and are available online[1]. They consist of the following:

- I propose a new encoding of wands at the SMT level, fixing a soundness issue I discovered (Section 4).
- I extend the Viper language with a new construct allowing users to "attach facts" to existing wands fully automatically, leveraging the new encoding. This enables modular and incremental specifications of programs using wands, as the permission reasoning is carried by the wand itself and functional specifications can be independently "attached" to it in a modular fashion (Section 5).
- I explore the design space of another highly-automated construct to "attach expressions", effectively allowing one to evaluate heap-dependent expressions whose required permissions are in the footprint of the wand (Section 6).

Arguments supporting their validity

A new encoding for magic wands. The existing encoding of wands is relatively intricate, leaving space for unanticipated issues such as the one I discovered. I overcome this shortcoming by proposing a conceptually simpler encoding. Apart from this theoretical argument, the new encoding is evaluated against the large test suite for wands of Viper, exhibiting only one understood completeness regression. Moreover, this encoding is benchmarked against real-world verified codebases and shows equivalent performance.

Attached facts. The potential of attached facts is demonstrated on several classic examples, as well as some new ones. Although a case study on large verified codebases suggests that adapting existing specifications to use them might be non-trivial, early experiments show they could be very useful in Prusti [5], the Rust frontend for Viper.

Attached expressions. Attached expressions are still experimental and only showcased on small examples, that highlight some current limitations, notably that they are not always well-defined. Although I identified a sufficient condition to ensure well-definedness, automating the verification of this condition in Viper remains future work.

Summary and future work

In summary, I propose a sound encoding for magic wands for a symbolic-execution-based verifier and introduce two novel lightweight constructs building upon wands. Attached facts enable incremental and modular functional specifications by clearly decoupling them from permission reasoning, while attached expressions improve the expressiveness of the verifier by allowing one to evaluate expressions depending on the footprint of a wand.

Several questions are left to future work. The integration of attached facts in Viper frontends still has to be investigated. Ultimately, the same work must be done for attached expressions as well, but they currently face automation challenges. Notably, my analysis suggests that a good next step to lift these limitations is to soundly determine under which conditions a valid left-hand side of a wand exists.

Other interesting ideas to explore include formalizing and proving the soundness of the new encoding and "attached" constructs in a proof assistant, or investigating how these constructs could be ported to the second backend of Viper.

[1] My contributions are available on the following repositories:
- for my extensions to the Viper language and the test suite, see: https://github.com/remigerme/silver/tree/rgerme-mw-attached-facts;
- for my extensions to the SE backend and other evaluation artifacts, see: https://github.com/remigerme/silicon/tree/mw-attached-facts.

1 Introduction and Related Work

This section outlines the Viper project, compares it to other automated verifiers, and discusses the formalization of SL and IDF in interactive theorem provers.

Viper [4] provides a simple yet expressive intermediate verification language, the Viper language. Several frontends have been developed on top of Viper, such as Prusti [5, 6] for Rust, Gobra [7] for Go, Nagini [8] for Python, and the VerCors project [9] for Java, C, and more. They all encode the target program (*e.g.*, a Rust program annotated with specifications for Prusti) into a Viper program. Viper programs can be verified by one of two backends based on *symbolic execution* and *verification condition generation* respectively. Figure 2 provides a visual overview of the different components.

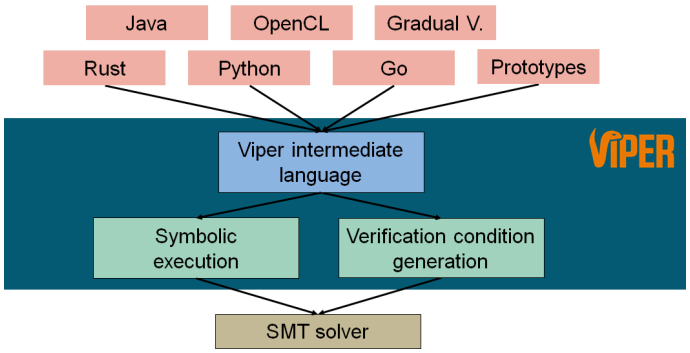


Figure 2: An overview of the Viper project.
This diagram is from the official homepage of the project: https://viper.ethz.ch/[2].

Symbolic execution (SE). SE maintains a symbolic state containing a symbolic heap as well as path conditions, indicating, among other things, which branch is currently being executed. It leverages SMT solvers to check path feasibility and prove the correctness of the program with respect to its specification.

Verification condition generation (VCG). VCG translates a program and its specification into a set of logical formulas whose validity guarantees correctness of the program. Checking the validity of these formulas is then discharged to an SMT solver or another verification tool such as Boogie [10].

Both techniques heavily rely on SMT solvers such as Z3 [11] or CVC5 [12]. Yet, an experience report [13] suggests that verification via SE is generally faster than the VCG approach, although the opposite holds in certain scenarios. My internship solely focused on the Viper SE engine [14, 15].

Other separation-logic-based verifiers. Many other program verifiers based on separation logic (or variants) have been developed. Table 1 provides an overview of some of them. It is worth noting that, to the best of my knowledge, the only other automated verifier to support magic wands is the VerCors project, which builds on top of Viper but without relying on its wand support. In VerCors, a specification with magic wands is encoded into a specification without wands, relying on user-provided annotations [16]. However, this approach requires heavy annotation efforts from users, whereas the support of magic wands in Viper is more automated.

Verifier	Target language(s)	Approach	Supports magic wands
VeriFast [17]	C, Rust, Java	SE	no
GRASShopper [18]	GRASShopper language	VCG	no
VerCors [9]	Java, C, ...	encoding to Viper	yes
Gillian [19, 20]	C, JavaScript, Rust	SE	no

Table 1: A brief overview of some existing verifiers based on separation logic (and variants).

Non-separation-logic-based verifiers. Although separation logic provides a convenient framework to reason about heap-manipulating programs, its full expressiveness, and thus complexity [21], is not always required. For

[2] Accessed 24.07.2026

instance, it might be possible to statically control aliases [22], or leverage the ownership type system of Rust [23, 24] to fall back on a more traditional Hoare logic.

Dafny [25] and Why3 [26] are examples of such projects. Why3 has a structure similar to Viper: its intermediate language, WhyML, is targeted by several frontends: Creusot [23] for Rust or Frama-C [27] for C, for example. Why3 only comes with a VCG backend, but it allows delegating proof obligations to interactive theorem provers, such as Rocq [28] or Isabelle/HOL [29]. This enables users to complete proofs in cases where SMT solvers would fail to conclude automatically.

Separation logic in interactive theorem provers. SL has been formalized in several proof assistants, allowing users to manually verify (concurrent) programs, or prove meta-theoretic results, for example on the Rust type system [30]. The flagship SL formalization is Iris [31], in Rocq. There are also ongoing efforts to port the project to Lean [32, 33], which provides convenient automation features [34]. The key feature of implicit dynamic frames, heap-dependent expression assertions, was also recently brought to Iris in Daenerys [35].

Relevant formalization efforts also include projects aiming at proving the correctness of the verifiers themselves. Verifiers based on both VCG [36] and SE [37] have been investigated, as well as frontends relying on translating a target language to an intermediate verification language [38, 39].

Finally, a logical framework has been developed and formalized in Isabelle/HOL to prove the correctness of algorithms to support magic wands in automated verifiers [40], and it was used to prove the correctness of the current package algorithm of the VCG backend of Viper.

A promising direction for future work is to investigate to what extent these existing formalizations can be leveraged to formally verify my contributions.

2 Logical Foundations

In this section, I present key definitions of SL and highlight how IDF differs from it.

2.1 Separation logic

Heap-manipulating programs require careful reasoning, as illustrated in Example 1 where two pointers can alias the same memory location, which would prevent the specification from holding no matter the code of the `assign` function.

```
void assign(int *x, int *y)
  requires x ≠ NULL && y ≠ NULL
  ensures  *x == 1 && *y == 2
```

Example 1: A motivating (flawed) example for separation logic: the specification cannot hold if x and y alias.

SL [1] is a popular framework to reason about such programs and overcome limitations that stem from working with shared memory. It natively features heaps and access permissions to heap locations. Reading or writing to a memory location requires having the permission to that location, which is unique, thus preventing aliasing issues. A heap is a partial map from memory locations whose access is granted to their values. To manipulate heaps, SL introduces several spatial connectives to reason about permissions.

The points-to predicate ($x \mapsto v$). The points-to predicate is the fundamental building block for specifying memory locations. The assertion $x \mapsto v$ holds in a heap that has (exclusive) permission to the memory location x, and such that the stored value is v. Formally, satisfying this predicate for a heap σ is defined as follows:

$$\sigma \vDash x \mapsto v \overset{\text{def}}{\iff} \sigma = \{(x, v)\} \land x \neq \texttt{null}$$

The separating conjunction ($A * B$). The separating conjunction specifies that A and B must hold (*i.e.* the conjunction $A \land B$ holds) for disjoint heaps. Formally, it is defined as follows, where $\sigma_A \perp \sigma_B$ indicates that the heaps σ_A and σ_B are disjoint and $\sigma_A \uplus \sigma_B$ denotes their disjoint union:

$$\sigma \vDash A * B \overset{\text{def}}{\iff} \exists \sigma_A, \sigma_B : \begin{bmatrix} \sigma & = \sigma_A \uplus \sigma_B \quad (\sigma_A \perp \sigma_B) \\ \sigma_A & \vDash A \\ \sigma_B & \vDash B \end{bmatrix}$$

Permissions from two separately conjuncted assertions are guaranteed not to alias. For instance, the correct precondition for Example 1 is $\exists v_x : x \mapsto v_x * \exists v_y : y \mapsto v_y$, and the postcondition is $x \mapsto 1 * y \mapsto 2$. A heap σ satisfying the precondition can be split in two disjoint heaps σ_x and σ_y, satisfying the points-to predicate for x and y respectively, thus x and y cannot alias.

Predicates. Predicates can represent (potentially recursive) data structures. Assuming a memory model where heap locations can be accessed via object fields, a linked list predicate in SL is usually[3] defined inductively as follows:

$$\begin{aligned} \text{list}(p, []) &:= p = \texttt{null} \\ \text{list}(p, v :: L) &:= \exists p' : p.\text{val} \mapsto v * p.\text{next} \mapsto p' * \text{list}(p', L) \end{aligned}$$

where $\text{list}(p, L)$ denotes that the object p points to a valid linked list in the heap, whose elements exactly correspond to the mathematical sequence L. Note that this definition prevents circular lists as the head of the list must be separated from the suffix list due to the separating conjunction.

A motivating example for magic wands [41]. Consider the function head(p) that returns the (pointer to the) head of the list represented by p, without modifying the list. Its precondition is $\exists v, L : \text{list}(p, v :: L)$, which guarantees that p represents a valid non-empty list. Its postcondition, using the special keyword ret to denote the return value, should look like this: $\text{ret.val} \mapsto v * \text{list}(p, v :: L)$. However, this is not valid in SL as the two assertions are not separated: a valid heap for $\text{list}(p, v :: L)$ must have permission to the location ret.val. Another way to see that this postcondition is wrong is to remark that the caller might later assign the value v' to ret.val for example (because it has permission to ret.val after the call), which invalidates the assertion $\text{list}(p, v :: L)$. In that case, it should then be $\text{list}(p, v' :: L)$.

Instead, the postcondition should describe the first element and a suffix list, such that, if we provide back the first element with a value v', the whole list is valid once again and is mathematically $v' :: L$. This incomplete list, missing its first element, can be described using a magic wand:

$$\text{ret.val} \mapsto v * (\forall v' : \text{ret.val} \mapsto v' \text{ --* } \text{list}(p, v' :: L))$$

The magic wand (A --* B). The magic wand A --* B can be seen as a promise: "if the left-hand side (LHS) assertion A is provided, the wand can be exchanged for its right-hand side (RHS) assertion B". Exchanging A and the wand is called *applying* the wand. On the example above, if we provide back permission to ret.val with any value v', the wand can be traded for the whole updated list $v' :: L$. Intuitively, to hold that promise, magic wands can contain permissions, captured in their *footprint*. In the example, this footprint is the permission to the suffix list, illustrated in Figure 3. The magic wand is also known as the separating implication, as it is an implication $A \Rightarrow B$ augmented with permission reasoning: it can only be applied once because assertion A may require permissions. Formally, it is defined as follows, where σ is the footprint of the wand:

$$\sigma \vDash A \text{ --* } B \overset{\text{def}}{\iff} \forall \sigma_A : \sigma_A \perp \sigma \wedge \sigma_A \vDash A \Rightarrow \sigma_A \uplus \sigma \vDash B$$

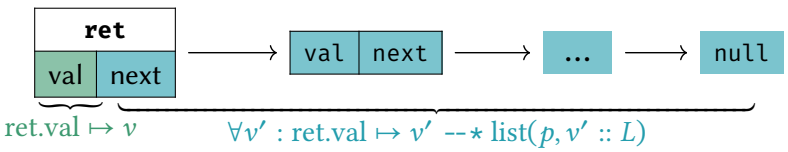


Figure 3: Memory footprints for the `head(p)` example.

Vacuous wands. However, magic wands can be very counterintuitive. One of the reasons is that a wand is fundamentally an implication, and that implications can be vacuously true (if their LHS is false). For instance, the wand from the `head` example is satisfied by two footprints σ:

- σ = "the suffix list": permissions to the rest of the list are packaged in the footprint of the wand, which is the intuitive scenario described in Figure 3;

[3] For my presentation to be smoother when transitioning to IDF, I chose to use fields in SL. However this is not so standard to the best of my knowledge.

- $\sigma = \{(\text{ret.val}, v'')\}$ for any v'': in that case, there does not exist any σ_A such that $\sigma_A \perp \sigma \wedge \sigma_A \vDash \text{ret.val} \mapsto v'$. Indeed, $\sigma_A \vDash \text{ret.val} \mapsto v'$ implies that $\sigma_A = \{(\text{ret.val}, v')\}$, which contradicts $\sigma_A \perp \sigma$ as both heaps would have permission to ret.val. Thus, the LHS of the implication in the definition of the wand is false, and as a result, the wand is valid, without having to be able to provide its RHS. Yet, the wand will never be applied, as the permission required by its LHS has been packaged inside its footprint.

Fractional permissions. The previous setup can be augmented with a more granular definition for permissions [42]. Fractional permissions, denoted by $x \overset{p}{\mapsto} v$, are indicated by a fraction p: 0 indicates no permission at all, whereas 1 indicates full permission. Read-only operations require any non-zero amount of permission, whereas mutability requires the full permission.

When combining two heaps σ_A and σ_B, the resulting permission of a memory location is the sum of the permissions held in both heaps. Moreover, fractional permissions must agree on the value for that memory location. For example, $x \overset{1/2}{\mapsto} v_1 * x \overset{1/4}{\mapsto} v_2$ can be simplified to $x \overset{3/4}{\mapsto} v_1$, and $v_1 = v_2$ holds.

Fractional permissions can be used to verify concurrent programs and ensure data race freedom. Permission to a location can be split across multiple threads, allowing each thread (that has a non-zero permission) to read the value but not mutate it, unless that thread has full permission, implying that all other threads have zero permission.

2.2 Implicit dynamic frames

IDF [2, 3] is a variant of SL which separates the specification of the permission to a heap location from the value of that heap location. Whereas in SL both permission and value are bundled by the points-to predicate $x.f \mapsto v$, in IDF these two parts are split:

- permissions are represented using the *accessibility assertion*, denoted by $\text{acc}(x.f)$. Its formal definition is very close to the points-to predicate, but it does not specify a value: $\sigma \vDash \text{acc}(x.f) \overset{\text{def}}{\iff} \exists v : \sigma = \{(x.f, v)\} \wedge x \neq \texttt{null}$;
- heap-dependent expressions such as $x.f$ can be used to refer directly to the value of the heap location, provided that the current state holds the permission to it.

Formally, the points-to predicate $x.f \mapsto v$ in SL is equivalent to $\text{acc}(x.f) * x.f = v$ in IDF. This separation has consequences for representation predicates, for example, the list predicate in IDF is usually defined as follows:

$$\text{list}(p) \coloneqq p \neq \texttt{null} \Rightarrow \text{acc}(p.\text{val}) * \text{acc}(p.\text{next}) * \text{list}(p.\text{next})$$

This definition does not explicitly specify the values of the nodes, but they can be retrieved using a *pure heap-dependent function*. These functions only exist in IDF as they are defined using heap-dependent expressions. Moreover, they must be pure, meaning they cannot have side effects. Because of this guarantee, callers of heap-dependent functions only need to show they have the permissions required by the precondition, but do not need to give them up. As an example, the values of the nodes of a list can be retrieved using the following heap-dependent function, which requires $\text{list}(p)$ as a precondition:

$$\text{elems}(p) \ \#\text{requires list}(p) \coloneqq \begin{cases} [] & \text{if } p = \texttt{null} \\ p.\text{val} :: \text{elems}(p.\text{next}) & \text{otherwise} \end{cases}$$

This function is well-framed because all required permissions are contained in the precondition $\text{list}(p)$, that is, both accessed fields p.val and p.next, as well as $\text{list}(p.\text{next})$ (to meet the precondition of the recursive call), are contained in $\text{list}(p)$.

Finally, the precondition of the aforementioned (non-heap-dependent) function `head(p)` is $\text{list}(p)$ and its postcondition is $\text{acc}(\text{ret.val}) * (\text{acc}(\text{ret.val}) \mathrel{-\!\!*} \text{list}(p))$, similar to the SL specification which nonetheless involved values on top of these permissions.

3 Viper

In this section, I give an overview of the Viper language, and then introduce general mechanisms of the SE backend and more specifically the handling of magic wands.

3.1 The Viper language

The Viper intermediate verification language [4, 43] is an imperative programming language featuring specification annotations. I will now review some relevant features of the language, some of which are exemplified in Example 2 (&& denotes the separating conjunction $*$).

```
field val: Int
field next: Ref

predicate list(x: Ref) {
  x ≠ null ⟹ acc(x.val) && acc(x.next) && list(x.next)
}

function len(x: Ref): Int
  requires list(x)
{
  x == null ? 0 : 1 + unfolding list(x) in len(x.next)
}

method iter_demo(root: Ref)
  requires list(root)
  ensures  list(root)
  ensures  len(root) == old(len(root))
{
  // Parametric shorthands for the LHS and RHS of the wand
  define A(c, l) list(c) && len(c) == l - 1
  define B       list(root) && len(root) == old(len(root))

  var cur: Ref := root
  var len_cur: Int := len(cur)
  package A(cur, len_cur) --* B

  while (cur ≠ null)
    invariant list(cur)
    invariant A(cur, len_cur) --* B
    invariant len_cur == len(cur)
  {
    var prev_cur: Ref := cur
    var prev_len_cur: Int := len_cur

    unfold list(cur)
    cur := cur.next
    len_cur := len(cur)

    package A(cur, len_cur) --* B {
      fold list(prev_cur)
      apply A(prev_cur, prev_len_cur) --* B
    }
  }

  apply A(cur, len_cur) --* B
}
```

Example 2: (left-then-right) A Viper program manipulating linked lists using a magic wand to ensure that `root` is still a valid head of a list after the loop.

Data types. Built-in data types are provided such as `Int`, `Bool`, and `Ref` for references. To encode rich constructs, arbitrary data types can be built using either predicates or domains, which are outside the scope of this report.

Fields. Every reference has all fields, which can be accessed provided the reference has permission to the field.

Permissions. Permissions are indicated by the built-in `acc(resource, amount)` assertion, where `resource` can be a field (*e.g.*, `x.val`) or a predicate (*e.g.*, `list(x)`). The amount is the full permission by default. For predicates, the `acc` keyword can be omitted, leaving just the predicate.

Functions. These are the pure heap-dependent functions mentioned in Section 2. The body of a function is optional and consists in the single returned heap-dependent expression if provided.

The fact that a concrete argument `arg` satisfies the precondition of a function `f` for a function call `f(arg)` is represented at the SMT level by an uninterpreted function $\mathtt{f}_{\text{token}}$. Then, the actual definition of the function `f` is guarded by the token for that argument: $\forall x : \mathtt{f}_{\text{token}}(x) \Rightarrow \mathtt{f}(x) = \mathtt{def}$, guaranteeing the function definition is available only for valid inputs. When `f` is called with argument `arg`, the verifier checks that `arg` respects the precondition and then yields a token $\mathtt{f}_{\text{token}}(\mathtt{arg})$, allowing the solver to access the actual definition. My contributions are inspired from this idea.

Predicates. In Viper, recursive predicates are *iso-recursive*, meaning a predicate and its body are considered distinct but can be exchanged one another. The statement `unfold p` (resp. `fold p`) allows exchanging a predicate `p` for its body (and vice versa). The construct `unfolding p in e` can be used to temporarily unfold a predicate `p` to prove the wall-framedness of a heap-dependent expression `e`.

Specification. Viper functions and methods can be annotated with preconditions and postconditions using `requires` and `ensures` respectively. Loop invariants can be specified using `invariant`. Labels can be used to refer to the value of a variable at a given point of the program labelled `l` using `old[l](e)`. For magic wands, `old[lhs](e)` can be used in the RHS to use the value of `e` from the LHS. Viper also features the expected `assert` (resp.

`assume`) to assert (resp. assume) general assertions from IDF (*i.e.* boolean expressions, but also the accessibility assertion `acc`, predicates, and wands).

Magic wands. Wands can be manipulated using two ghost operations: `package` to create a new wand instance, and `apply` to use an available wand. Packaging a wand might require a *proof script*, explaining how the RHS can be obtained from the LHS and the current state. More information is provided in Section 3.3.

Example 2 provides a classic example of magic wand. `list(root)` must hold at the end of the call to `iter_demo`, *i.e.*, it must hold after the loop. However, it is not possible to have `list(root)` and `list(cur)` as invariants at the same time, as they refer to overlapping memory locations. Instead, the strategy is to split the list in two parts: the suffix list `list(cur)`, and the prefix represented by `list(cur) --* list(root)`. This situation was already portrayed in Figure 1. After the loop, `cur = null` and so the wand `list(cur) --* list(root)` can be applied for free as its LHS `list(null)` trivially holds.

Revisiting my goal. I can now state the goal of the internship more precisely: my goal is to design a new operation for magic wands, allowing the solver to prove the well-framedness of a heap-dependent expression which requires permissions from the footprint of a wand. This is analogous to `unfolding` for predicates, which allows doing this, *e.g.*, `x.val` is well-framed in `unfolding list(x) in x.val`. However, unlike predicates which can trivially folded and unfolded, magic wands cannot always be applied (because we might not have a valid LHS) nor easily packaged (they require a user-provided proof script).

3.2 The symbolic execution backend

The SE backend of Viper [14, 15] verifies programs by maintaining a symbolic state while symbolically executing them. To ensure that specifications hold, proof obligations are delegated to an SMT solver.

3.2.1 Symbolic state

The symbolic state consists in three main components:

- a *store*: mapping local program variables to their symbolic SMT representation, known as *terms*;
- a *heap*: a collection of *chunks* representing currently held resources. A resource can be a field access (storing the reference accessing it, its fractional permission and the term representing its value), a predicate, or a magic wand instance;
- a set of *path conditions* (PCs): logic formulas known to be true in the current execution path. They track branch conditions, known facts about terms, function precondition tokens (decribed in Section 3.1), and wand definitions (described below).

3.2.2 Internal procedures

To manipulate this state, the verifier relies on a set of core internal procedures. Note that the verifier is implemented using continuation-passing style (CPS). Instead of returning a value directly, procedures take a continuation function `Q` as an argument and call it with the computed result. In effect, the "returned" values of a function correspond to the arguments of the continuation. CPS allows handling forks easily, for example when branching over an if-then-else construct. Indeed, the continuation can be executed any number of times by the callee (once for each branch), and the caller does not have to maintain a work list of every recorded branch.

I rely on four core internal procedures to present both the existing algorithms to support magic wands and the implementation of my contributions:

`eval(e: Expression, s: State)(Q: Term => ...)`. Evaluates an expression `e` in the state `s`. Because IDF allows heap-dependent expressions, this operation succeeds only if `s` holds the required permissions. If successful, `Q` is called with the resulting symbolic representation (*i.e.*, an SMT `Term`).

`execute(stmt: Statement, s: State)(Q: State => ...)`. Symbolically executes a program statement. If `stmt` induces branching (over an if-then-else statement for example), `Q` may be called multiple times with the updated states corresponding to each branch.

`consume(a: Assertion, s: State)(Q: (Term, State) => ...)`. Asserts that `a` is satisfied in the current state `s`, and removes the resources mentioned by `a`. For example, consuming `acc(x.f)` will check that the permission to `x.f` is held, and consequently remove it. Consuming `x.f ⩾ 0` will simply check the boolean expression is true. If

successful, `Q` is called with the term representing a (*e.g.*, `f1 ⩾ 0` for `x.f ⩾ 0` if `f1` is the symbolic term for `x.f`) and the diminished state.

`produce(a: Assertion, t: Term, s: State)(Q: State => ...)`. Dual operation to `consume`: adds the resources mentioned by `a` into the state `s`, associating them with the term `t`, and assumes `a` is satisfied. `Q` is called with the expanded state.

3.3 Magic wands support in the SE backend

This section reviews the algorithms of both `package` and `apply` operations [44, 45].

3.3.1 Package algorithm

A simplified version of the algorithm is presented in Algorithm 1. At a high level, packaging a wand works as follows: 1. an arbitrary LHS is produced - 2. the proof script is executed, if any - 3. the RHS is consumed.

First, an artificial term representing the LHS is created. Note that a concrete LHS will be provided when applying the wand, but it is unknown when packaging, hence the need for introducing an artificial LHS. Using this term, the proof script is executed, which might fork over several branches, for example if the proof script contains an `if` statement. For each branch, it is checked whether the RHS holds by consuming it. If this is successful, the definition of the wand is subsequently assumed as a path condition: $\forall \mathtt{t_A} : \mathtt{mwsf}(\mathtt{t_A}) = \mathtt{t_B}$, where $\mathtt{t_B}$ is the term representing the RHS, and `mwsf` is a *magic wand snapshot function* (MWSF): an SMT function representing the wand being packaged (one MWSF per packaged wand).

In the SMT encoding, when a term must capture multiple memory locations, it is represented as a nested tuple, forming a binary tree. Individual components of this captured state are then extracted using the `First` and `Second` projection operators.

Note that the details of `handlingPathConditions` are quite intricate and heuristic, and so they are not presented in this report.

```
def package(w: Wand, s: State, Q: (Chunk, State) => ...) => ...:
  tA = freshTerm()
  produce(w.lhs, tA, s)(sA =>
    execute(proofScript)(sBranch =>
      consume(w.rhs, sBranch)((tB, sB) =>
        chunk = createWandChunk(tA, tB)
        sR = handlePathConditions(sB)
        sR.assume(Forall(tA, chunk.mwsf(tA) = tB))
        Q(chunk, sR)
      )
    )
  )
```

Algorithm 1: Simplified existing package algorithm.

Footprint selection. The footprint are the permissions that are taken from the context to enable running the proof script and to get the RHS in the end. The footprint selection algorithm [44] (*i.e.*, how to choose which these permissions) does not appear explicitly in Algorithm 1. Yet, this is an essential part of packaging wands. The SE verifier implements a greedy approach to consume resources (either when executing the proof script, or consuming the RHS): it first attempts to extract permissions from the LHS, resorting to the external heap when that fails. However, this still involves some heuristics, and there is no reason to prefer one heuristic over the others in general (example in Appendix A).

Examples of wand package. Example 2 features two package statements `package list(cur) --* list(root)`:

- before the while loop: `cur` and `root` are equal, so the wand has an empty footprint and is defined as $\forall \mathtt{t_A} : \mathtt{mwsf}_{\text{initial}}(\mathtt{t_A}) = \mathtt{t_A}$, *i.e.*, when applying the wand, RHS `list(root)` will be exactly the provided LHS `list(cur)`;
- at the end of the loop: note that we have two wands here: the initial wand assumed at the beginning of the loop, and the wand packaged at the end of the loop. The footprint of the latter is `acc(prev_cur.val) && acc(prev_cur.next)` plus the footprint of the initial wand (intuitively, this is the prefix of the list before `prev_cur`), and the registered definition is $\forall \mathtt{t_A} : \mathtt{mwsf}_{\text{loop}}(\mathtt{t_A}) = \mathtt{mwsf}_{\text{initial}}(\mathtt{t}_{\mathtt{prev\_cur}} :: \mathtt{t_A})$, where $\mathtt{t}_{\mathtt{prev\_cur}}$ denotes the term for the first element of the list starting in `cur` at the *beginning* of the loop.

Not all wands are packaged (equal). All wands owned in a state might not have an associated definition in the path conditions. Indeed, wands that have been obtained through a precondition or an invariant for example do not come with a definition. Intuitively, definitions cannot be inferred from the wand's syntax as there is no unique footprint selection strategy (two wands syntactically equal could have different footprints and definitions).

3.3.2 Apply algorithm

The algorithm is presented in Algorithm 2.

```
def apply(w: Wand, s: State, Q: State => ...) => ...:
  consume(w, s)((mwsf, s1) =>
    consume((w.lhs, s1)((tA, s2) =>
      produce(w.rhs, mwsf(tA), s2)(s3 =>
        Q(s3)
      )
    )
  )
```

Algorithm 2: Applying a wand simply consumes the wand itself and its LHS, and provides the RHS in exchange.

Temporarily apply a wand. Similarly to `unfolding ... in ...` for predicates, `applying ... in ...` can be used to temporarily apply a wand in an expression [45, 46]. Note that unlike predicates which are trivial to fold back, wands are difficult to package. After an `applying ... in ...` expression, the initial state is simply reverted.

4 Sound packaging of path conditions

Although this was not initially a goal of my internship, I discovered (while implementing a new Viper construct described in Section 5) both a soundness and a completeness issue in the algorithm to package wands. Another soundness issue was subsequently discovered by Jonáš, related to *state consolidation*. In this section, I review these issues and present a sound encoding of wands. Finally, I evaluate this new encoding against the test suite and benchmark it.

4.1 Packaging and path conditions

The issues. The existing implementation was wrong in two distinct ways, resulting in both unsoundness and incompleteness:

- the continuation was executed for each branch of the package separately, *i.e.*, the rest of the verification was performed under the assumption that the wand was packaged using this one branch only. Because a wand can be applied multiple times using the `applying`, it results in an unsoundness as demonstrated in Example 3. Intuitively, we cannot assume at the time of the package which branch will be used when applying the wand.
- path conditions from within the package were not properly propagated outside of it[4], *e.g.* branch conditions were simply discarded. Combined with the first issue, this can cause an incompleteness. A program exhibiting this issue can be found in Appendix B.1.

The fix. Instead of calling the continuation once per recorded branch from within the package, the continuation is only called once with a state that summarizes all the different recorded states (one per branch). This way, no specific branch is assumed for the rest of the verification. Such a merged state can be obtained by joining heaps and path conditions. I first detail the latter.

A first attempt at joining path conditions. A natural idea is to conditionalize the path conditions of each branch by their respective branch condition, leading to the following joined path conditions:

$$\forall \mathtt{t_A} : \bigwedge_i \mathtt{bc}_i \Rightarrow \mathtt{mwsf}(\mathtt{t_A}) = \mathtt{def}_i \wedge \mathtt{pc}_i$$

where i iterates over recorded branches, $\mathtt{bc}_i$ denotes the branch condition, $\mathtt{mwsf}(\mathtt{t_A}) = \mathtt{def}_i$ is the magic wand definition, and $\mathtt{pc}_i$ denotes all the other path conditions on that branch (*e.g.* function precondition tokens). Unfor-

[4] As mentioned in Section 3.3, the handling of path conditions was intricate and the full description is beyond the scope of this report.

```
field b: Bool
field f: Int
field g: Int

method unsoundness_applying_using_different_branches(x: Ref)
  requires acc(x.b) && acc(x.f) && acc(x.g)
{
    define W acc(x.b) --* acc(x.b) && (x.b ? acc(x.f) : acc(x.g))
    package W

    x.b := false
    assert applying (W) in acc(x.g)

    x.b := true
    assert applying (W) in acc(x.f)

    assert false
}
```

Example 3: A program exhibiting an unsoundness by applying a wand several times. The SE engine was branching at the end of the package, running the verification two times: assuming the wand is `acc(x.f) --* acc(x.b) && acc(x.f)` (resp. `acc(x.f) --* acc(x.b) && acc(x.g)`) and that only `acc(x.f)` (resp. `acc(x.g)`) has been packed in the footprint. Neither of these two wands is correct, and both `acc(x.f)` and `acc(x.g)` should be packed in the footprint.

tunately, this approach does not work as branch conditions might depend on the RHS of the wand. For example, in Example 3 we branch on the RHS value of `x.b`, which is $\texttt{First}(\texttt{mwsf}(\texttt{t}_\texttt{A}))$. This introduces a circular reasoning issue, preventing the solver from ever accessing the definition.

A second attempt at joining path conditions. To circumvent this problem, an intuitive workaround is to propose the following joined path conditions:

$$\forall \texttt{t}_\texttt{A} : \bigvee_i \texttt{bc}_i \wedge \texttt{mwsf}(\texttt{t}_\texttt{A}) = \texttt{def}_i \wedge \texttt{pc}_i$$

which indicates that the path conditions from (at least) one branch hold true.

However, path conditions from within a package statement should not be propagated carelessly outside of the package. Within the package, it is assumed that a valid LHS exists, which might not be the case: `false --* false` for example. Packaging this wand would be encoded as $\forall \texttt{t}_\texttt{A} : \texttt{mwsf}(\texttt{t}_\texttt{A}) = () \wedge \texttt{false}$. The `false` should not be propagated outside of the package. Note that this vacuous wand is completely valid, it will simply never be applied.

Joining path conditions. My proposed solution is to leverage a new *magic wand token*, similar to how Viper functions are encoded at the SMT level. A token $\texttt{MW\_token}(\texttt{w}, \texttt{t})$ is tied to a magic wand and one concrete LHS, and can be interpreted as "t is a valid LHS for the wand w". Then, we can encode the overall path condition resulting from the package as the following formula:

$$\forall \texttt{t}_\texttt{A} : \texttt{MW\_token}(\texttt{w}, \texttt{t}_\texttt{A}) \implies \bigvee_i \texttt{bc}_i \wedge \texttt{mwsf}(\texttt{t}_\texttt{A}) = \texttt{def}_i \wedge \texttt{pc}_i$$

which indicates that, *for each valid LHS* $\texttt{t}_\texttt{A}$, the path conditions of (at least) one branch hold true.

When applying a wand w with a concrete LHS t, a token $\texttt{MW\_token}(\texttt{w}, \texttt{t})$ is emitted after checking that the LHS holds (and thus is valid), allowing the solver to instantiate the quantifier and get $\bigvee_i \texttt{bc}_i[\texttt{t}_\texttt{A} := \texttt{t}] \wedge \texttt{mwsf}(\texttt{t}) = \texttt{def}_i[\texttt{t}_\texttt{A} := \texttt{t}] \wedge \texttt{pc}_i[\texttt{t}_\texttt{A} := \texttt{t}]$. Furthermore, if some information is known about t, only one branch condition might be compatible. In that case, the solver can retrieve the appropriate magic wand definition and path conditions.

Joining heaps. Heaps from several branches can be joined by conditionalizing the permissions of chunks by their respective branch condition. Conditional permissions were already supported in Viper. Table 2 shows how heaps from several branches can be joined, reusing the example given in Example 3.

However, this final joined heap features permissions depending on t_A. Yet, t_A does not have any meaning outside of the package: is this an issue? Well, it is not: the solver will just never be able to prove that $\texttt{First}(\texttt{mwsf}(t_A))$ (or its negation) holds true[5]. This successfully prevents the verifier from accessing both `x.f` and `x.g`, which is what we expect.

Branch	Branch condition	Remaining heap	Conditionalized heap	Final joined heap
`x.b`	$\texttt{First}(\texttt{mwsf}(t_A))$	$(\texttt{x.g} \rightarrow \ldots \ \#\ 1)$	$(\texttt{x.g} \rightarrow \ldots \ \#\ \texttt{First}(\texttt{mwsf}(t_A))?\,1:0)$	$(\texttt{x.g} \rightarrow \ldots \ \#\ \texttt{First}(\texttt{mwsf}(t_A))?\,1:0,$
`!x.b`	$!\texttt{First}(\texttt{mwsf}(t_A))$	$(\texttt{x.f} \rightarrow \ldots \ \#\ 1)$	$(\texttt{x.f} \rightarrow \ldots \ \#\ !\texttt{First}(\texttt{mwsf}(t_A))?\,1:0)$	$\texttt{x.f} \rightarrow \ldots \ \#\ !\texttt{First}(\texttt{mwsf}(t_A))?\,1:0)$

Table 2: Joining the heaps from the different branches recorded in the example of Example 3.

4.2 State consolidation issue

I briefly cover the other unsoundness issue discovered by Jonáš.

State consolidation. State consolidation performs various simplifications to keep the state as simple as possible for further use and SMT queries. This is not only a matter of performance: because the SE engine consumes chunks greedily by default, not performing these simplifications can result in incompleteness. To prevent these issues, the SE engine provides an experimental feature named SE-PC [13] (for *partial* heaps *combined*), which allows consuming chunks in a more complete way. The SE-PC feature can be enabled by passing a command-line flag. However, this experimental feature did not support packaging a wand.

The issue. Such simplifications include merging chunks when they are proven to be equal (*e.g.*, two references proven equal accessing the same field). Example 4 provides an example of a program triggering the unsoundness. On the example, the consolidator merges the chunks `x.f` and `y.f` during the package, leading to have a permission `2/1` to that chunk (which is equivalent to false as the maximum valid permission in SL is `1`). Unfortunately, this side effect remains after the packaging of the wand, allowing us to prove false after the package.

```
method unsound_consolidation(x: Ref, y: Ref)
    requires acc(x.f)
    requires acc(y.f)
{
    package x == y --* false
    assert false
}
```

Example 4: Program exhibiting an unsoundness of the SE verifier by using state consolidation in a package.

The fix. My proposed fix is simply not to perform any side effects on the external heaps (not packed inside the footprint) during state consolidation. I also added support of wand packages for the SE-PC procedure. Incompleteness that would result from not performing side effects in the consolidator are fixed by enabling the SE-PC flag.

4.3 Evaluation

Against the test suite. The new encoding was validated against the SE verifier test suite. It comprises 178 test files featuring wands, 166 of which still pass without issue. Out of the 12 currently failing tests, 7 involve quantified wands which I did not support. The remaining 5 failing tests are briefly described in Appendix B.2. Only one of them truly exhibits a completeness regression in the current implementation: two others can be verified using the SE-PC flag, and the remaining two should never have passed in the first place.

I also added two new regression test files to the test suite to check for the newly identified issues (`package_path_conditions.vpr`[6] and `consolidation_outer_heaps.vpr`[7]).

[5] There is a slight engineering tweak: every joined heap must use a fresh (and thus different) t_A. This matters because of some side effect, hinted at in the next footnote, which would effectively reintroduce the same unsoundness. The full description of this is painful and beyond the scope of this report.

[6] https://github.com/remigerme/silver/blob/rgerme-mw-attached-facts/src/test/resources/wands/regression/package_path_conditions.vpr

[7] https://github.com/remigerme/silver/blob/rgerme-mw-attached-facts/src/test/resources/wands/regression/consolidation_outer_heaps.vpr

Benchmark. Both the old and new encodings were benchmarked using hyperfine [47] on large real-world verified software, specifically an Internet router [48] and a library for verifying security protocol implementations [49], both written in Go and verified with Gobra. The new encoding shows equivalent performance to the old one, as displayed in Appendix B.3.

5 Modular Functional Specifications for Wands

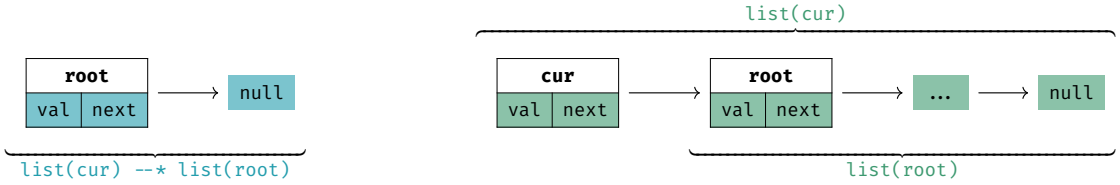
This section introduces my first contribution to the actual research problem: a new Viper construct enabling to specify functional properties on top of an existing wand rather than in the wand directly.

5.1 Overview of difficulties

I first recall the goal of the internship and review some initial difficulties, that led me to consider a simpler problem at first.

Goal of the internship. Recall that, on one hand, IDF allows accessing heap-dependent expressions if the state holds the appropriate permissions. On the other hand, magic wands contain permissions in their footprint. However, these permissions are not directly accessible, for reasons described below. The goal is to propose a construct making these permissions accessible, enabling to write expressions that depend on them.

Wands are implicit. The core issue with wands is that they *implicitly* describe some resources contained in their footprint. Indeed, the footprint of a given wand is never written explicitly. Moreover, it cannot be inferred from the wand's syntax as a wand can have an entire set of valid footprints, not just one, some of which are less intuitive. For example, `list(cur) --* list(root)` could be the reasonable wand presented in Figure 1, yet, it could also be one of the less intuitive wands displayed in Figure 4, or even a wand that is vacuously true as explained in Section 2.



(a) The wand RHS might not depend on its LHS. (b) The wand RHS might depend entirely on its LHS.

Figure 4: Two examples of unintuitive `list(cur) --* list(root)` instances.

Thus, we would need a way to constrain a wand, to convey our intuition of what is the expected shape of the wand to the verifier and guarantee that a given permission is stored inside the footprint. All my (inconclusive) efforts in this direction eventually converged towards the traditional list segment predicate:

```
predicate lseg(cur: Ref, end: Ref) {
  cur ≠ end ⟹ acc(cur.val) && acc(cur.next) && lseg(cur.next, end)
}
```

This predicate has the benefit of explicitly specifying the inductive shape, meaning that the verifier knows directly which permissions are associated with it. If my proposed solution replicates the list segment predicates with extra steps, its utility is questionable.

On mutability. A motivating use case for the construct was to enable developers to write a `merge_sort` method using wands. However, another issue arises from mutability. The footprint of a wand obviously cannot be mutated in place, consider `true --* acc(x.f) && constraint_on_value(x.f)` that has `acc(x.f)` in its footprint: modifying `x.f` would potentially invalidate the wand. Thus, a construct which allows mutating the footprint of the wand would have to consume the wand. For this construct to be useful, we cannot simply lose the wand in the process. We could imagine that it also yields a new "repackaged" wand, consuming the initial wand to yield a new one instead. Yet, my investigations in this direction did not yield anything valuable.

Towards my first contribution. In reaction to these initial difficulties, I first aimed at what seemed a lower-hanging fruit: improving the practical convenience of working with wands. Feedback from verifying real-world

codebases indicates that verification happens in phases: memory safety is proven first, then functional behavior is specified incrementally. When adding functional specifications, magic wands can quickly become difficult to read and maintain. My first contribution makes this process smoother by introducing *attached facts*.

5.2 Attached facts by example

Attached facts provide a modular mechanism for specifying functional properties on top of a wand carrying permissions, by "attaching" boolean expressions to the wand. Attached facts are specified and verified independently from one another, allowing users to decompose a wand specifying several distinct properties into independent pieces, as demonstrated in Figure 5. They can be `required`, `ensured`, `asserted`, or listed as `invariant` like any other boolean expression.

```
        list(cur)
        && len(cur) == len_cur
        && elems(cur) == elm_cur[0..|elm_cur|]
    --* list(root)
        && len(root) == old(len(root))
        && elems(root) == old(elems(root))[0..|old(elems(root))|]
```

(a) The wand from Example 2, enhanced with specification on the elements.

```
list(cur) --* list(root)

attached old[lhs](len(cur)) == len_cur
         ==> len(root) == old(len(root))
      to list(cur) --* list(root)

attached old[lhs](elems(cur)) == elm_cur[0..|elm_cur|]
         ==> elems(root) == old(elems(root))[0..|old(elems(root))|]
      to list(cur) --* list(root)
```

(b) The same specification using attached facts. Each attached fact is specified and verified independently.

Figure 5: Comparison between a classical wand (packing all specifications inside) and a wand using attached facts.

In addition to providing a smoother verification experience, Example 5 presents other benefits of using attached facts over traditional monolithic wand:

- writing wands in such a modular way is not only a matter of convenience, it is actually a gain in expressiveness, as shown by the function that can only be called if using attached facts in Example 5a or the possibility to *weaken* a wand in Example 5b;
- attached facts do not have to be specified explicitly in the package instruction[8], as they can be deduced at any point afterwards as long as the definition of the wand is accessible[9]. This leads to simpler wands and `package` statements, as the functional specification does not have to be explicitly written.

5.3 Semantics of attached facts

I now present the semantics of this new construct in two steps. First, I describe a generic evaluation procedure, that properly sets the stage to define the core semantics.

First step: generic evaluation procedure. To evaluate `attached e to w`, the expression `e` must be properly framed in the symbolic state. The first procedure handles this framing. Intuitively, `attached e to A --* B` evaluates the boolean expression `e` as it would be evaluated in `A --* B && e`, thus allowing `e` to use permissions from both the LHS (using an `old[lhs]( ... )`) and the RHS. An outline of this algorithm is presented in Algorithm 3.

The algorithm is parametrized by a function `evalExp`, which specifies the actual semantics of the `attached` construct. It takes the computed SMT term `tE` for `e` as an input (and other relevant arguments such as the magic wand term `mws`) and returns the computed SMT term for the whole `attached e to w` expression.

[8] Attached facts do not need to be specified explicitly, although they can be using the `attaching` keyword in a package statement: `package w attaching f1 attaching f2 ... { proofScript }`. This is a historical remain from when I was still playing and exploring new constructs.

[9] And for properties whose proofs require some lemma, as long as the lemma is called in the proof script (like for traditional wands).

```
function f( ... ): ...
  requires A && P1l --* B && P1r

method m( ... ) {
  package A && P1l && P2l --* B && P1r && P2r
  f( ... ) // error: wand not found
}
```

Without attached facts: the call to `f` fails as the packaged wand is not the expected one.

```
function f( ... ): ...
  requires A --* B
  requires attached old[lhs](P1l) ⟹ P1r to A --* B

method m( ... ) {
  package A --* B
  f( ... ) // works fine
}
```

With attached facts: the call succeeds, and the packaged wand does not even have to specify the functional properties as they can be deduced afterwards.

(a) Side-by-side comparison of calling a function requiring a wand as a precondition.

```
method weaken(x: Ref)
  requires true --* acc(x.f)
  requires attached x.f ⩾ 42 to true --* acc(x.f)
  ensures  true --* acc(x.f)
  ensures  attached x.f ⩾ 0  to true --* acc(x.f)
{ /* We don't need to do anything to prove the postcondition. */ }
```

(b) Attached facts can be weakened for free.

Example 5: Two examples showcasing the expressiveness gain brought by attached facts.

```
def evalAttachedGeneric(e: Exp, w: Wand, s: State,
                        evalExp: (State, MWSF, Var, Term) => Term,
                        Q: (State, Term) => ...) => ...:
  consume(w, s)((mws, sW) =>
    tA = freshTerm()
    produce(w.lhs, tA, sW)(sA =>
      produce(w.rhs, mws.apply(tA), sA)(sB =>
        eval(e, sB)(tE =>
          tRes = evalExp(sB, mws, tA, tE)
          Q(sE, tRes)
        )
      )
    )
  )
```

Algorithm 3: A generic evaluation procedure (simplified) for `attached e to w`, parametrized by the `evalExp` function which specifies the semantics of the construct.

Second step: specifying the semantics. Intuitively, the semantics for `attached e to w` is "for any valid LHS used to apply the wand, the boolean expression e holds". This naturally leads to the following semantics, reusing the magic wand token function to encode valid LHS for a given wand:

```
def evalExpAttachedFacts(s: State, mws: MagicWandSnapshot,
                         tA: Term, tE: Term) => Term:
  Forall(tA, Implies(mws.token(tA), tE))
```

Alternatively, this can be presented informally as:

$$[\![\texttt{attached e to w}]\!] \stackrel{\text{def}}{=} \forall \texttt{t}_\texttt{A} : \texttt{MW\_token}(\texttt{w}, \texttt{t}_\texttt{A}) \Rightarrow [\![\texttt{e}]\!]$$

where $\texttt{t}_\texttt{A}$ denotes the hypothetical valid LHS[10]. Note that $[\![\texttt{e}]\!]$ depends on this value $\texttt{t}_\texttt{A}$.

Why does it work? To understand how this encoding works in practice, I quickly review two key situations:

- when the wand `w` is applied with a concrete LHS $\texttt{t}_\texttt{LHS}$, a token $\texttt{MW\_token}(w, \texttt{t}_\texttt{LHS})$ is emitted. If `attached e to w` holds true, the SMT solver can instantiate the forall quantifier from above and conclude that $[\![\texttt{e}]\!][\texttt{t}_\texttt{A} := \texttt{t}_\texttt{LHS}]$ holds;
- when trying to prove that an attached fact holds, the SMT solver has to prove that e holds for any LHS, assuming the token holds. The solver can then use all the knowledge that can be derived from this token, such as the

[10] Any other binding variable x could be used instead of $\texttt{t}_\texttt{A}$, but we would then need to replace every occurrence of $\texttt{t}_\texttt{A}$ in $[\![e]\!]$ by x.

wand definition (if the wand has been packaged and not obtained through a precondition for example) or other attached facts that were previously shown to hold.

5.4 Evaluation

This section demonstrates the use of attached facts in three different situations.

Application to classic examples. The initial use cases for attached facts were demonstrated by rewriting existing examples from papers about magic wand automation [4, 50], as well as proposing new ones (see Appendix C). These examples showcase how attached facts enable more modular specifications.

Application to verified real-world software. I also explored applications to larger existing codebases, namely the Internet router [48] and the verification library [49] used to benchmark the new encoding (Section 4.3). However, almost all wands in these projects are of the form `predicate_data_structure_1( ... ) --* predicate_data_structure_2( ... )`, which causes two issues. First, the permissions and functional specifications are sometimes deeply entangled within (nested) predicates, making it more difficult to rewrite the specifications using attached facts. It can be argued that specifications were written this way because attached facts were not available, and that they still might have proven useful if writing specifications from scratch. The second issue is that predicates sometimes model a data structure by bundling both its permissions and invariants. In such a case, it is not desirable to separate the permissions from their associated invariants.

Application to Prusti. However, Prusti, the Viper frontend for Rust, appears to be a very promising target. Indeed, magic wands naturally emerge when encoding Rust's borrow, even if they were not specified explicitly by the user, as shown in Example 6. The functional specification provided by the user then needs to be integrated within the wand, which makes an ideal use case for attached facts as we naturally face a clear separation between permission reasoning and functional properties. I demonstrated this potential on several files[11], including a binary search tree implementation [6], by manually rewriting the Viper file outputed by Prusti. Investigating further how the attached facts can be integrated in Prusti directly is left as future work. One expected consequence is to allow more modular specifications for Prusti end users.

```
struct Pair {
  fst: i32,
  snd: i32,
}

fn borrow_first(p: &mut Pair) → &mut i32 {
  &mut p.fst
}
```

```
field fst: Int
field snd: Int

predicate pair(x: Ref) {
  acc(x.fst) && acc(x.snd)
}

method borrow_first(p: Ref)
  requires pair(p)
  ensures  acc(p.fst)
  ensures  acc(p.fst) --* pair(p)
```

Example 6: Side-by-side comparison of a Rust program and its idealized Viper encoding. A magic wand is naturally introduced to model the borrow of the first element: when it ends, full permission to the whole pair is retrieved.

6 Accessing the Footprint

I showed how we can "attach" boolean expressions to wands. Can we attach arbitrary expressions? And what about the initial goal, accessing permissions hidden inside the footprint of a wand to prove well-framedness of heap-dependent expressions? This section investigates these questions, providing both a meaning to "attached expressions" and a theoretical sufficient condition to check their well-definedness. Yet, automating the verification of this condition and implementing it in Viper remains future work.

Why is that so different from attaching facts? Before detailing the semantics of attached expressions, it is important to highlight some fundamental differences with attached facts.

First, the fact that the latter are restricted to boolean expressions might seem artificial, but it is not. Indeed,

[11] All files available at https://github.com/remigerme/silicon/tree/mw-attached-facts/remi-demo.

specifications are boolean assertions: we can require a boolean to hold true as a precondition, or require a resource using `acc` or a predicate, whereas it does not make sense to "require an integer as a precondition".

Second, they serve different purposes in different context. Attached facts provide a way to retrieve functional properties *once the wand has been applied*, whereas attached expressions aim at *accessing some heap-dependent expression without applying the wand*, the expression being well-framed in the footprint of the wand. If we apply the wand, then we would trivially get back the permissions to these values from the RHS[12], so it is important that the construct works without any available valid LHS.

A quick example of attached expression. Consider a pair of integers, and the magic wand `acc(x.snd) --* pair(x)`, *i.e.*, the promise that the wand can be applied to exchange permission to the second element for permission to the whole pair. The intuitive footprint of this wand is the permission to the first element `acc(x.fst)`. The goal of attached expressions is to ensure that `x.fst` is well-framed, for example to assign it to another variable or use it in specification.

First attempt at attached expressions. A natural idea is to remember during the package which permissions are contained in the footprint, and allow users to access them later on. However, this would require significant extensions to Viper as the handling of these resources would have to be special (*e.g.*, only read-only operations should be allowed even if the full permission to a field is inside the footprint, because of mutability issues mentioned in Section 5). Moreover, this approach is less expressive than the one described below, as it only applies to wands that have been packaged, and only allows accessing explicit resources (fields, predicates, or wands) and not arbitrary heap-dependent expressions (see Example 8). For these reasons, it was discarded in favor of the approach detailed in the rest of this section.

6.1 Semantics of attached expressions

A fundamental limitation of wands that were not packaged but obtained through a precondition or an invariant is that nothing is known about them, and it is not possible to rely on the wand's syntax to infer the footprint. To overcome this limitation, I propose a helper construct, indicating that an expression `e` is well-framed in the footprint of a wand `w`. This helper construct can then be used along with the wand in pre and postconditions, and invariants. Then, for wands that have been packaged, the goal is to be able to prove the well-framedness of such expressions `e` without having to assume the helper construct.

This approach leads to the following constructs[13]:

- `attachedexp_valid e to w`, whose value is a boolean indicating whether `e` is valid in the footprint of `w`;
- `attachedexp e to w`, whose value is `e` if it was proven well-framed, or an error otherwise.

Notably, this separation between the well-framedness of an expression and its value is idiomatic in IDF. As discussed in Section 2, this mirrors how IDF splits the SL points-to assertion into the `acc( ... )` assertion and the value itself.

A well-framedness condition. Intuitively, only expressions that are well-framed in the RHS but do not depend on the LHS should be allowed. Indeed, accessing an expression that depends on the LHS without providing an LHS does not make sense (*e.g.* `x.snd` is not well-framed/does not have a value in `acc(x.snd) --* pair(x)`). This independence on the LHS can easily be verified automatically: we evaluate `e` using the generic evaluation procedure from Section 5 (*i.e.*, framed as in `A --* B && e`) with two artificial LHS t_1 and t_2, and check that both resulting terms can be proven equal. Reusing the formalism from Section 5, I propose the following semantics:

$$[\![\texttt{attachedexp\_valid e to w}]\!] := \forall t_1, t_2 : \texttt{MW\_token}(\texttt{w}, t_1) \wedge \texttt{MW\_token}(\texttt{w}, t_2) \Rightarrow [\![\texttt{e}]\!][t_A := t_1] = [\![\texttt{e}]\!][t_A := t_2]$$

Once again, the tokens model the fact that t_1 and t_2 are valid LHS for the wand `w`. I will subsequently show that this condition alone is not sufficient, and introduce another condition after reviewing examples.

[12] The RHS does not have to give back all permissions captured in the footprint in general. It is possible to create a leaking wand, that captures some permissions in its footprint yet does not provide them back in the RHS.

[13] Relevant quote: "There are only two hard things in Computer Science: cache invalidation and *naming things*."[14], courtesy of Phil Karlton to the best of my knowledge.

[14] And off-by-one errors, of course.

Evaluating the expression. Following my intuition, I propose the following semantics, once again reusing the generic evaluation procedure and formalism from Section 5:

$$[\![\texttt{attachedexp e to w}]\!] := [\![\texttt{e}]\!] \text{ using the condition above to check well-framedness}$$

However, $[\![e]\!]$ still syntactically depends on $\texttt{t}_\texttt{A}$, even though it was proven not to depend on the specific value of the considered LHS by checking the formula above. Also, $[\![e]\!]$ is not guarded by any token this time, as it should be accessible even without having a valid LHS. This raises an issue, as the solver has no way to retrieve the wand definition, guarded by a token.

In the next sections, I analyze these issues throughout two examples[15]. They both demonstrate the potential of attached expressions and highlight a current limitation. I then present a theoretical solution to this limitation, although the automation of this solution is left as future work.

6.2 Attached expressions by example

6.2.1 Attaching the first element of a pair

The first example, provided in Example 7, demonstrates how the first element of a pair can be attached to the wand `acc(x.snd) --* pair(x)`. Resulting symbolic states are annotated after each instruction.

```
field fst: Int
field snd: Int
predicate pair(x: Ref) { acc(x.fst) && acc(x.snd) }

method set_fst_to_zero_and_package(x: Ref)
  requires acc(x.fst)
  ensures  acc(x.snd) --* pair(x)
  ensures  attachedexp_valid unfolding pair(x) in x.fst to acc(x.snd) --* pair(x)
  ensures  (attachedexp unfolding pair(x) in x.fst to acc(x.snd) --* pair(x)) == 0
{
  // Symbolic state: S: (x → x@1), H: (x@1.fst → fst@2), PC: ()
  x.fst := 0
  // Symbolic state: S: (x → x@1), H: (x@1.fst → fst@3), PC: (fst@3 = 0)
  package acc(x.snd) --* pair(x) { fold pair(x) }
  // Symbolic state: S: (x → x@1), H: (), PC: (fst@3 = 0,
  //                 ∀ tA : MW_token(w, tA) ⟹ mwsf(tA) = (fst@3, tA))
  // Example: attachedexp can also be used in actual code, not only specification
  var f: Int := attachedexp unfolding pair(x) in x.fst to acc(x.snd) --* pair(x)
  // Symbolic state: S: (x → x@1, f → First(mwsf(tA))), H: (), PC: ( ... )
  assert f == 0
}
```

Example 7: Attaching the first element of a pair to a magic wand.

When using `attachedexp` or `attachedexp_valid` either in the code or the specification, $[\![\texttt{unfolding pair(x) in x.fst}]\!]$ evaluates to $\texttt{First}(\texttt{mwsf}(\texttt{t}_\texttt{A}))$[16], where $\texttt{t}_\texttt{A}$ is the dangling artificial LHS introduced by the generic evaluation procedure. I show that the helper construct works fine, whereas actual attached expressions face limitations.

For `attachedexp_valid`. The postcondition verifies successfully, *i.e.*, `unfolding pair(x) in x.fst` can be proven independent of the LHS of `acc(x.snd) --* pair(x)`. Indeed, following the semantics from above, for all $\texttt{t}_1$, $\texttt{t}_2$, such that $\texttt{MW\_token}(w, \texttt{t}_1)$ and $\texttt{MW\_token}(w, \texttt{t}_2)$ hold, we have:

$$\begin{aligned}
[\![\texttt{unfolding pair(x) in x.fst}]\!][\texttt{t}_\texttt{A} := \texttt{t}_1] &= \texttt{First}(\texttt{mwsf}(\texttt{t}_\texttt{A}))[\texttt{t}_\texttt{A} := \texttt{t}_1] \text{ by evaluation using the generic procedure} \\
&= \texttt{First}(\texttt{mwsf}(\texttt{t}_1)) \\
&= \texttt{First}(\texttt{fst@3}, \texttt{t}_\texttt{A}) \qquad \text{by def. of the wand (requires } \texttt{MW_token}(w, \texttt{t}_1)) \\
&= \texttt{fst@3}
\end{aligned}$$

[15] These examples and more can be found at https://github.com/remigerme/silver/tree/rgerme-mw-attached-facts/src/test/resources/wands/attached.

[16] This is a function of the RHS, which is expected as I recall that e is framed as it would be in `A--* B && e`.

The same reasoning can be applied to $[\![\texttt{unfolding pair(x) in x.fst}]\!][\texttt{t}_A := \texttt{t}_2]$, which concludes the proof. Note that the definition of the wand is needed, and thus, a token is needed to access it.

For `attachedexp`. The assertion `f == 0`, as well as the postcondition asserting the same equality, will fail to verify. Indeed, `f` evaluates to $\texttt{First}(\texttt{mwsf}(\texttt{t}_A))$, but this time, there is no $\texttt{MW\_token}(w, \texttt{t}_A)$ in the context enabling to access the definition of the wand to establish that $\texttt{First}(\texttt{mwsf}(\texttt{t}_A)) = \texttt{fst@3}$ ($= 0$ thanks to the path conditions).

In this first case, it suffices to know the wand definition to successfully use attached expressions. A tentative lightweight solution is discussed in Appendix D.

The issue: vacuous wands. The key issue attached expressions are facing are vacuous wands, which already were the reasons why tokens were introduced in the first place. For example, the wand `acc(x.snd) --* pair(x)` could also have been packaged with `acc(x.snd)` in the footprint, making the wand vacuously true. In that case, with the current semantics, `attachedexp ... x.fst to w` will be an arbitrary value. However, this is a language design question, as it would also be possible to imagine requiring the user of an attached expression to prove that the wand of interest is not vacuously true. We will come back to that idea after a second example.

6.2.2 Attaching the length of a prefix list

The second example, provided in Example 8, shows that an expression which is not just a field access can also be attached. Indeed, the attached expression is the length of the prefix list implicitly described by the wand, obtained through a computation that syntactically depends on $\texttt{t}_A$ (here, representing `list(cur)`) but that we expect to be able to prove independent from it. It also provides an example of how inductive proofs can be done using attached expressions.

```
define w list(cur) --* list(root)
var i: Int := 0
while (cur ≠ null)
  ...
  invariant w
  invariant attachedexp_valid len(root) - old[lhs](len(cur)) to w
  invariant (attachedexp len(root) - old[lhs](len(cur)) to w) == i
{
  ...
  cur := cur.next
  i := i + 1
  ...
}
```

Example 8: Attaching the length of the prefix list described by the wand - this is not a simple field access.

Proving that the attached expression at the end of the loop is valid and is equal to `i` requires a bit more work than the `pair` example. The full proof is available in Appendix E. The essential insight is that, this time, accessing the wand definition is not sufficient: we also need to instantiate the quantifier in the definition of `attachedexp_valid` with valid tokens for any of the proofs to pass. The same situation as the `pair` example can be witnessed:

- the `attachedexp_valid` invariant verifies successfully, as the helper construct provides the required tokens;
- the `attachedexp` invariant fails to verify, due to two tokens missing: $\texttt{MW\_token}(\texttt{w}_{\texttt{old}}, \texttt{t}_A)$ and $\texttt{MW\_token}(w, \texttt{t}_B)$ where:
  - $\texttt{w}_{\texttt{old}}$ (resp. `w`) is the wand assumed at the beginning of the loop (resp. packaged during the loop);
  - $\texttt{t}_A$ (resp. $\texttt{t}_B$) is the artificial LHS introduced when assuming (resp. asserting) the `attachedexp` invariant at the beginning (resp. end) of the loop.

We do not know anything about $\texttt{t}_A$ and $\texttt{t}_B$, they just witness the existence of such tokens, *i.e.*, $\exists\ \texttt{t} : \texttt{MW\_token}(w, \texttt{t})$ for $w = \texttt{w}$ or $w = \texttt{w}_{\texttt{old}}$. This can be interpreted as the existence of one (any) valid LHS for these wands. In this case, to use attached expressions, we need to prove the existence of a valid LHS for the wand, *i.e.*, prove the wand is not vacuously true.

Question. How can we soundly prove the existence of a valid LHS?

For the initial wand. Note that no wand definition is known for the initial wand. I believe the only reasonable solution is to introduce another specification construct along the lines of `wand_satisfiable list(cur) --* list(root)`, allowing us to assume the token from an invariant. This follows the usual inductive proof scheme.

For the packaged wand. For the wand that was packaged however, we have to prove that a token can be emitted, *i.e.*, prove that a valid LHS can exist, *i.e.*, that the hypothetical `wand_satisfiable list(cur) --* list(root)` would be preserved.

6.3 Proving the existence of a valid LHS

Note that this section is still exploratory, and no implementation beyond this point was provided.

Appliability. Recall that formally a heap σ satisfies a wand $A \mathrel{-\!\!*} B$ if $\forall \sigma_A : \sigma_A \perp \sigma \wedge \sigma_A \vDash A \Rightarrow \sigma_A \uplus \sigma_B \vDash B$. We say that a wand is *appliable* if there exists a heap satisfying the LHS of this implication, *i.e.*, $\exists \sigma_A : \sigma_A \perp \sigma \wedge \sigma_A \vDash A$, *i.e.*, if the wand is not vacuously true. Checking that a wand is appliable suffices to soundly emit a token, and is more satisfying than checking for an actual concrete LHS in the current state, as done when applying the wand, as appliability is a more general notion. Intuitively, this is the semantics that `wand_satisfiable` should have, although the current definition of appliability is not formulated in Viper terms.

Towards checking appliability in an automated verifier. Appliability depends on the footprint σ of the wand, which we need to keep track of otherwise it is not even possible to say whether `acc(x.f) --* acc(x.f)` is appliable or not (depending on the presence of `acc(x.f)` in the footprint).

Even with a known footprint, it is difficult to check if two heaps are disjoint and if an assertion is satisfiable. For example, the footprints of predicates are not obviously accessible and require unfolding. Predicates might also hide some path conditions that would make the wand unappliable. Small examples of such situations are provided in Example 9, but they could be arbitrary complex (*e.g.* using recursive predicates).

Moreover, a procedure determining whether a wand is appliable must be very conservative when dealing with constructs that might hide information, such as functions and predicates. Indeed, missing a path condition for example leads to an over-approximation of the possible states, which usually results in incompleteness as the verifier fails to prove the property because of spurious counterexamples. However, when checking for wand appliability, this dynamic is inverted. If we over-approximate the set of valid LHS states by missing a hidden contradiction, we might incorrectly conclude that this set is non-empty, potentially causing an unsoundness as we expose the path conditions of an unappliable wand.

To soundly conclude that a wand is appliable, the verifier must guarantee that no contradiction lies hidden within some predicates or function calls by under-approximating the set of valid LHS states.

```
predicate Pf(x: Ref) { acc(x.f) }
method hidden_unappliable_fp(x: Ref)
  requires acc(x.f)
{
  package Pf(x) --* acc(x.f) {
    // Explicitly moving x.f to the footprint
    assert acc(x.f)
  }
}
```

(a) An example of unappliable wand because of its footprint cannot be disjoint from a heap satisfying the LHS.

```
predicate eq(x: Ref, y: Ref) { x == y }

method hidden_unappliable_pc(x: Ref, y: Ref)
  requires eq(x, y)
{
  package acc(x.f) && acc(y.f) --* true
}
```

(b) An example of unappliable wand because of some path conditions hidden within in a predicate.

Example 9: Wands might be unappliable for non obvious reasons, *e.g.*, reasons hidden within a predicate.

6.4 Potential use cases of attached expressions

The use of attached expressions has only been demonstrated on small examples such as the ones presented previously, so the question of their practical utility for real-world verification remains. Yet, as for attached facts, Prusti is a promising target since it naturally uses wands to model Rust borrows. Investigating this integration is left to future work.

Conclusion

Because of space constraints, the reader is invited to refer to the second page of this report for a summary of my contributions and a discussion of future work.

References

[1] J. C. Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures," in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, in LICS '02. USA: IEEE Computer Society, 2002, pp. 55–74.

[2] J. Smans, B. Jacobs, and F. Piessens, "Implicit dynamic frames: Combining dynamic frames and separation logic," in *European Conference on Object-Oriented Programming*, 2009, pp. 148–172.

[3] M. J. Parkinson and A. J. Summers, "The relationship between separation logic and implicit dynamic frames," *Logical Methods in Computer Science*, vol. 8, 2012.

[4] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A Verification Infrastructure for Permission-Based Reasoning," in *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, in VMCAI 2016. St. Petersburg, FL, USA: Springer-Verlag, 2016, pp. 41–62. doi: 10.1007/978-3-662-49122-5_2.

[5] 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)*, ACM, 2019, p. 147:1–147:30. doi: 10.1145/3360573.

[6] V. Astrauskas *et al.*, "The Prusti Project: Formal Verification for Rust," in *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings*, Pasadena, CA, USA: Springer-Verlag, 2022, pp. 88–108. doi: 10.1007/978-3-031-06773-0_5.

[7] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular Specification and Verification of Go Programs," in *Computer Aided Verification (CAV)*, A. Silva and K. R. M. Leino, Eds., in LNCS, vol. 12759. Springer International Publishing, 2021, pp. 367–379. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-81685-8_17

[8] M. Eilers and P. Müller, "Nagini: a static verifier for Python," in *International Conference on Computer Aided Verification*, 2018, pp. 596–603.

[9] S. Blom and M. Huisman, "The VerCors tool for verification of concurrent programs," in *International Symposium on Formal Methods*, 2014, pp. 127–131.

[10] K. R. M. Leino, "This is boogie 2," *manuscript KRML*, vol. 178, no. 131, p. 9, 2008.

[11] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, in TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340.

[12] H. Barbosa *et al.*, "cvc5: A Versatile and Industrial-Strength SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds., Cham: Springer International Publishing, 2022, pp. 415–442.

[13] M. Eilers, M. Schwerhoff, and P. Müller, "Verification Algorithms for Automated Separation Logic Verifiers." [Online]. Available: https://arxiv.org/abs/2405.10661

[14] M. H. Schwerhoff, "Advancing automated, permission-based program verification using symbolic execution," Doctoral dissertation, 2016.

[15] T. V. team, "Silicon: A Viper Verifier Based on Symbolic Execution." Accessed: Jul. 28, 2026. [Online]. Available: https://github.com/viperproject/silicon

[16] S. Blom and M. Huisman, "Witnessing the elimination of magic wands," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 6, pp. 757–781, 2015.

[17] B. Jacobs and F. Piessens, "The VeriFast program verifier," technical report, 2008.

[18] R. Piskac, T. Wies, and D. Zufferey, "GRASShopper," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 124–139.

[19] J. Fragoso Santos, P. Maksimović, S.-É. Ayoun, and P. Gardner, "Gillian, part i: a multi-language platform for symbolic execution," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, in PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 927–942. doi: 10.1145/3385412.3386014.

[20] P. Maksimović, S.-É. Ayoun, J. F. Santos, and P. Gardner, "Gillian, Part II: Real-World Verification for JavaScript and C," in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II*, Berlin, Heidelberg: Springer-Verlag, 2021, pp. 827–850. doi: 10.1007/978-3-030-81688-9_38.

[21] R. Brochenin, S. Demri, and E. Lozes, "On the almighty wand," *Information and Computation*, vol. 211, pp. 106–137, 2012, doi: https://doi.org/10.1016/j.ic.2011.12.003.

[22] J.-C. Filliâtre, L. Gondelman, and A. Paskevich, "A Pragmatic Type System for Deductive Verification," Feb. 2016. [Online]. Available: https://inria.hal.science/hal-01256434

[23] X. Denis, J.-H. Jourdan, and C. Marché, "Creusot: a Foundry for the Deductive Verification of Rust Programs," in *Lecture Notes in Computer Science*, in Lecture Notes in Computer Science. Madrid, Spain: Springer Verlag, Oct. 2022. [Online]. Available: https://inria.hal.science/hal-03737878

[24] A. Lattuada *et al.*, "Verus: Verifying Rust Programs using Linear Ghost Types," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, Apr. 2023, doi: 10.1145/3586037.

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

[26] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich, "Why3: Shepherd Your Herd of Provers," in *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, Aug. 2011, pp. 53–64.

[27] F. Kirchner, N. Kosmatov, V. Prévosto, J. Signoles, and B. Yakobowski, "Frama-C: A software analysis perspective," *Formal Aspects of Computing*, vol. 27, no. 3, pp. 573–609, May 2015, doi: 10.1007/s00165-014-0326-7.

[28] T. R. D. Team, "The Rocq Prover Reference Manual, Version 9.2." 2026. Accessed: Jul. 24, 2026. [Online]. Available: https://rocq-prover.org/doc/V9.2.0/refman/index.html

[29] T. Nipkow, "Programming and proving in Isabelle/HOL," in *Technical report, University of Cambridge*, 2013.

[30] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the foundations of the Rust programming language," in *45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2018)*, 2018, p. 66.

[31] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *Journal of functional programming*, vol. 28, p. e20, 2018.

[32] L. De Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer, "The Lean theorem prover (system description)," in *International Conference on Automated Deduction*, 2015, pp. 378–388.

[33] L. König, "An Improved Interface for Interactive Proofs in Separation Logic," Master's thesis, Karlsruher Institut für Technologie (KIT), 2022. doi: 10.5445/IR/1000153230.

[34] K. Morrison and L. de Moura, "grind: An SMT-Inspired Tactic for Lean 4 (Short Paper–System Description)," in *International Joint Conference on Automated Reasoning*, 2026, pp. 106–114.

[35] S. Spies *et al.*, "Destabilizing Iris," *Proc. ACM Program. Lang.*, vol. 9, no. PLDI, Jun. 2025, doi: 10.1145/3729284.

[36] G. Parthasarathy, P. Müller, and A. J. Summers, "Formally validating a practical verification condition generator," in *International Conference on Computer Aided Verification*, 2021, pp. 704–727.

[37] S. Keuchel, S. Huyghebaert, G. Lukyanov, and D. Devriese, "Verified symbolic execution with Kripke specification monads (and no meta-programming)," *Proc. ACM Program. Lang.*, vol. 6, no. ICFP, Aug. 2022, doi: 10.1145/3547628.

[38] G. Parthasarathy, T. Dardinier, B. Bonneau, P. Müller, and A. J. Summers, "Towards trustworthy automated program verifiers: formally validating translations into an intermediate verification language," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 1510–1534, 2024.

[39] H. Ling, T. Dardinier, E. Arlt, and P. Müller, "Sound State Encodings in Translational Separation Logic Verifiers (Extended Version)." [Online]. Available: https://arxiv.org/abs/2603.20001

[40] T. Dardinier, G. Parthasarathy, N. Weeks, P. Müller, and A. J. Summers, "Sound Automation of Magic Wands," in *Computer Aided Verification*, S. Shoham and Y. Vizel, Eds., Cham: Springer International Publishing, 2022, pp. 130–151.

[41] J.-M. Madiot, "MPRI Course: Proofs of Programs." [Online]. Available: https://madiot.fr/progproofs/

[42] J. Boyland, "Checking interference with fractional permissions," in *International Static Analysis Symposium*, 2003, pp. 55–72.

[43] T. V. Team, "Viper Tutorial." Accessed: Apr. 08, 2026. [Online]. Available: https://viper.ethz.ch/tutorial/

[44] M. Schwerhoff and A. J. Summers, *Lightweight support for magic wands in an automatic verifier*, vol. 37. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015.

[45] M. Dublanc, "Separating Magic Wands and Functional Properties," Bachelor's thesis, 2024.

[46] N. Becker, "Generalized Verification Support for Magic Wands," Bachelor's thesis, 2017.

[47] D. Peter, "hyperfine." [Online]. Available: https://github.com/sharkdp/hyperfine

[48] J. Pereira *et al.*, "Protocols to Code: Formal Verification of a Secure Next-Generation Internet Router," in *Computer and Communications Security (CCS)*, in CCS '25. Taipei, Taiwan: Association for Computing Machinery, 2025. doi: 10.1145/3719027.3765104.

[49] L. Arquint, M. Schwerhoff, V. Mehta, and P. Müller, "A Generic Methodology for the Modular Verification of Security Protocol Implementations," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, in CCS '23. Copenhagen, Denmark: Association for Computing Machinery, 2023, pp. 1377–1391. doi: 10.1145/3576915.3623105.

[50] M. Huisman, V. Klebanov, and R. Monahan, "VerifyThis 2012," *Int. J. Softw. Tools Technol. Transf.*, vol. 17, no. 6, pp. 647–657, Nov. 2015, doi: 10.1007/s10009-015-0396-8.

A No unique package strategy

This example, from [40], shows that there is no unique footprint selection strategy. Consider the following wand: `acc(x.b, 1/2) --* acc(x.b, 1/2) && (x.b ⟹ acc(x.f))`, and suppose that: the current state holds permissions to both `x.f` and `x.b`, and that `x.b` is false. Valid footprints for this wand include:

- full permission to `x.f`: this guarantees that no matter the value of the concrete LHS used to apply the wand, permissions to `x.f` can be provided if needed;
- any non-zero permission to `x.b`: packaging a non-zero permission from the current state into the footprint will also register that `x.b` is false (as fractional permissions must agree on the value), thus circumventing the need of providing `acc(x.f)`.

B Details of soundness issues

B.1 Incompleteness of the existing wand encoding

The following program fails to verify despite being correct. Note that the magic wand is a fancy version of the identity wand, as the RHS provides exactly the permissions provided by the LHS (potentially in a different order), and thus packaging then applying the wand is expected to be a no-op.

```
method incompleteness(x: Ref)
    requires acc(x.b) && acc(x.f) && acc(x.g)
{
    x.g := 5

    define A acc(x.b) && acc(x.f) && acc(x.g)
    define B acc(x.b) && ((old[lhs](x.b)) ? acc(x.f) && acc(x.g) : acc(x.g) && acc(x.f))
    package A --* B

    apply A --* B

    assert x.g  == 5 // Fails!
}
```

B.2 Tests failing with the new encoding

Table 3 briefly presents the five tests failing with the new encoding of path conditions when packaging wands. Three of these tests are described in more details below, whereas the two others, `conditionals3.vpr` and `PackageStateConsolidator.vpr`, are really convoluted and can be verified successfully using the SE-PC flag to ensure a more systematic way to consume permissions from the context.

B.2.1 `conditionals.vpr` and `SnapshotsBranching.vpr`

These tests used to verify successfully but I argue that they should never had passed in the first place. In essence, both tests are the following:

```
method branching(x: Ref)
  requires acc(x.f) && acc(x.g, write) && acc(x.h, write)
```

Failing test file	Comment
`conditionals.vpr` `SnapshotsBranching.vpr`	should have been failing in the first place as they exhibit a wrong footprint computation for wands (essentially described in [40])
`conditionals3.vpr` `PackageStateConsolidator.vpr`	can be successfully verified by enabling the SE-PC flag
`packaging_1.vpr`	true completeness regression, the problem has been identified (unification of terms with fractional permissions), but the engineering efforts to make this behavior verifiable were deemed not worth it

Table 3: A brief description of the 5 tests now failing.

```
{
  define A acc(x.f)
  define B acc(x.f) && (x.f ? acc(x.g) : acc(x.h))

  package A --* B
  apply A --* B

  assert acc(x.f) && acc(x.g) && acc(x.h)
}
```

Because it is not possible to know in advance the value of the provided LHS `x.f`, both permissions to `x.g` and `x.h` must be packaged inside the footprint. However, in any case, only one of these permissions is provided back by the RHS, the other is leaked and lost forever. Then, the final assertion is not expected to hold. The new encoding successfully prevents these tests from passing.

B.2.2 `packaging_1.vpr`

This exemple is the one true regression where the new encoding is incomplete. It involves two nested packages, to exhibit an incompleteness regarding fractional permissions (which all must agree on the value of the memory location):

```
method incomplete(x)
  requires acc(x.f, 1/2)
{
  package acc(x.f, 1/4) && x.f == 2 --* acc(x.f, 1/4) && x.f == 2 {
      package true --* acc(x.f, 1/2) && x.f == 2
  }
}
```

Schematically:

- the external context provides a chunk with $1/2$ permission to `x.f` with an unknown value f_1
- the LHS of the outer wand provides a chunk with $1/4$ permission to `x.f` with a value 2
  - because fractional permissions must agree on the value of the pointed memory location, we should ideally learn that $f_1 = 2$, however, doing so is unsound in general (see below) and thus we do not learn this fact,
- when packaging the inner wand, we consume the chunk provided by the LHS of the outer wand, and $1/4$ permission from the chunk of the external context
- when trying to consume the RHS of the outer wand, the only $1/4$ permission left is from the chunk of the external context, with value f_1 which cannot be proven equal to 2 as we didn't derive the equality originating from fractional permissions - the package fails.

Registering equalities from fractional permissions is a side effect, and thus, can lead to a similar unsoundness issue as the one based on the state consolidator.

B.3 Benchmarking the new encoding

All source files and the benchmark script can be found on my repository[17]. The new encoding of wands was benchmarked using hyperfine [47] on two large, real-world verified software projects:

- an Internet router [48], in which two methods leverage magic wands;
- a library for verifying security protocol implementations [49], in which three methods leverage magic wands.

Hyperfine generated the results shown in Table 4 and Table 5, displaying equivalent performance for both encodings.

[17] https://github.com/remigerme/silicon/blob/9ca9b342a9ad12dc888b573d7ca046ba9f749783/remi-demo

Encoding	Mean [s]	Min [s]	Max [s]	Relative
old	139.253 ± 21.251	123.357	173.944	1.05 ± 0.17
new	132.219 ± 7.593	124.566	143.658	1.00

Table 4: Hyperfine benchmark results for the Internet router (5 runs per encoding).

Encoding	Mean [s]	Min [s]	Max [s]	Relative
old	25.433 ± 0.471	24.367	26.227	1.00
new	26.129 ± 0.647	25.138	27.397	1.03 ± 0.03

Table 5: Hyperfine benchmark results for the security protocol verification library (15 runs per encoding).

C Programs Showcasing Attached Facts

On top of the classic programs from [4] (available at https://github.com/remigerme/silver/blob/rgerme-mw-attached-facts/src/test/resources/examples/vmcai2016/linked-list-predicates-with-attached-wands.vpr) and [50] (available at https://github.com/remigerme/silver/blob/rgerme-mw-attached-facts/src/test/resources/wands/examples_paper/tree_delete_min.vpr), I propose new demo files to showcase attached facts. One of them can be found below: it simply deletes the last element of non-empty list, preserving all the other elements. This program and others can be found at https://github.com/remigerme/silver/tree/rgerme-mw-attached-facts/src/test/resources/wands/attached.

```
field val: Int
field next: Ref

predicate list(x: Ref) { x ≠ null ⟹ acc(x.val) && acc(x.next) && list(x.next) }
function len(x: Ref): Int requires list(x) {
    x == null ? 0 : 1 + unfolding list(x) in len(x.next)
}
function elems(x: Ref): Seq[Int] requires list(x) {
    x == null ? Seq[Int]() : unfolding list(x) in Seq(x.val) ++ elems(x.next)
}

method delete_last(x: Ref) returns (y: Ref)
    requires list(x) && len(x) > 0
    ensures  list(y)
    ensures  len(y) == old(len(x)) - 1
    ensures  elems(y) == old(elems(x))[0..|old(elems(x))| - 1]
{
    unfold list(x)
    if (x.next == null) {
        y := null; fold list(y)
    } else {
        fold list(x)
        y := x

        define W(c) list(c) --* list(y)
        define F(c, l) old[lhs](len(c)) == l - 1 ⟹ len(y) == old(len(x)) - 1
        define G(c, e) old[lhs](elems(c)) == e[0..|e| - 1] ⟹ elems(y) == old(elems(x))[0..|
old(elems(x))| - 1]

        // The traditional approach is to define the following wand,
        // encapsulating all the specification:
        // define W(c, l, e) list(c) && len(c) == l - 1 && elems(c) == e[0..|e| - 1] --* list(y)
&& len(y) == old(len(x)) - 1 && elems(y) == old(elems(x))[0..|old(elems(x))| - 1]

        var cur: Ref := y
        var len_cur: Int := len(cur)
        var elm_cur: Seq[Int] := elems(cur)
```

```
        package W(cur) // We don't have to explicitly attach the facts!

        while (unfolding list(cur) in unfolding list(cur.next) in cur.next.next ≠ null)
            invariant list(cur) && cur ≠ null && unfolding list(cur) in cur.next ≠ null
            invariant len_cur == len(cur)
            invariant elm_cur == elems(cur)
            invariant W(cur)
            invariant attached F(cur, len_cur) to W(cur)
            invariant attached G(cur, elm_cur) to W(cur)
        {
            var prev_cur: Ref := cur
            var prev_len_cur: Int := len_cur
            var prev_elm_cur: Seq[Int] := elm_cur

            unfold list(cur)
            cur := cur.next
            len_cur := len(cur)
            elm_cur := elems(cur)

            package W(cur) // Once again, not explicitly attaching the facts!
            {
                fold list(prev_cur)
                apply W(prev_cur)
            }
        }

        // Lemma for the length (required with or without attached facts)
        assert unfolding list(cur) in unfolding list(cur.next) in len(cur.next.next) == 0

        // Helping for the elements (idem)
        assert unfolding list(cur) in unfolding list(cur.next) in |elems(cur.next.next)| == 0

        unfold list(cur)
        cur.next := null
        fold list(cur.next)
        fold list(cur)

        apply W(cur)
    }
}
```

D Exposing the wand definition unguarded

Recall that for this example, accessing the wand definition suffices to successfully use attached expressions.

Question. Is it sound to expose the wand definition unguarded? That is, is it sound to register an extra path condition when packaging a wand: $\forall \mathtt{t_A} : \bigvee_i \mathtt{mwsf}(\mathtt{t_A}) = \mathtt{def}_i$ where i iterates over branches?

Intuitively, my first answer to that question is that there is no risk exposing the definition, as I believe it can only allow to access values that were packed in the footprint. However, we have to be careful regarding what we expect from wands: is it always sound (or desirable) to read these values? When considering vacuous wands, the value packed in the footprint might differ from the value promised by the RHS, as shown in Example 10. In that case, it is unclear to me what is expected, and this question is left as future work.

```
method is_this_legal(x: Ref)
  requires acc(x.f)
{
  x.f := 0
  define W acc(x.f, 1/2) && x.f == 42 --* acc(x.f) && x.f == 42
  package W
  assert (attachedexp x.f to W) == 0
}
```

Example 10: Exposing the wand definition without any guard allows to access values packed inside the footprint of a vacuous wand (which will never be applied).

E Proving invariants for the length of the prefix list

Proving that the attached expression at the end of the loop is valid and is equal to `i` requires a bit more work than the `pair` example. Table 6 summarizes the path conditions induced by inhaling the invariants as well as packaging the wand. Function precondition tokens are omitted for simplicity. $\mathtt{w}_{\mathtt{old}}$ and its associated MWSF $\mathtt{mwsf}_{\mathtt{old}}$ denote the wand produced by the first invariant. $\mathtt{i}_{\mathtt{old}}$ (resp. $\mathtt{t}_{\mathtt{cur}}$) denotes the value of `i` (resp. the first element of `cur`) at the beginning of the loop iteration. $\mathtt{t}_{\mathtt{cur}}$ is packed inside the footprint when packaging the wand at the end of the loop.

Path condition	Origin
$\forall \mathtt{t}_1, \mathtt{t}_2 : \mathtt{MW\_token}(\mathtt{w}_{\mathtt{old}}, \mathtt{t}_1) \wedge \mathtt{MW\_token}(\mathtt{w}_{\mathtt{old}}, \mathtt{t}_2) \Rightarrow$ $\mathtt{len}(\mathtt{mwsf}_{\mathtt{old}}(\mathtt{t}_1)) - \mathtt{len}(\mathtt{t}_1) = \mathtt{len}(\mathtt{mwsf}_{\mathtt{old}}(\mathtt{t}_2)) - \mathtt{len}(\mathtt{t}_2)$	invariant `attachedexp_valid` ... refered to as $\star$
$\mathtt{len}(\mathtt{mwsf}(\mathtt{t}_A)) - \mathtt{len}(\mathtt{t}_A) = \mathtt{i}_{\mathtt{old}}$	invariant `attachedexp` ...
$\forall \mathtt{t}_A : \mathtt{MW\_token}(\mathtt{w}, \mathtt{t}_A) \Rightarrow [\mathtt{mwsf}(\mathtt{t}_A) = \mathtt{mwsf}_{\mathtt{old}}(\mathtt{t}_{\mathtt{cur}} :: \mathtt{t}_A)$ $\wedge \mathtt{MW\_token}(\mathtt{w}_{\mathtt{old}}, \mathtt{t}_{\mathtt{cur}} :: \mathtt{t}_A)]$	packaging the new wand `w` ($\mathtt{w}_{\mathtt{old}}$ is applied during the proof script, hence its token being registered as a PC)

Table 6: Summary of the relevant path conditions registered in the loop iteration of Example 8.

With all this, I can review the proof of preservation of the independence property:

$$
\begin{aligned}
\mathtt{len}(\mathtt{mwsf}(\mathtt{t}_1)) - \mathtt{len}(\mathtt{t}_1) &= \mathtt{len}(\mathtt{mwsf}_{\mathtt{old}}(\mathtt{t}_{\mathtt{cur}} :: \mathtt{t}_1)) - \mathtt{len}(\mathtt{t}_1) && \text{by def } \mathtt{mwsf} \text{ (requires } \mathtt{MW\_token}(\mathtt{w}, \mathtt{t}_1)) \\
&= \mathtt{len}(\mathtt{mwsf}_{\mathtt{old}}(\mathtt{t}_{\mathtt{cur}} :: \mathtt{t}_1)) - \mathtt{len}(\mathtt{t}_{\mathtt{cur}} :: \mathtt{t}_1) + 1 && \text{by def } \mathtt{len} \text{ (requires } \mathtt{MW\_token}(\mathtt{w}_{\mathtt{old}}, \mathtt{t}_{\mathtt{cur}} :: \mathtt{t}_1)) \\
&= \mathtt{len}(\mathtt{mwsf}_{\mathtt{old}}(\mathtt{t}_{\mathtt{cur}} :: \mathtt{t}_2)) - \mathtt{len}(\mathtt{t}_{\mathtt{cur}} :: \mathtt{t}_2) + 1 && \text{by } (\star) \text{ (requires } \mathtt{MW\_token}(\mathtt{w}_{\mathtt{old}}, \mathtt{t}_{\mathtt{cur}} :: \mathtt{t}_1) \text{ and } \mathtt{MW\_token}(\mathtt{w}_{\mathtt{old}}, \mathtt{t}_{\mathtt{cur}} :: \mathtt{t}_2)) \\
&= \mathtt{len}(\mathtt{mwsf}_{\mathtt{old}}(\mathtt{t}_{\mathtt{cur}} :: \mathtt{t}_2)) - \mathtt{len}(\mathtt{t}_2) && \text{by def } \mathtt{len} \text{ (requires } \mathtt{MW\_token}(\mathtt{w}_{\mathtt{old}}, \mathtt{t}_{\mathtt{cur}} :: \mathtt{t}_2)) \\
&= \mathtt{len}(\mathtt{mwsf}(\mathtt{t}_2)) - \mathtt{len}(\mathtt{t}_2) && \text{by def } \mathtt{mwsf} \text{ (requires } \mathtt{MW\_token}(\mathtt{w}, \mathtt{t}_2))
\end{aligned}
$$

All the required tokens are available in the context and the proof succeeds.

Now, the proof of preservation of the last invariant, relating the attached expression and `i`, is:

$$
\begin{aligned}
\mathtt{len}(\mathtt{mwsf}(\mathtt{t}_B)) - \mathtt{len}(\mathtt{t}_B) &= \mathtt{len}(\mathtt{mwsf}_{\mathtt{old}}(\mathtt{t}_{\mathtt{cur}} :: \mathtt{t}_B)) - \mathtt{len}(\mathtt{t}_B) && \text{by def } \mathtt{mwsf} \text{ (requires } \mathtt{MW\_token}(\mathtt{w}, \mathtt{t}_B)) \\
&= \mathtt{len}(\mathtt{mwsf}_{\mathtt{old}}(\mathtt{t}_{\mathtt{cur}} :: \mathtt{t}_B)) - \mathtt{len}(\mathtt{t}_{\mathtt{cur}} :: \mathtt{t}_B) + 1 && \text{by def } \mathtt{len} \text{ (requires } \mathtt{MW\_token}(\mathtt{w}_{\mathtt{old}}, \mathtt{t}_{\mathtt{cur}} :: \mathtt{t}_B)) \\
&= \mathtt{len}(\mathtt{mwsf}_{\mathtt{old}}(\mathtt{t}_A)) - \mathtt{len}(\mathtt{t}_A) + 1 && \text{by } (\star) \text{ (requires } \mathtt{MW\_token}(\mathtt{w}_{\mathtt{old}}, \mathtt{t}_{\mathtt{cur}} :: \mathtt{t}_B) \text{ and } \mathtt{MW\_token}(\mathtt{w}_{\mathtt{old}}, \mathtt{t}_A)) \\
&= \mathtt{i}_{\mathtt{old}} + 1 && \text{by path condition on } \mathtt{t}_A \text{ and } \mathtt{i}_{\mathtt{old}}
\end{aligned}
$$

Tokens in maroon can be retrieved from $\mathtt{MW\_token}(\mathtt{w}, \mathtt{t}_B)$, $\mathtt{t}_B$ being the artificial LHS introduced when evaluating the `attachedexp` at the end of the loop. On the other hand, $\mathtt{MW\_token}(\mathtt{w}_{\mathtt{old}}, \mathtt{t}_A)$ refers to the initial `attachedexp`. Neither of these tokens is available. Note that we do not know anything about $\mathtt{t}_A$ and $\mathtt{t}_B$, they just witness the existence of such tokens, *i.e.*, $\exists\, \mathtt{t} : \mathtt{MW\_token}(w, \mathtt{t})$ for $w = \mathtt{w}$ or $w = \mathtt{w}_{\mathtt{old}}$.