Idiomatic specification of magic wands in implicit dynamic frames

Rémi Germe

Supervised by Jonáš Fiala, Marco Eilers, Peter Müller

Programming Methodology Group, ETH Zürich

Implicit dynamic frames in a nutshell

Verifying heap-manipulating programs...

```
method mergesort(root: Ref)
  requires list(root)
  ensures  list(root) && sorted(root) &&  ...
```

Implicit dynamic frames in a nutshell

Verifying heap-manipulating programs...

```
method mergesort(root: Ref)
  requires list(root)
  ensures  list(root) && sorted(root) && ...
```

... with implicit dynamic frames (IDF)

Permission to heap location required to read/write

> **IDF:** permission reasoning
>
> `acc(x.val)`

Implicit dynamic frames in a nutshell

Verifying heap-manipulating programs...

```
method mergesort(root: Ref)
  requires list(root)
  ensures  list(root) && sorted(root) &&  ...
```

... with implicit dynamic frames (IDF)

Permission to heap location required to read/write

IDF: permission reasoning ⊕ functional specification

`acc(x.val)` `x.val + 1` **heap-dependent-expression**

Implicit dynamic frames in a nutshell

Verifying heap-manipulating programs...

```
method mergesort(root: Ref)
  requires list(root)
  ensures  list(root) && sorted(root) &&  ...
```

... with implicit dynamic frames (IDF)

Permission to heap location required to read/write

IDF: permission reasoning $\oplus$ functional specification

`acc(x.val)` `x.val + 1` **heap-dependent-expression**

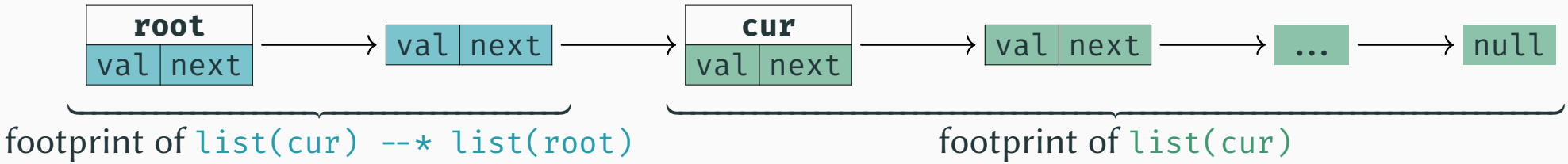
```
predicate list(x: Ref) { x ≠ null ⟹ acc(x.val) && acc(x.next) && list(x.next) }

function sorted(x: Ref): Bool
  requires list(x) {
  x ≠ null && x.next ≠ null ⟹ x.val ⩽ x.next.val && sorted(x.next)
}
```

Magic wands in a nutshell

Magic wand A `--*` B can be *applied* to exchange A for B

Typical use case

```
while(cur ≠ null)
  invariant list(cur)
  invariant list(cur) --* list(root)
```

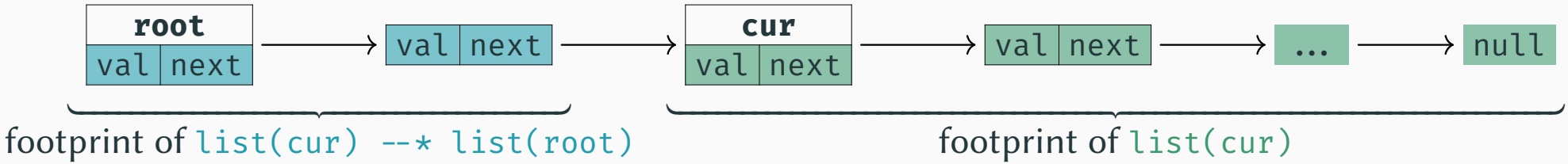


Magic wands in a nutshell

Magic wand A `--*` B can be *applied* to exchange A for B

Typical use case

```
while(cur ≠ null)
  invariant list(cur)
  invariant list(cur) --* list(root)
```



Heap-dependent expressions: predicates vs wands

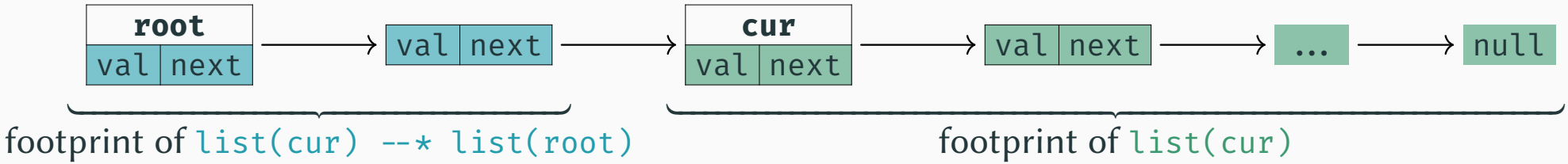
- **predicates:** framing `cur.val`: `unfolding list(cur) in cur.val`

Magic wands in a nutshell

Magic wand A `--*` B can be *applied* to exchange A for B

Typical use case

```
while(cur ≠ null)
  invariant list(cur)
  invariant list(cur) --* list(root)
```



Heap-dependent expressions: predicates vs wands

- **predicates:** framing `cur.val`: `unfolding list(cur) in cur.val`
- **magic wands:** framing `root.val` `??? list(cur) --* list(root) in root.val`

Goal of the internship

Goal: prove well-framedness of heap-dependent expressions whose required permissions are hidden inside the footprint of a wand.

Goal of the internship

Goal: prove well-framedness of heap-dependent expressions whose required permissions are hidden inside the footprint of a wand.

- **Incremental** and **modular** specification for wands
- Gain of **expressiveness** (*e.g.*, promising usage in Rust)

Goal of the internship

Goal: prove well-framedness of heap-dependent expressions whose required permissions are hidden inside the footprint of a wand.

- **Incremental** and **modular** specification for wands
- Gain of **expressiveness** (*e.g.*, promising usage in Rust)

Tasks

1. Design a new IDF construct
2. Implementation in the **automated** verifier Viper (symbolic execution backend)

Contributions

A sound algorithm and encoding for packaging path conditions

- discovered and fixed **soundness** issues

Contributions

A sound algorithm and encoding for packaging path conditions

- discovered and fixed **soundness** issues

Attached facts

- enable **incremental** and **modular** functional specification of magic wands
- highly **automated** (come for free when packaging)
- promising application to Rust in Prusti

Contributions

A sound algorithm and encoding for packaging path conditions

- discovered and fixed **soundness** issues

Attached facts

- enable **incremental** and **modular** functional specification of magic wands
- highly **automated** (come for free when packaging)
- promising application to Rust in Prusti

Attached expressions

- design space exploration
- theoretical **sufficient validity conditions**

Main challenge: the footprint of a wand is not explicit (1)

Predicates (explicit footprint)

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

Wands (implicit footprint)

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

Main challenge: the footprint of a wand is not explicit (1)

Predicates (explicit footprint)

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

Wands (implicit footprint)

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

Vacuous wands

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

Main challenge: the footprint of a wand is not explicit (1)

Predicates (explicit footprint)

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

Wands (implicit footprint)

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

Vacuous wands

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

- `false --* anything`

Main challenge: the footprint of a wand is not explicit (1)

Predicates (explicit footprint)

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

Wands (implicit footprint)

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

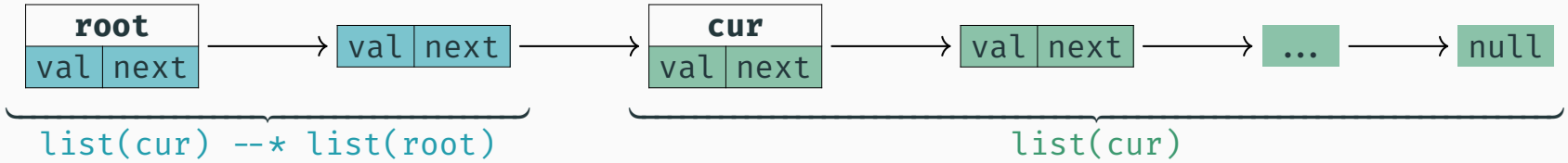
Vacuous wands

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

- `false --* anything`
- `acc(x.f) --* acc(x.f)`

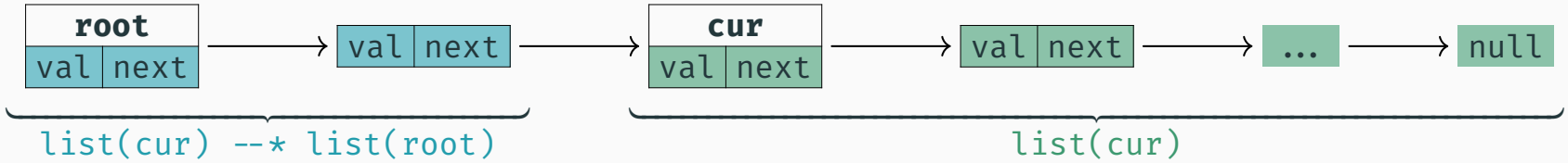
Main challenge: the footprint of a wand is not explicit (2)

Remainder, intuitive scenario:

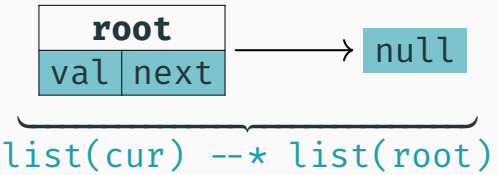


Main challenge: the footprint of a wand is not explicit (2)

Remainder, intuitive scenario:



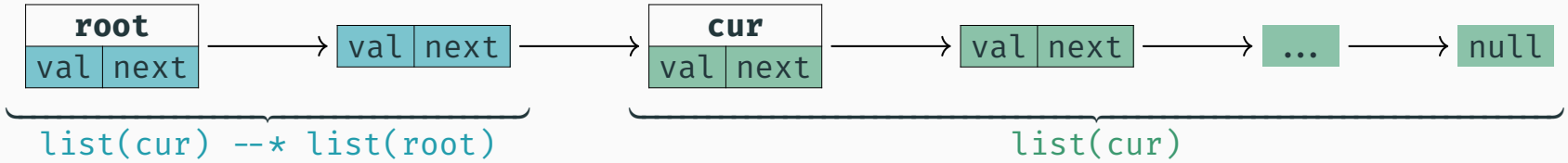
Unintuitive wands



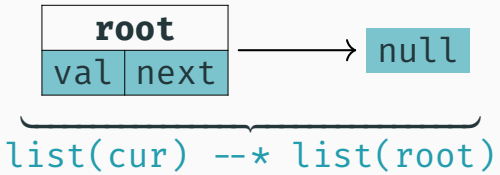
Do not depend on the LHS

Main challenge: the footprint of a wand is not explicit (2)

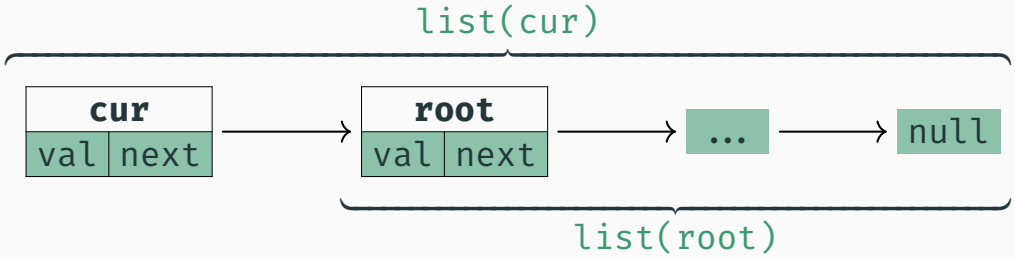
Remainder, intuitive scenario:



Unintuitive wands



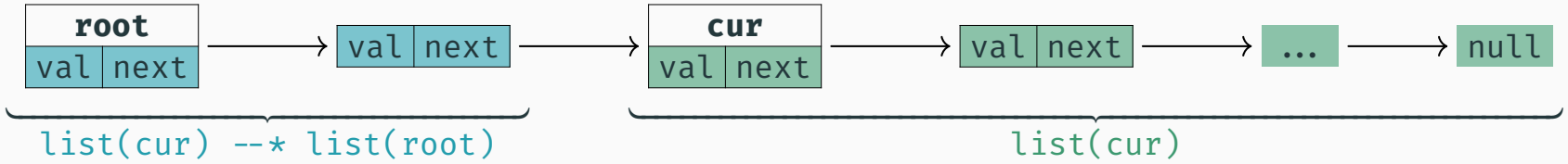
Do not depend on the LHS



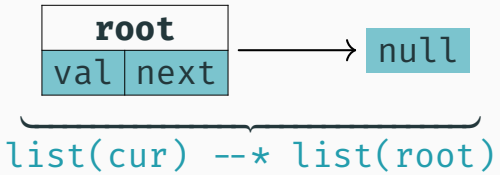
Empty footprint

Main challenge: the footprint of a wand is not explicit (2)

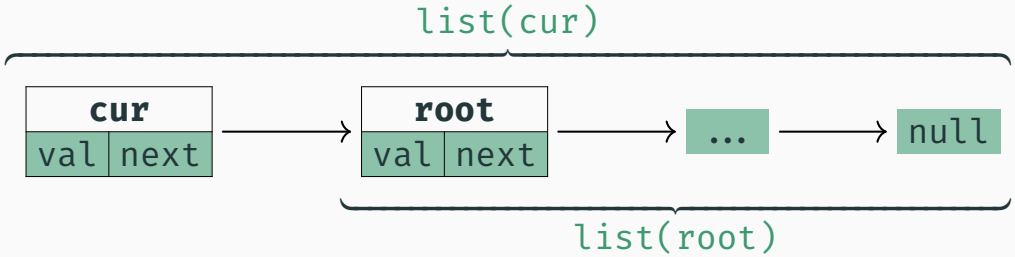
Remainder, intuitive scenario:



Unintuitive wands



Do not depend on the LHS

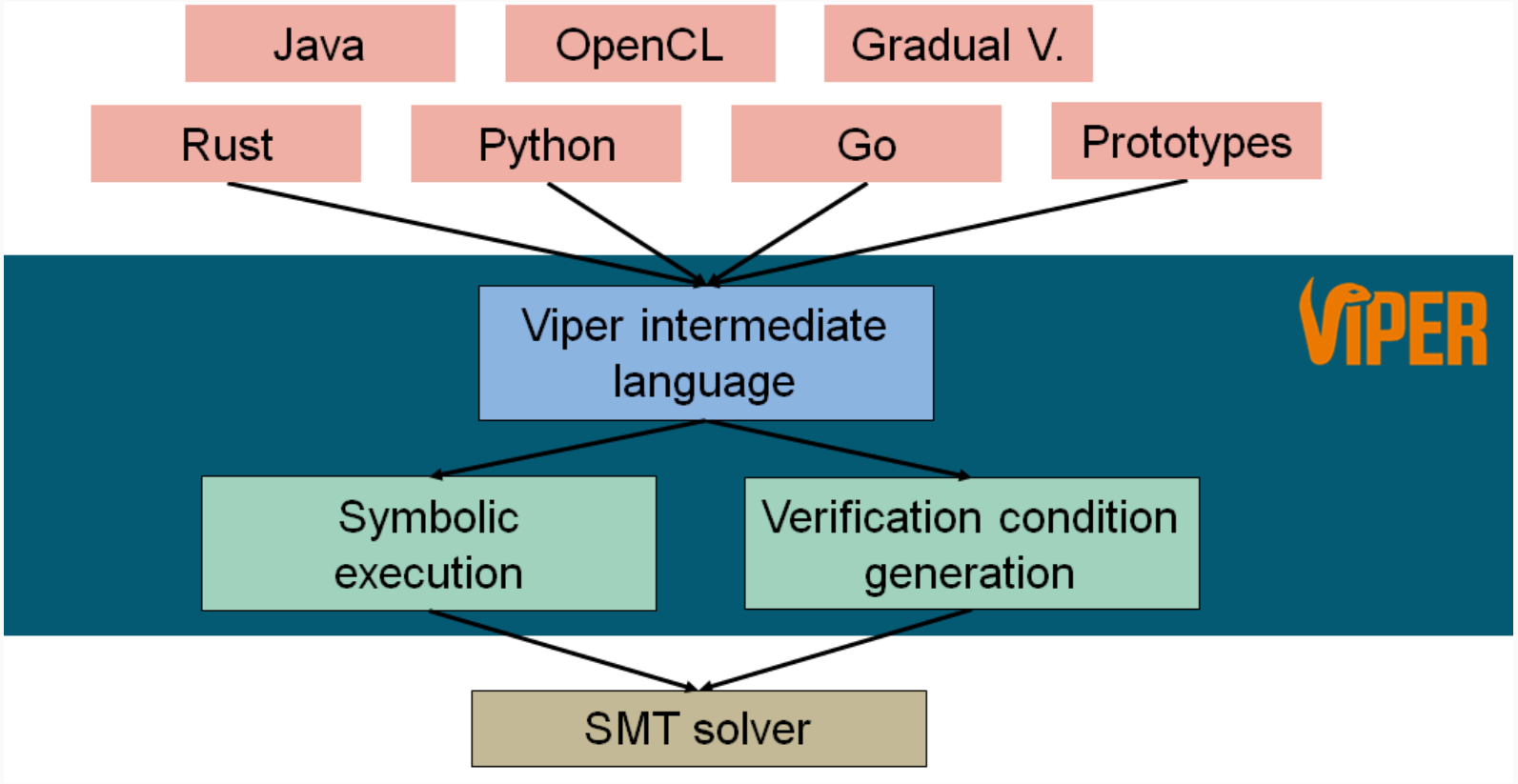


Empty footprint

Conclusion: the wand's syntax is not enough to infer its footprint.

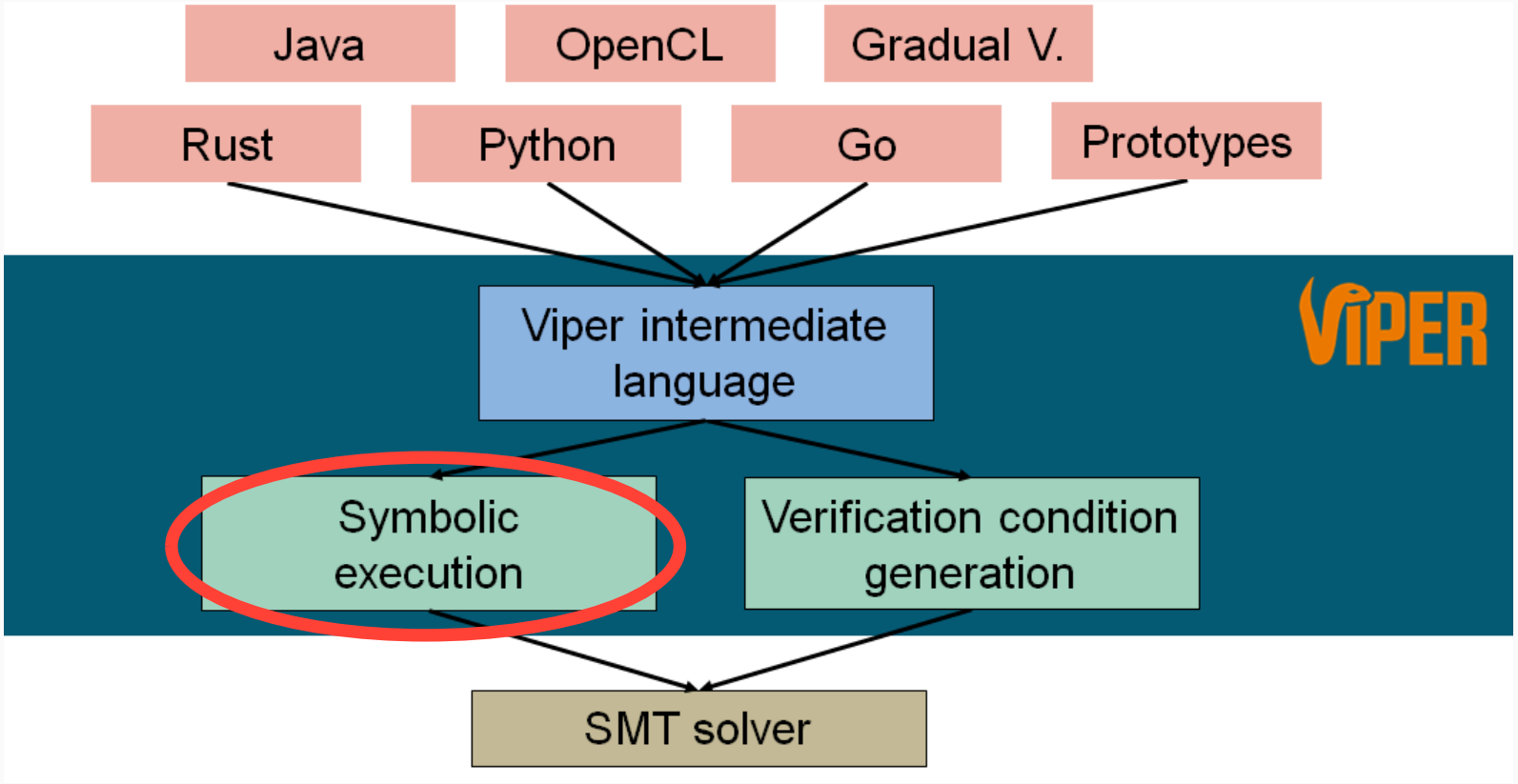
A sound package algorithm for path conditions

An overview of Viper



From viper.ethz.ch (accessed on 24.07.2026)

An overview of Viper



From viper.ethz.ch (accessed on 24.07.2026)

Symbolic state

Symbolic variables

Symbolic state

Symbolic variables + symbolic heaps

Symbolic state

Symbolic variables + symbolic heaps + path conditions

Silicon: the symbolic execution backend of Viper

Symbolic state

Symbolic variables + symbolic heaps + path conditions

Internal procedures

- `eval(e: Expression, s: State)(Q: Term => ...)`

  e must be well-framed

Symbolic state

Symbolic variables + symbolic heaps + path conditions

Internal procedures

- `eval(e: Expression, s: State)(Q: Term => ...)`
  e must be well-framed
- `execute(stmt: Statement, s: State)(Q: State => ...)`
  can branch and call `Q` several times

Silicon: the symbolic execution backend of Viper

Symbolic state

Symbolic variables + symbolic heaps + path conditions

Internal procedures

- `eval(e: Expression, s: State)(Q: Term => ...)`
  e must be well-framed
- `execute(stmt: Statement, s: State)(Q: State => ...)`
  can branch and call `Q` several times
- `consume(a: Assertion, s: State)(Q: (Term, State) => ...)`
  asserts that `a` holds in `s`, and removes the resources in `a`,

Silicon: the symbolic execution backend of Viper

Symbolic state

Symbolic variables + symbolic heaps + path conditions

Internal procedures

- `eval(e: Expression, s: State)(Q: Term => ...)`
  e must be well-framed
- `execute(stmt: Statement, s: State)(Q: State => ...)`
  can branch and call `Q` several times
- `consume(a: Assertion, s: State)(Q: (Term, State) => ...)`
  asserts that `a` holds in `s`, and removes the resources in `a`, *e.g.*,
  - `consume(x.f ⩾ 0)`: regular assertion
  - `consume(acc(x.f))`: asserts the permission is held and removes it

Silicon: the symbolic execution backend of Viper

Symbolic state

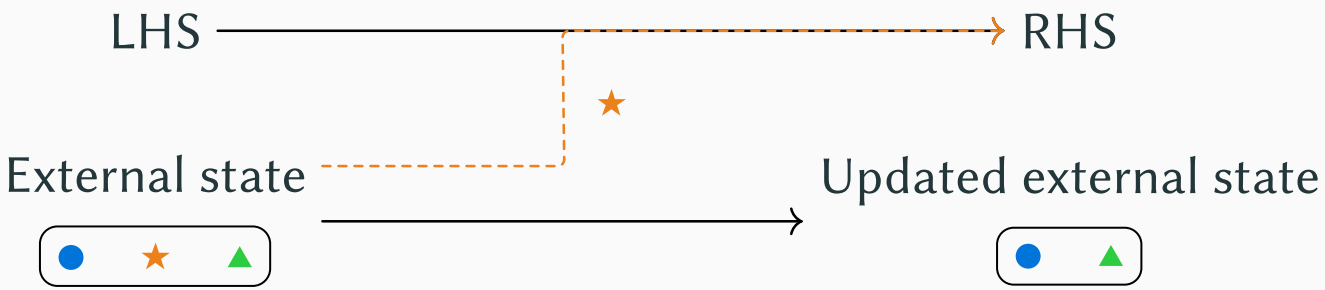
Symbolic variables + symbolic heaps + path conditions

Internal procedures

- `eval(e: Expression, s: State)(Q: Term => ...)`
  e must be well-framed
- `execute(stmt: Statement, s: State)(Q: State => ...)`
  can branch and call `Q` several times
- `consume(a: Assertion, s: State)(Q: (Term, State) => ...)`
  asserts that a holds in s, and removes the resources in a, *e.g.*,
  - `consume(x.f ⩾ 0)`: regular assertion
  - `consume(acc(x.f))`: asserts the permission is held and removes it
- `produce(a: Assertion, t: Term, s: State)(Q: State => ...)`
  adds the resources in a and assumes a holds

The existing package algorithm

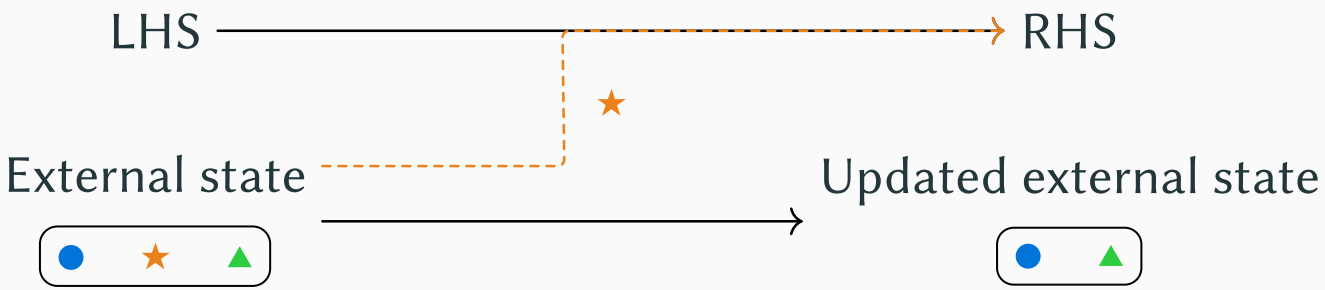
Packaging a wand



Footprint = ★

The existing package algorithm

Packaging a wand



Footprint = ★

Overview of the package algorithm

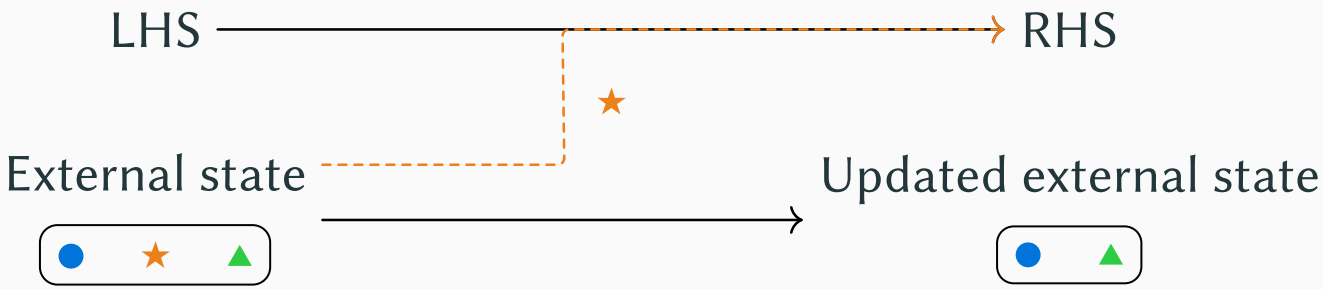
1. produce LHS t_A
2. execute proof script
3. consume RHS

} Any of these steps might **branch** and put resources in the footprint

For each recorded branch:

The existing package algorithm

Packaging a wand



Footprint = ★

Overview of the package algorithm

1. produce LHS t_A
2. execute proof script
3. consume RHS

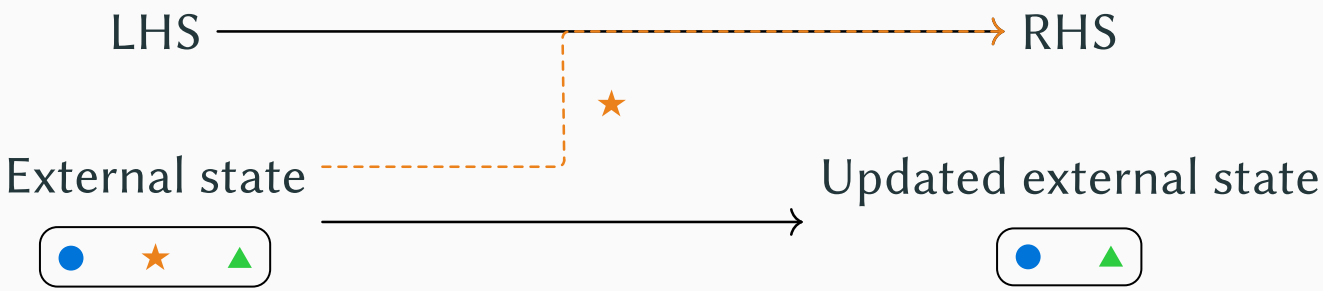
} Any of these steps might **branch** and put resources in the footprint

For each recorded branch:

4. record the wand definition: $\forall \mathbf{t_A} : \mathbf{mwsf}(\mathbf{t_A}) = \mathbf{def}_i$
5. handle path conditions (intricate)
6. call the continuation

The existing package algorithm

Packaging a wand



Footprint = ★

Overview of the package algorithm

1. produce LHS t_A
2. execute proof script
3. consume RHS

} Any of these steps might **branch** and put resources in the footprint

For each recorded branch:

4. record the wand definition: $\forall t_A : \mathtt{mwsf}(t_A) = \mathtt{def}_i$
5. handle path conditions (intricate)
6. call the continuation

Intuition: unsound to assume the branch at package time

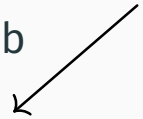
A program showcasing the unsoundness of the algorithm

```
acc(x.b) --* acc(x.b) && (x.b ? acc(x.f) : acc(x.g))
```

A program showcasing the unsoundness of the algorithm

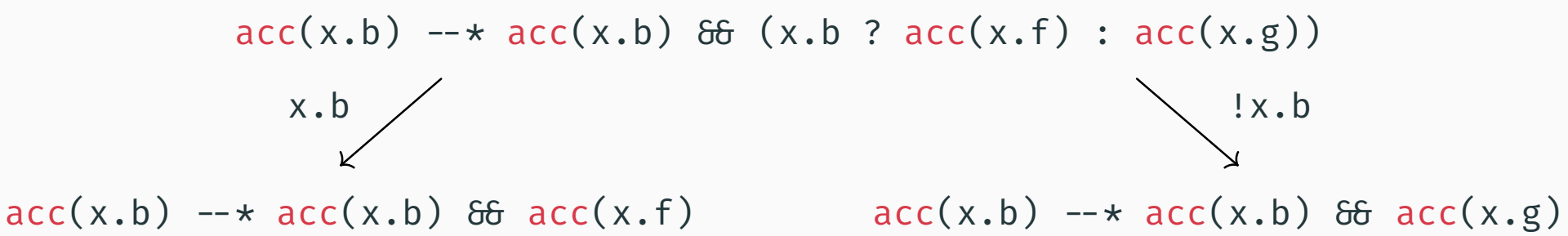
```
acc(x.b) --* acc(x.b) && (x.b ? acc(x.f) : acc(x.g))
```

x.b

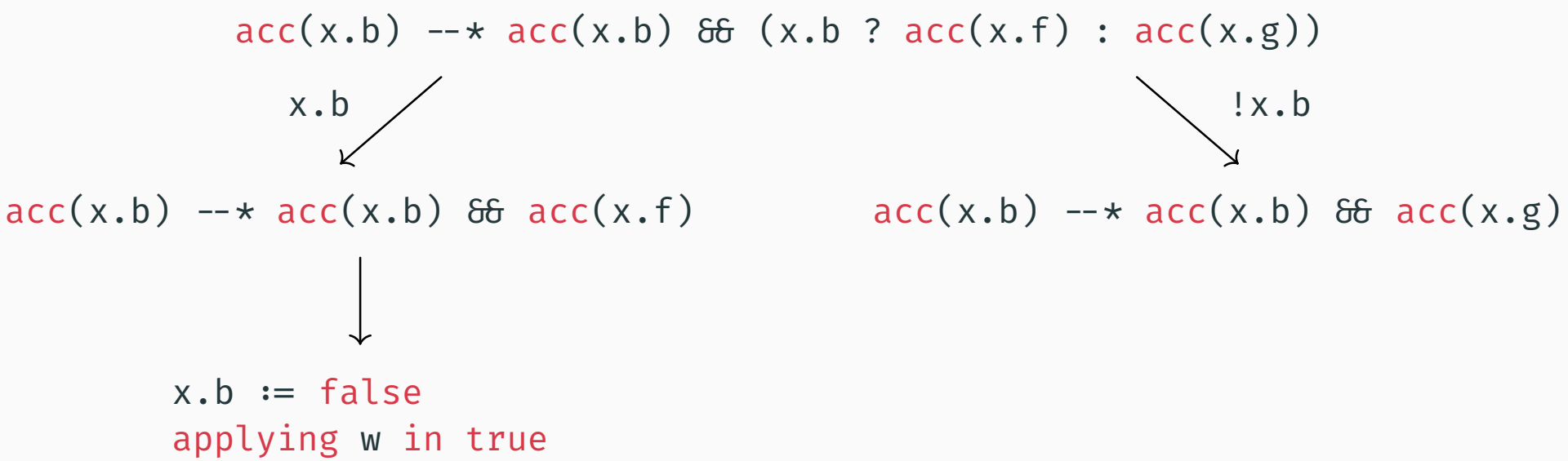


```
acc(x.b) --* acc(x.b) && acc(x.f)
```

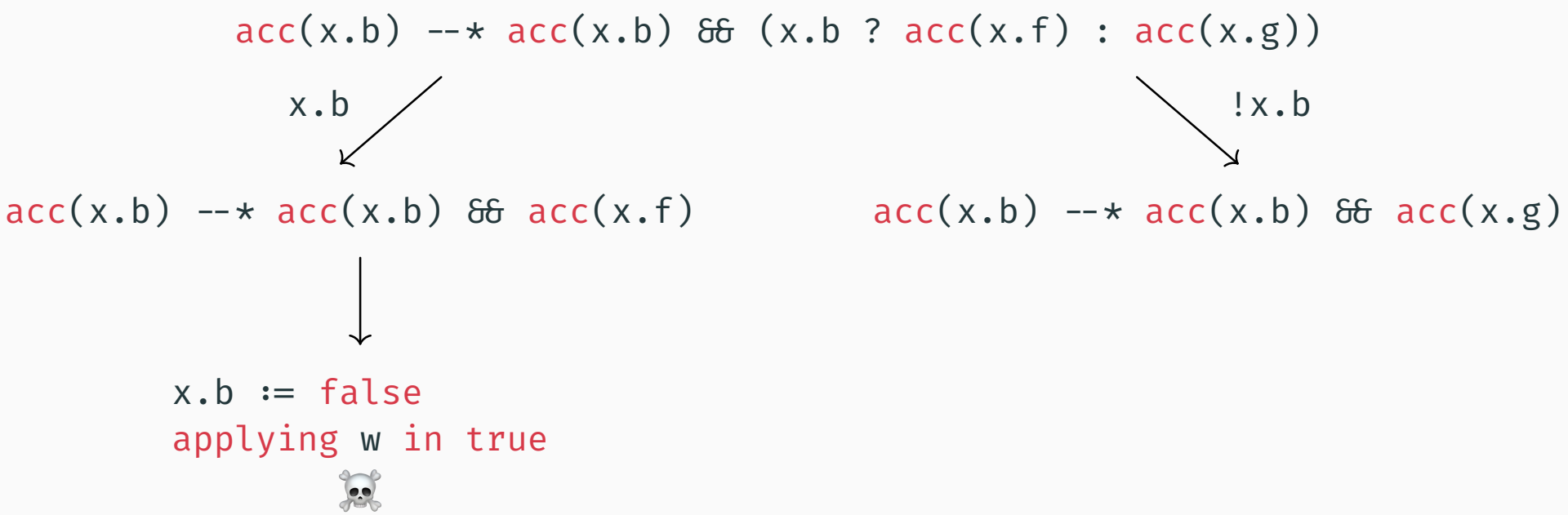
A program showcasing the unsoundness of the algorithm



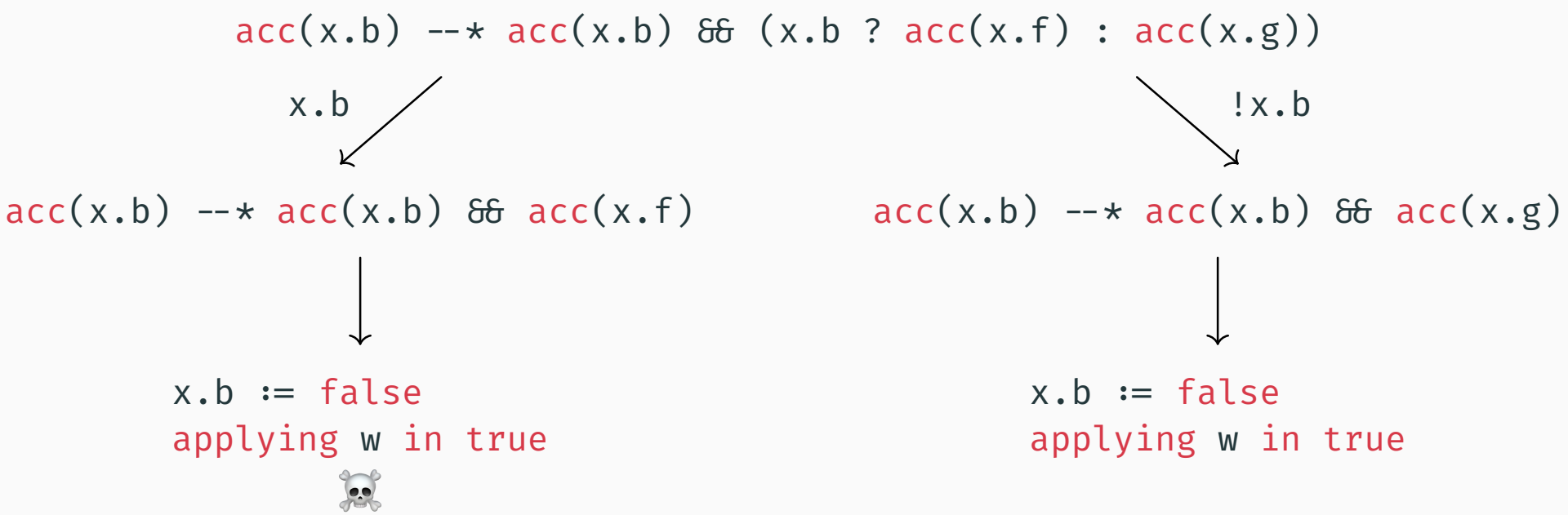
A program showcasing the unsoundness of the algorithm



A program showcasing the unsoundness of the algorithm

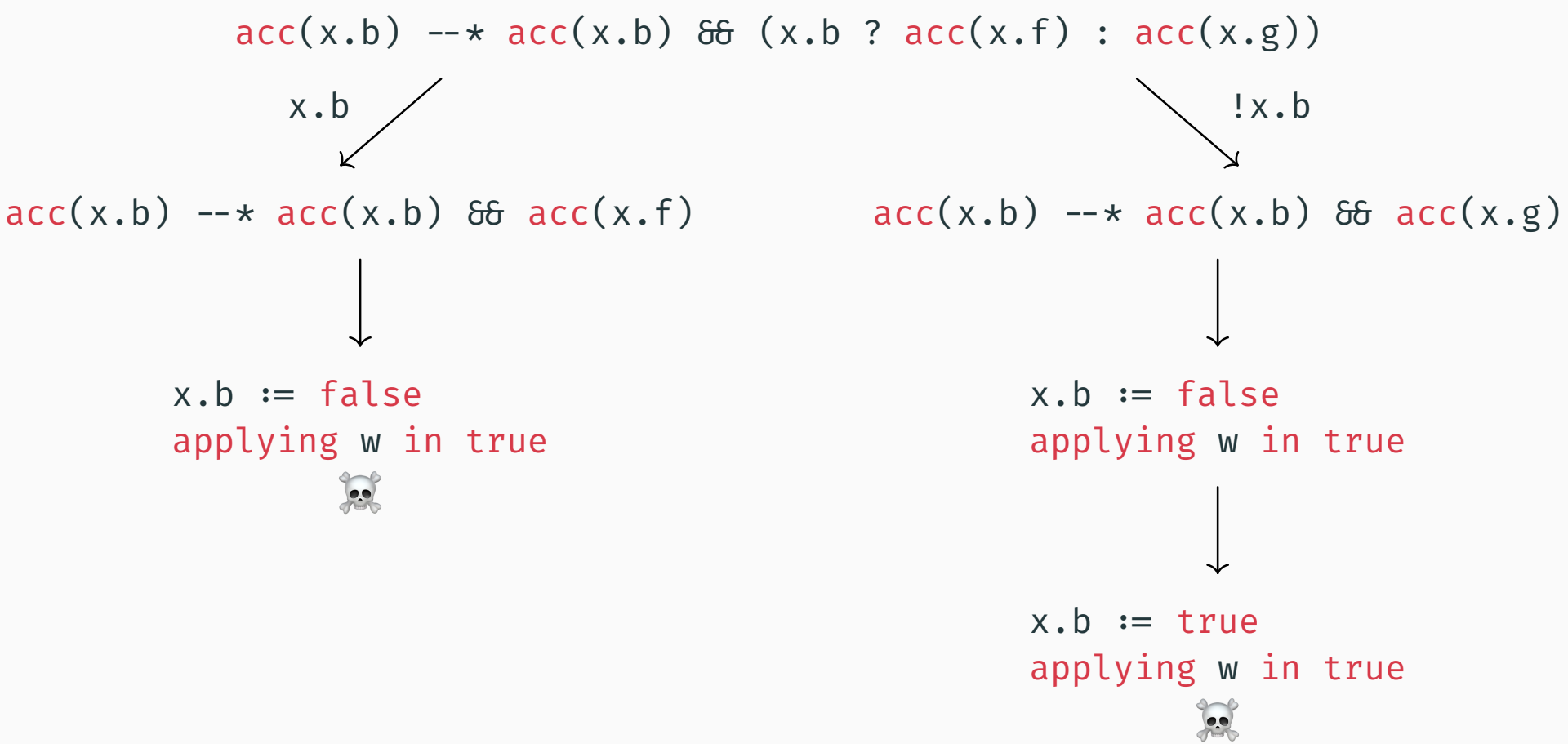


A program showcasing the unsoundness of the algorithm



A program showcasing the unsoundness of the algorithm

A program showcasing the unsoundness of the algorithm



Unsound: `assert false` succeeds in both branches!

A sound packaging algorithm

The fix

1. join all recorded branches (states and path conditions)
2. verify the rest of the program only once

A sound packaging algorithm

The fix

1. join all recorded branches (states and path conditions)
2. verify the rest of the program only once

Joining heaps

1. for each branch: conditionalize the heap by the BCs (example)
2. merge conditionalized heaps

A sound packaging algorithm

The fix

1. join all recorded branches (states and path conditions)
2. verify the rest of the program only once

Joining heaps

1. for each branch: conditionalize the heap by the BCs (example)
2. merge conditionalized heaps

Joining path conditions

⚠️ PCs cannot be propagated carelessly: `false --* false` should not propagate `false`

A sound packaging algorithm

The fix

1. join all recorded branches (states and path conditions)
2. verify the rest of the program only once

Joining heaps

1. for each branch: conditionalize the heap by the BCs (example)
2. merge conditionalized heaps

Joining path conditions

⚠ PCs cannot be propagated carelessly: `false --* false` should not propagate `false`

`MW_token`$(\texttt{w}, \texttt{t}_{\texttt{LHS}})$: "$\texttt{t}_{\texttt{LHS}}$ is a valid LHS for wand `w`"

A sound packaging algorithm

The fix

1. join all recorded branches (states and path conditions)
2. verify the rest of the program only once

Joining heaps

1. for each branch: conditionalize the heap by the BCs (example)
2. merge conditionalized heaps

Joining path conditions

⚠ PCs cannot be propagated carelessly: `false --* false` should not propagate `false`

`MW_token`$(\mathtt{w}, \mathtt{t}_{\mathtt{LHS}})$: "$\mathtt{t}_{\mathtt{LHS}}$ is a valid LHS for wand `w`"

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

A sound packaging algorithm

The fix

1. join all recorded branches (states and path conditions)
2. verify the rest of the program only once

Joining heaps

1. for each branch: conditionalize the heap by the BCs (example)
2. merge conditionalized heaps

Joining path conditions

⚠ PCs cannot be propagated carelessly: `false --* false` should not propagate `false`

`MW_token`$(\mathtt{w}, \mathtt{t}_{\mathtt{LHS}})$: "$\mathtt{t}_{\mathtt{LHS}}$ is a valid LHS for wand `w`"

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

Yield a token $\mathtt{MW\_token}(\mathtt{w}, \mathtt{t}_{\mathtt{LHS}})$ when applying `w` with concrete LHS $\mathtt{t}_{\mathtt{LHS}}$

Evaluation of the new encoding

Against the test suite

178 test files with wands (all passing with the previous encoding):

- 166 still verify
- 12 failing:

Evaluation of the new encoding

Against the test suite

178 test files with wands (all passing with the previous encoding):

- 166 still verify
- 12 failing:
  - 7: quantified wands not supported

Evaluation of the new encoding

Against the test suite

178 test files with wands (all passing with the previous encoding):

- 166 still verify
- 12 failing:
  - 7: quantified wands not supported
  - 2: should not have passed in the first place: **expected behavior**

Evaluation of the new encoding

Against the test suite

178 test files with wands (all passing with the previous encoding):

- 166 still verify
- 12 failing:
  - 7: quantified wands not supported
  - 2: should not have passed in the first place: **expected behavior**
  - 2: can verify using an experimental feature (added support for wands)

Evaluation of the new encoding

Against the test suite

178 test files with wands (all passing with the previous encoding):

- 166 still verify
- 12 failing:
  - 7: quantified wands not supported
  - 2: should not have passed in the first place: **expected behavior**
  - 2: can verify using an experimental feature (added support for wands)
  - 1: **completeness regression** (details)

Evaluation of the new encoding

Against the test suite

178 test files with wands (all passing with the previous encoding):

- 166 still verify
- 12 failing:
  - 7: quantified wands not supported
  - 2: should not have passed in the first place: **expected behavior**
  - 2: can verify using an experimental feature (added support for wands)
  - 1: **completeness regression** (details)
- + 2 new test files (for the soundness issue described and another one)

Evaluation of the new encoding

Against the test suite

178 test files with wands (all passing with the previous encoding):

- 166 still verify
- 12 failing:
  - 7: quantified wands not supported
  - 2: should not have passed in the first place: **expected behavior**
  - 2: can verify using an experimental feature (added support for wands)
  - 1: **completeness regression** (details)
- + 2 new test files (for the soundness issue described and another one)

Benchmark

On large verified codebases (Internet router [1] / verification library for security protocols [2])

Equivalent performance (details)

Modular functional specifications for wands

Attached facts by example (1)

The rationale: “attach” functional spec on top of an existing wand carrying permissions

Attached facts by example (1)

The rationale: “attach” functional spec on top of an existing wand carrying permissions

Classic wand

```
    list(cur)
    && len(cur) == len_cur
    && elems_positive(cur)
--* list(root)
    && len(root) == old(len(root))
    && elems_positive(root)
```

Attached facts by example (1)

The rationale: “attach” functional spec on top of an existing wand carrying permissions

Classic wand

```
    list(cur)
    && len(cur) == len_cur
    && elems_positive(cur)
--* list(root)
    && len(root) == old(len(root))
    && elems_positive(root)
```

Wand and attached facts

```
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_positive(cur))
          ⟹ elems_positive(root)
     to list(cur) --* list(root)
```

Attached facts by example (1)

The rationale: “attach” functional spec on top of an existing wand carrying permissions

Classic wand

```
    list(cur)
    && len(cur) == len_cur
    && elems_positive(cur)
--* list(root)
    && len(root) == old(len(root))
    && elems_positive(root)
```

Wand and attached facts

```
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_positive(cur))
          ⟹ elems_positive(root)
     to list(cur) --* list(root)
```

- **incremental** specification: first memory safety using the wands
- **modular** functional spec: each functional property specified and verified independently

Attached facts by example (2)

Attached facts come for free

```
while ( ... )
  invariant w
  invariant attached F to w
  invariant attached G to W
{
   ...
  package w {  ...  }
}
```

Attached facts by example (2)

Attached facts come for free

```
while ( ... )
  invariant w
  invariant attached F to w
  invariant attached G to W
{
   ...
  package w {  ...  }
}
```

Weakening properties

```
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. */ }
```

Attached facts by example (2)

Attached facts come for free

```
while ( ... )
  invariant w
  invariant attached F to w
  invariant attached G to W
{
   ...
  package w {  ...  }
}
```

Modular spec for wands

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


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

Weakening properties

```
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. */ }
```

Attached facts by example (2)

Attached facts come for free

```
while ( ... )
  invariant w
  invariant attached F to w
  invariant attached G to W
{
   ...
  package w {  ...  }
}
```

Weakening properties

```
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. */ }
```

Modular spec for wands

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


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

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


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

Semantics of attached facts

Context: `attached e to A --* B`

- `e` is a (pure) boolean expression (no `acc( ... )`, predicates, or wands)

Semantics of attached facts

Context: `attached e to A --* B`

- `e` is a (pure) boolean expression (no `acc( ... )`, predicates, or wands)
- `e` is **well-framed** in `A --* B && e` (consequence: `e` can mention both RHS and `old[lhs](LHS)`)

Semantics of attached facts

Context: `attached e to A --* B`

- `e` is a (pure) boolean expression (no `acc( ... )`, predicates, or wands)
- `e` is **well-framed** in `A --* B && e` (consequence: `e` can mention both RHS and `old[lhs](LHS)`)

Generic evaluation procedure

Setup to evaluate `e` as in `A --* B && e`:

1. produce artificial LHS t_A
2. produce artificial RHS $\texttt{mwsf}(t_A)$
3. eval `e` in the resulting state

Semantics of attached facts

Context: `attached e to A --* B`

- `e` is a (pure) boolean expression (no `acc( ... )`, predicates, or wands)
- `e` is **well-framed** in `A --* B && e` (consequence: `e` can mention both RHS and `old[lhs](LHS)`)

Generic evaluation procedure

Setup to evaluate `e` as in `A --* B && e`:

1. produce artificial LHS t_A
2. produce artificial RHS $\texttt{mwsf}(t_A)$
3. eval `e` in the resulting state: **the resulting term t_e depends on t_A**

Semantics of attached facts

Context: `attached e to A --* B`

- `e` is a (pure) boolean expression (no `acc( ... )`, predicates, or wands)
- `e` is **well-framed** in `A --* B && e` (consequence: `e` can mention both RHS and `old[lhs](LHS)`)

Generic evaluation procedure

Setup to evaluate `e` as in `A --* B && e`:

1. produce artificial LHS t_A
2. produce artificial RHS $\texttt{mwsf}(t_A)$
3. eval `e` in the resulting state: **the resulting term t_e depends on t_A**

Semantics of attached facts

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

- does not depend on t_A
- $\texttt{MW\_token}(\texttt{w}, t_A)$ allows accessing the wand definition

Evaluation of attached facts

On examples

Successfull rewrites of examples from papers [3], [4] + new ones (*e.g.* `delete_last.vpr`, `stalin-sort.vpr`)

Evaluation of attached facts

On examples

Successfull rewrites of examples from papers [3], [4] + new ones (*e.g.* `delete_last.vpr`, `stalin-sort.vpr`)

Case-study on large verified codebases [1], [2]

Inconclusive: most wands are `predicate_1( ... ) --* predicate_2( ... )`

- a data structure and its associated invariants should not be separated
- intricate and nested permissions/functional spec: hard to rewrite

Evaluation of attached facts

On examples

Successfull rewrites of examples from papers [3], [4] + new ones (*e.g.* `delete_last.vpr`, `stalin-sort.vpr`)

Case-study on large verified codebases [1], [2]

Inconclusive: most wands are `predicate_1( ... ) --* predicate_2( ... )`

- a data structure and its associated invariants should not be separated
- intricate and nested permissions/functional spec: hard to rewrite

Early experiments with Prusti [5], [6]

Very promising on a few examples (*e.g.* binary search tree implem), wands naturally appear in Rust

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

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

```
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)
```

Accessing the footprint using attached expressions

Why is that different?

1. Specifications are **boolean assertions** (require an ~~integer as a precondition~~)

Why is that different?

1. Specifications are **boolean assertions** (require an ~~integer as a precondition~~)
2. Properties can be derived from attached facts **only once the wand has been applied**

Why is that different?

1. Specifications are **boolean assertions** (require an integer as a precondition)
2. Properties can be derived from attached facts **only once the wand has been applied**

Examples

Accessing the second element of a pair:

```
x.snd := 0
package acc(x.fst) --* pair(x) {  ...  }
var v: Int := attachedexp unfolding pair(x) in x.snd to W
assert v == 0
```

Why is that different?

1. Specifications are **boolean assertions** (~~require an integer as a precondition~~)
2. Properties can be derived from attached facts **only once the wand has been applied**

Examples

Accessing the second element of a pair:

```
x.snd := 0
package acc(x.fst) --* pair(x) { ... }
var v: Int := attachedexp unfolding pair(x) in x.snd to W
assert v == 0
```

(Computed) length of the prefix list:

```
while (...)
  invariant list(cur) --* list(root)
  invariant ...
  invariant (attachedexp len(root) - old[lhs](len(cur)) to W) == i
{ ... }
```

Why is that different?

1. Specifications are **boolean assertions** (~~require an integer as a precondition~~)
2. Properties can be derived from attached facts **only once the wand has been applied**

Examples

Accessing the second element of a pair:

```
x.snd := 0
package acc(x.fst) --* pair(x) { ... }
var v: Int := attachedexp unfolding pair(x) in x.snd to W
assert v == 0
```

(Computed) length of the prefix list:

```
while ( ... )
  invariant list(cur) --* list(root)
  invariant attachedexp_valid len(root) - old[lhs](len(cur)) to W
  invariant      (attachedexp len(root) - old[lhs](len(cur)) to W) == i
{ ... }
```

Validity conditions and semantics of attached expressions

Semantics

$$[\![\texttt{attachedexp e to w}]\!] := [\![\texttt{e}]\!] \text{ checking the conditions below}$$

Validity conditions and semantics of attached expressions

Semantics

$$[\![\texttt{attachedexp e to w}]\!] := [\![\texttt{e}]\!] \text{ checking the conditions below}$$

Independence from the LHS

$$[\![\texttt{attachedexp\_valid e to w}]\!] \stackrel{\text{def}}{=}$$

$$\forall \texttt{t}_1, \texttt{t}_2 : \texttt{MW\_token}(\texttt{w}, \texttt{t}_1) \wedge \texttt{MW\_token}(\texttt{w}, \texttt{t}_2) \Rightarrow [\![\texttt{e}]\!][\texttt{t}_\texttt{A} := \texttt{t}_1] = [\![\texttt{e}]\!][\texttt{t}_\texttt{A} := \texttt{t}_2]$$

Validity conditions and semantics of attached expressions

Semantics

$$[\![\texttt{attachedexp e to w}]\!] := [\![\texttt{e}]\!] \text{ checking the conditions below}$$

Independence from the LHS

$$[\![\texttt{attachedexp\_valid e to w}]\!] \stackrel{\text{def}}{=}$$

$$\forall \texttt{t}_1, \texttt{t}_2 : \texttt{MW\_token}(\texttt{w}, \texttt{t}_1) \land \texttt{MW\_token}(\texttt{w}, \texttt{t}_2) \Rightarrow [\![\texttt{e}]\!][\texttt{t}_\texttt{A} := \texttt{t}_1] = [\![\texttt{e}]\!][\texttt{t}_\texttt{A} := \texttt{t}_2]$$

- **not sufficient**: need a token to use that property: check the wand is not vacuously true

Appliability

Recall $\sigma \vDash A \mathrel{-\!\!*} B \iff \forall \sigma_A : \sigma_A \perp \sigma \land \sigma_A \vDash A \Rightarrow \sigma_A \uplus \sigma \vDash B$

A wand is *appliable* if there exists a valid LHS: $\exists \sigma_A : \sigma_A \perp \sigma \land \sigma_A \vDash A$

Validity conditions and semantics of attached expressions

Semantics

$$[\![\texttt{attachedexp e to w}]\!] := [\![\texttt{e}]\!] \text{ checking the conditions below}$$

Independence from the LHS

$$[\![\texttt{attachedexp\_valid e to w}]\!] \stackrel{\text{def}}{=}$$

$$\forall \texttt{t}_1, \texttt{t}_2 : \texttt{MW\_token}(\texttt{w}, \texttt{t}_1) \wedge \texttt{MW\_token}(\texttt{w}, \texttt{t}_2) \Rightarrow [\![\texttt{e}]\!][\texttt{t}_\texttt{A} := \texttt{t}_1] = [\![\texttt{e}]\!][\texttt{t}_\texttt{A} := \texttt{t}_2]$$

- **not sufficient**: need a token to use that property: check the wand is not vacuously true

Appliability

Recall $\sigma \vDash A \mathrel{-\!\!*} B \iff \forall \sigma_A : \sigma_A \perp \sigma \wedge \sigma_A \vDash A \Rightarrow \sigma_A \uplus \sigma \vDash B$

A wand is *appliable* if there exists a valid LHS: $\exists \sigma_A : \sigma_A \perp \sigma \wedge \sigma_A \vDash A$

Automation/integration in Silicon = future work

Conclusion

A sound algorithm and encoding for packaging path conditions

- introducing **magic wand tokens**

Attached facts

- enable **incremental** and **modular** functional specification of magic wands
- highly **automated** (come for free when packaging)
- promising application to Rust in Prusti

Attached expressions

- design space exploration
- **sufficient validity conditions** ✅ independence from the LHS ⚙️ appliability

Future work

Application to Rust Automation of appliability Formalization of this work

All artifacts available on github `remigerme/silver` and `remigerme/silicon` (see branches)

Example of joining heaps

Branch condition	Remaining heap	Conditionalized heap	Final joined heap
x.b - First(mwsf(t_A))	(x.g → ... # 1)	(x.g → ... # x.b ? 1 : 0)	(x.g → ... # x.b ? 1 : 0,
!x.b - !First(mwsf(t_A))	(x.f → ... # 1)	(x.f → ... # !x.b ? 1 : 0)	x.f → ... # !x.b ? 1 : 0)

Example of joining heaps

Branch condition	Remaining heap	Conditionalized heap	Final joined heap
`x.b` - `First(mwsf(`t_A`))`	`(x.g → ... # 1)`	`(x.g → ... # x.b ? 1 : 0)`	`(x.g → ... # x.b ? 1 : 0,`
`!x.b` - `!First(mwsf(`t_A`))`	`(x.f → ... # 1)`	`(x.f → ... # !x.b ? 1 : 0)`	`x.f → ... # !x.b ? 1 : 0)`

Wait, t_A is dangling... fine: this prevents the solver from accessing `x.f` and `x.g`.

Details of the tests

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 [7])
`conditionals3.vpr` `PackageStateConsolidator.vpr`	can be successfully verified by enabling the SE-PC flag
`packaging_1.vpr`	true completeness regression

`conditionals.vpr` and `SnapshotsBranching.vpr`

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

`packaging_1.vpr`

```
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
  }
}
```

Go back

Benchmarking the new encoding

Benchmarking on [1], [2] using hyperfine [8][1]:

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 1: 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 2: Hyperfine benchmark results for the security protocol verification library (15 runs per encoding).

Go back

[1] All source files and the benchmark script can be found on my repository https://github.com/remigerme/silicon/blob/9ca9b342a9ad12dc888b573d7ca046ba9f749783/remi-demo

Generic evaluation procedure

Context: `attached e to A --* B`. Setup to evaluate e as in `A --* B && e`:

1. produce artificial LHS t_A
2. produce artificial RHS $mwsf(t_A)$
3. eval e in the resulting state: **the resulting term t_e depends on t_A**

```
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)
        )
      )
    )
  )
```

Bibliography

[1] 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.

[2] 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.

[3] 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.

[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 *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.

Bibliography

[6] 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.

[7] 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.

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