Lightweight Support for Magic Wands in an Automatic Verifier

ETH zürich

Malte Schwerhoff and Alexander J. Summers

10th July 2015, ECOOP, Prague

Frame Problem

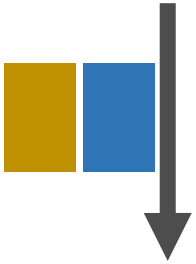
Modular, static verification of imperative programs
↳ Frame problem: Which memory locations change?

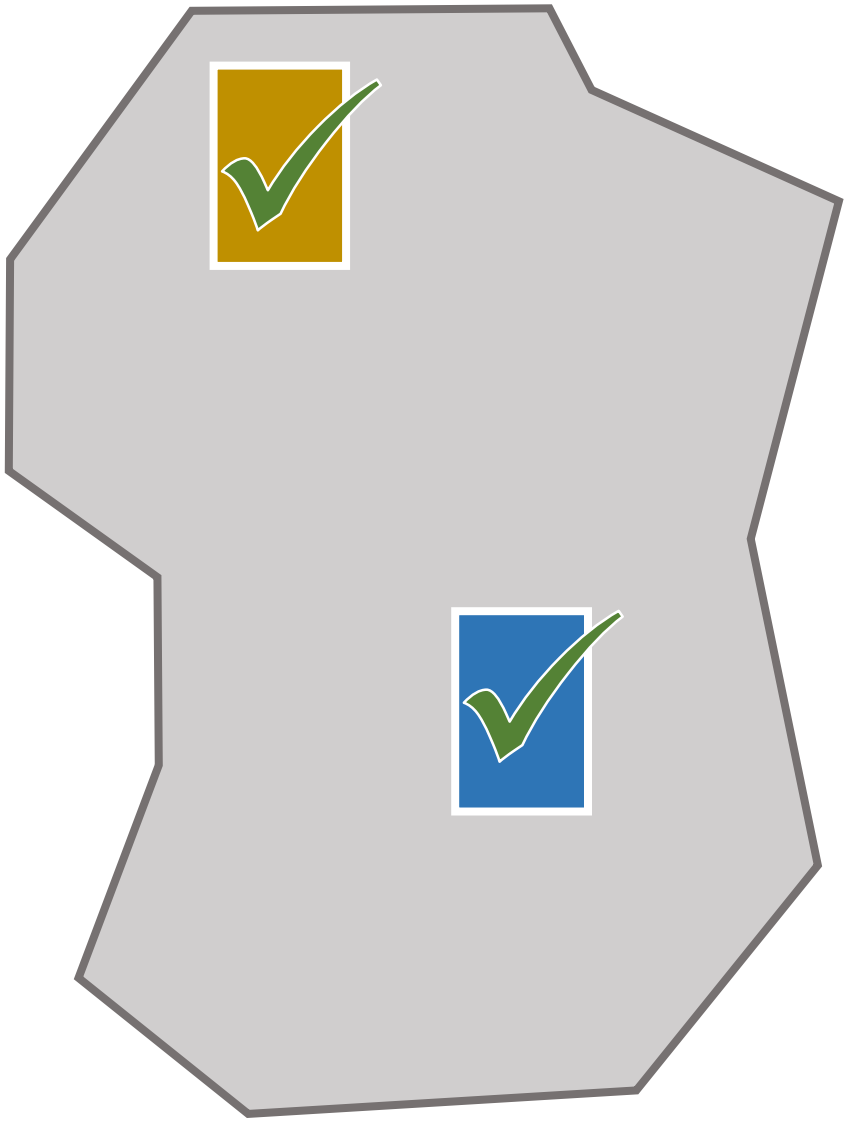
Automated verification: Pre-/Postconditions, invariants, ghost code

Well-known approach: Permissions (≈ Separation Logic)

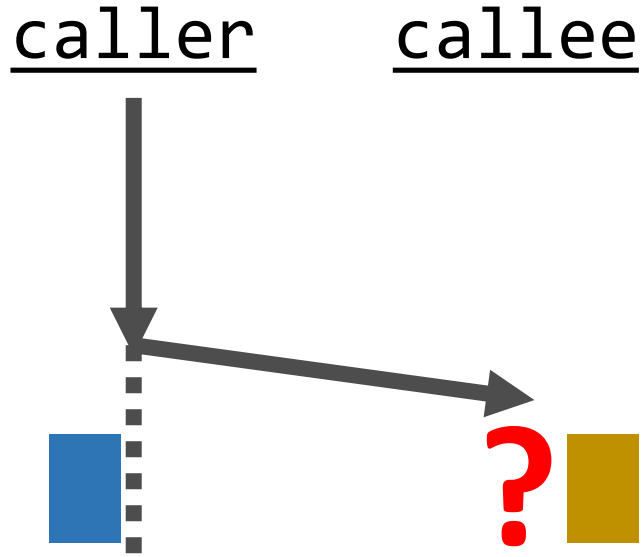
Permission Transfer

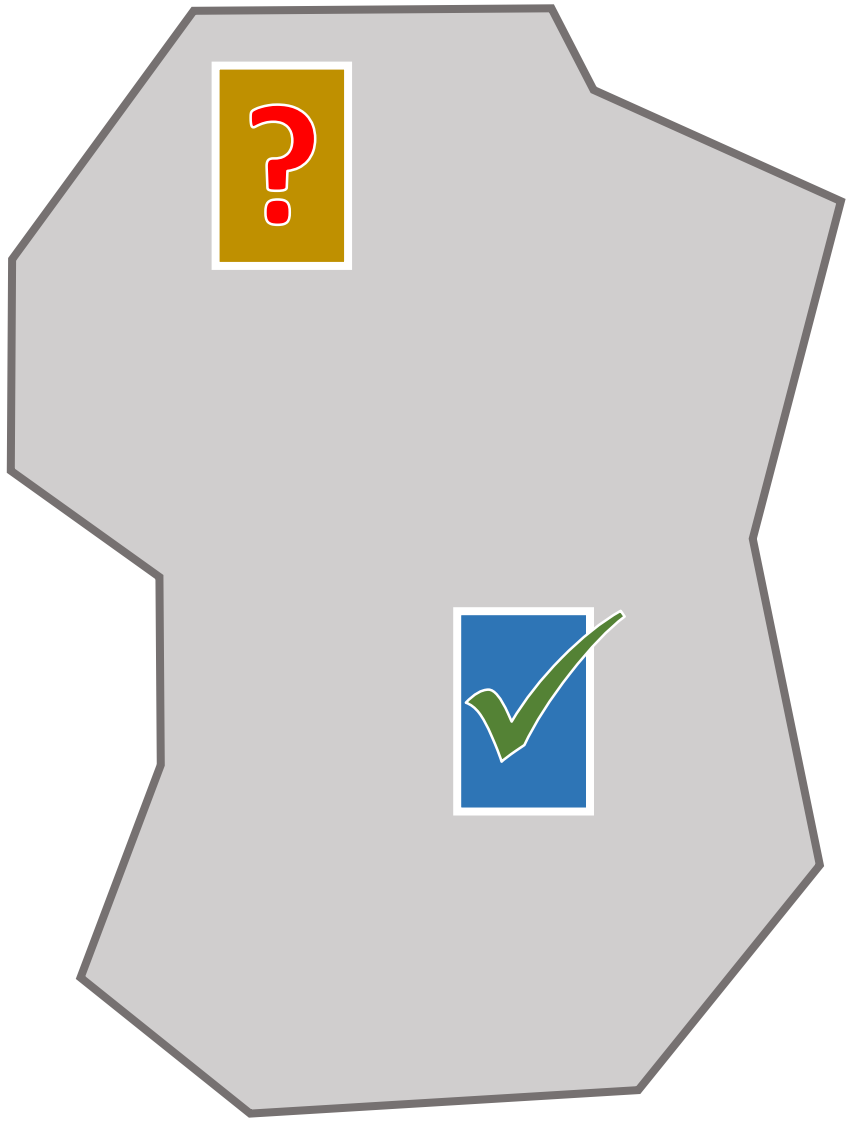
caller callee



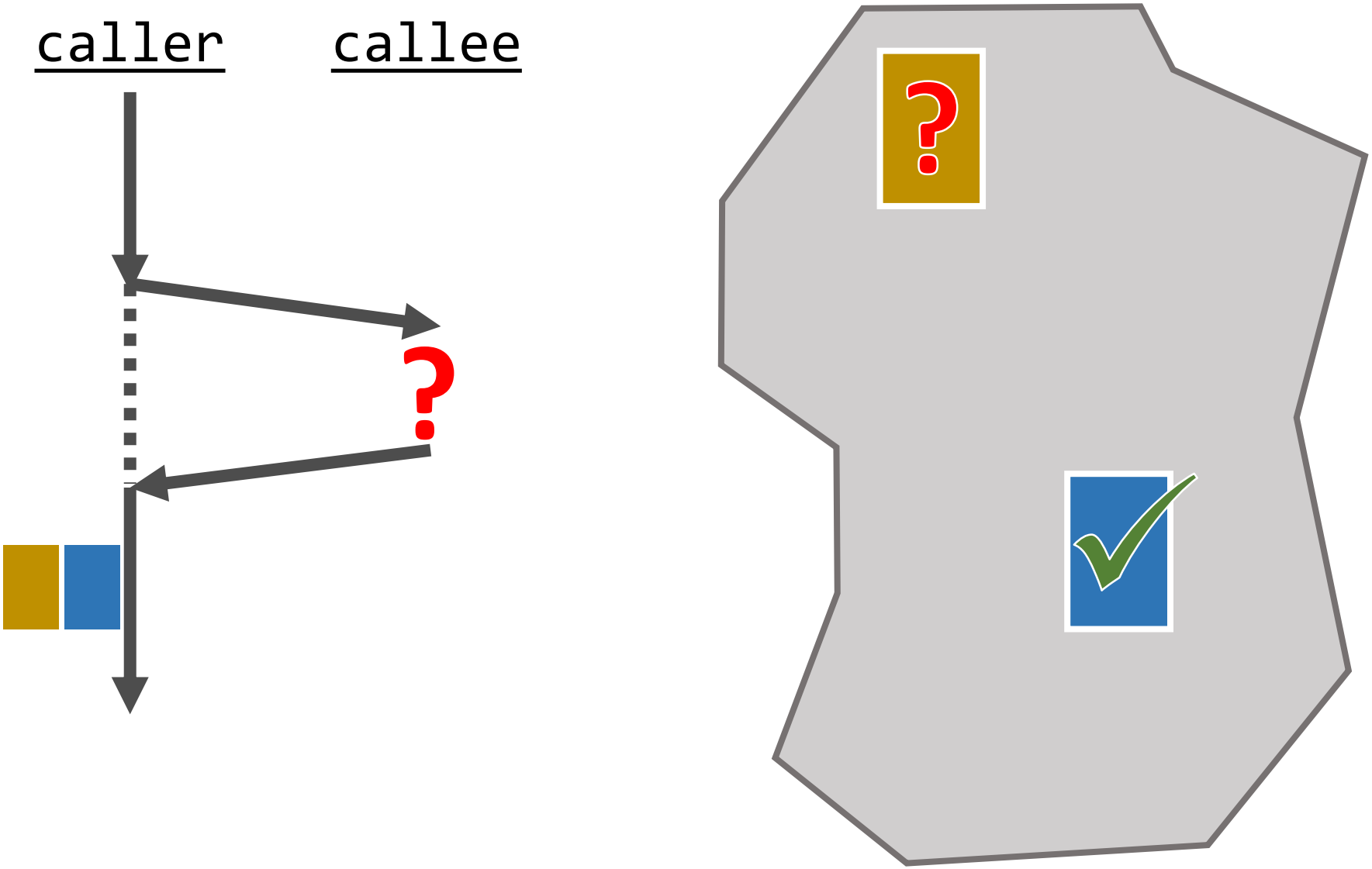


Permission Transfer





Permission Transfer



Syntax & Properties

Permissions (≈ Separation Logic)

Logical properties


```
acc(x.f) * x.f == 0
```

(≈ `x.f ↦ 0`)

Disjointness: x ≠ y

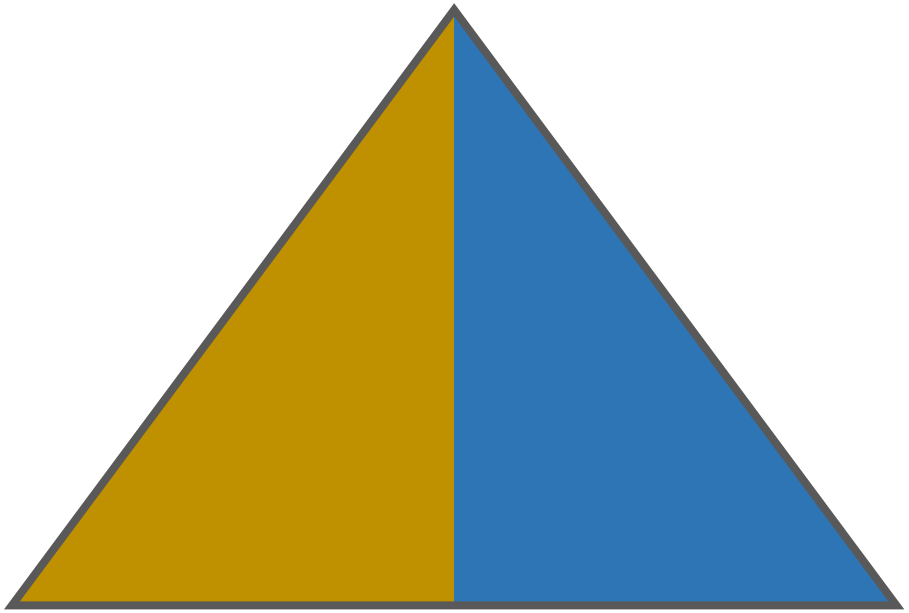
```
acc(x.f) * acc(y.f)
```

(≈ `x.f ↦ _ * y.f ↦ _`)

Separating Conjunction

$A * B$

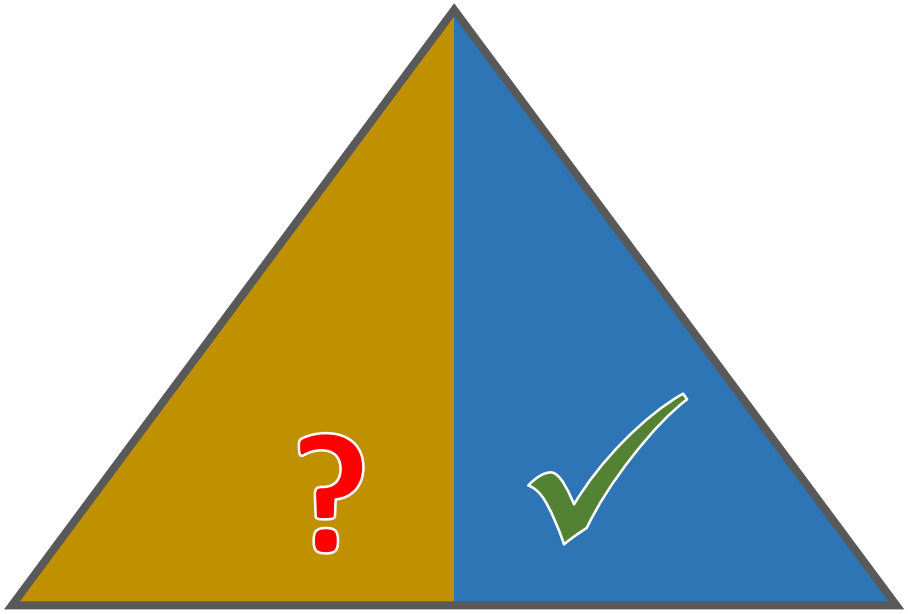
describes the current state in terms of disjoint substates



Separating Conjunction

$A * B$

describes the current state in terms of disjoint substates



Is at the heart of verifiers based on separation logic

Magic Wands

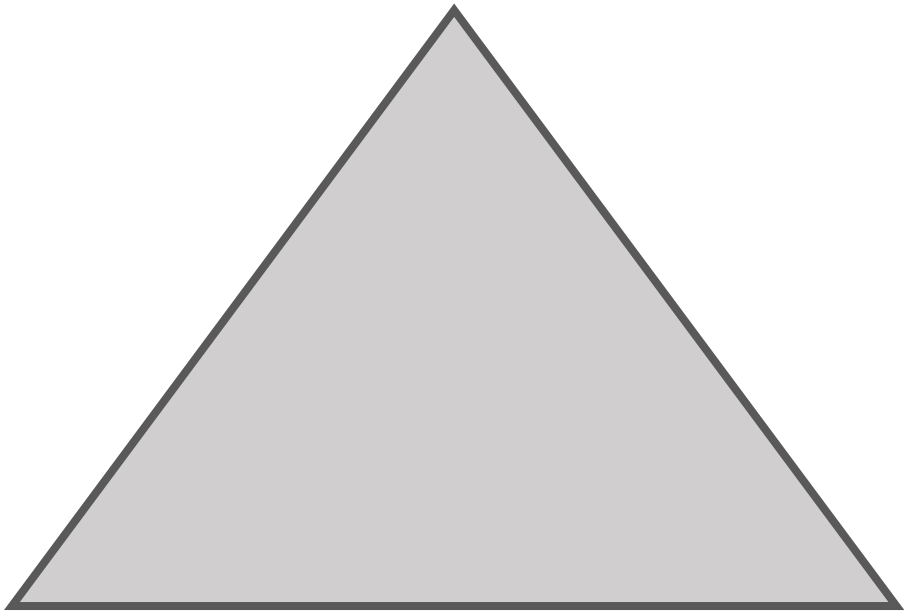
$$A \mathrel{-\!\!*} B$$

describes hypothetical states

Read as a promise: "In any state, if you provide A, then you will get B"

Partial Data Structures

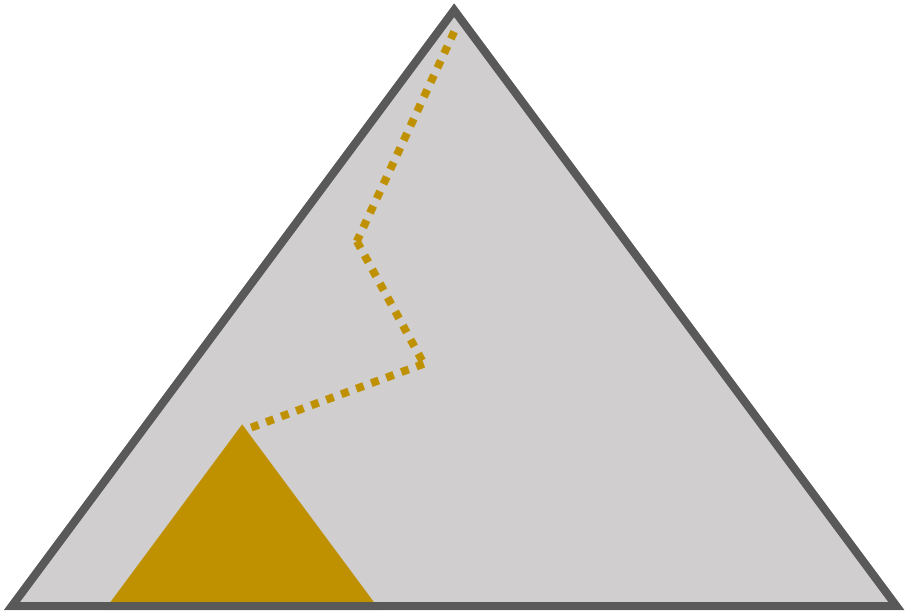
Scenario: Iteratively traverse a recursively defined tree (Verification Challenge at VerifyThis@FM'12)



Partial Data Structures

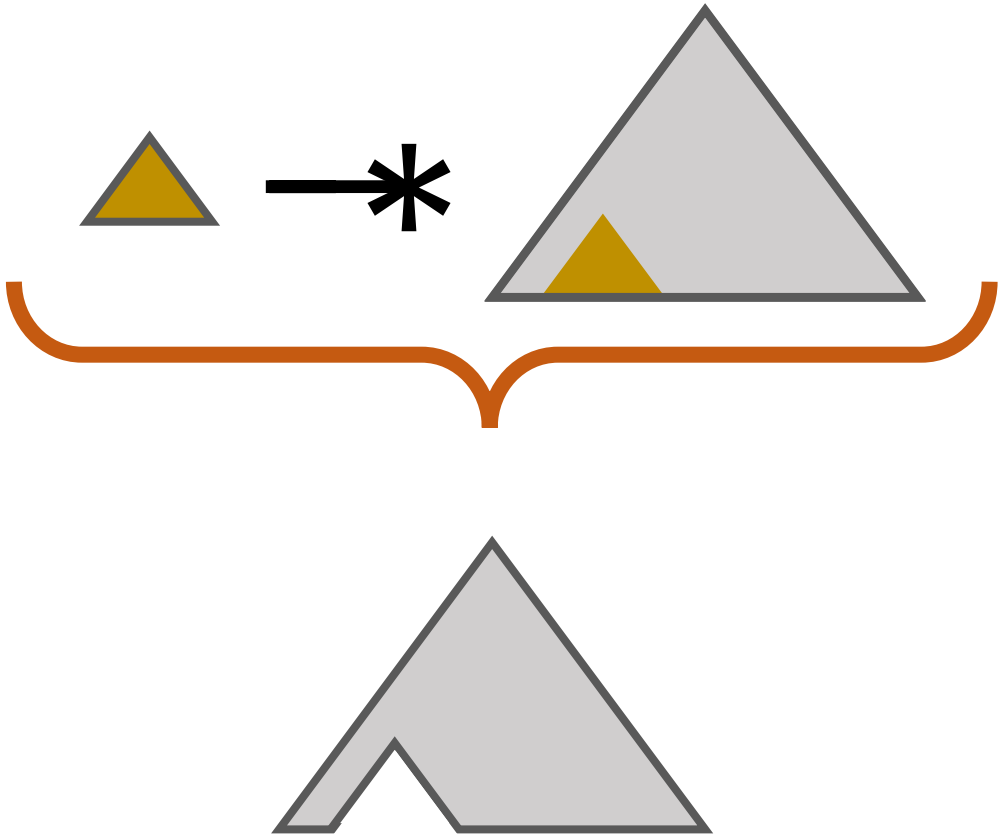
Scenario: Iteratively traverse a recursively defined tree

↳Loop invariant: Describe partial data structure



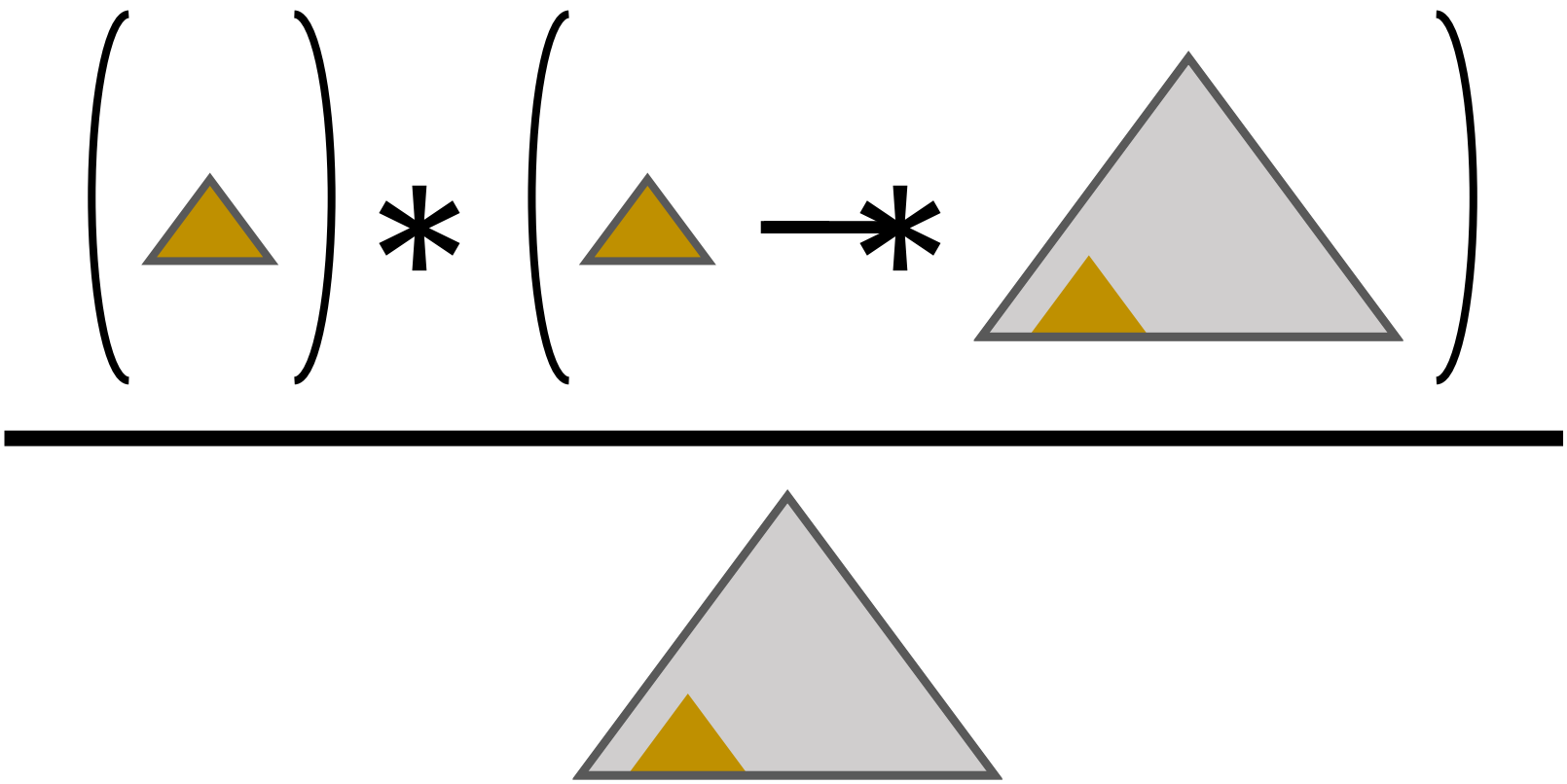
Partial Data Structures as Magic Wands

Indirectly describe partial data structure as a promise



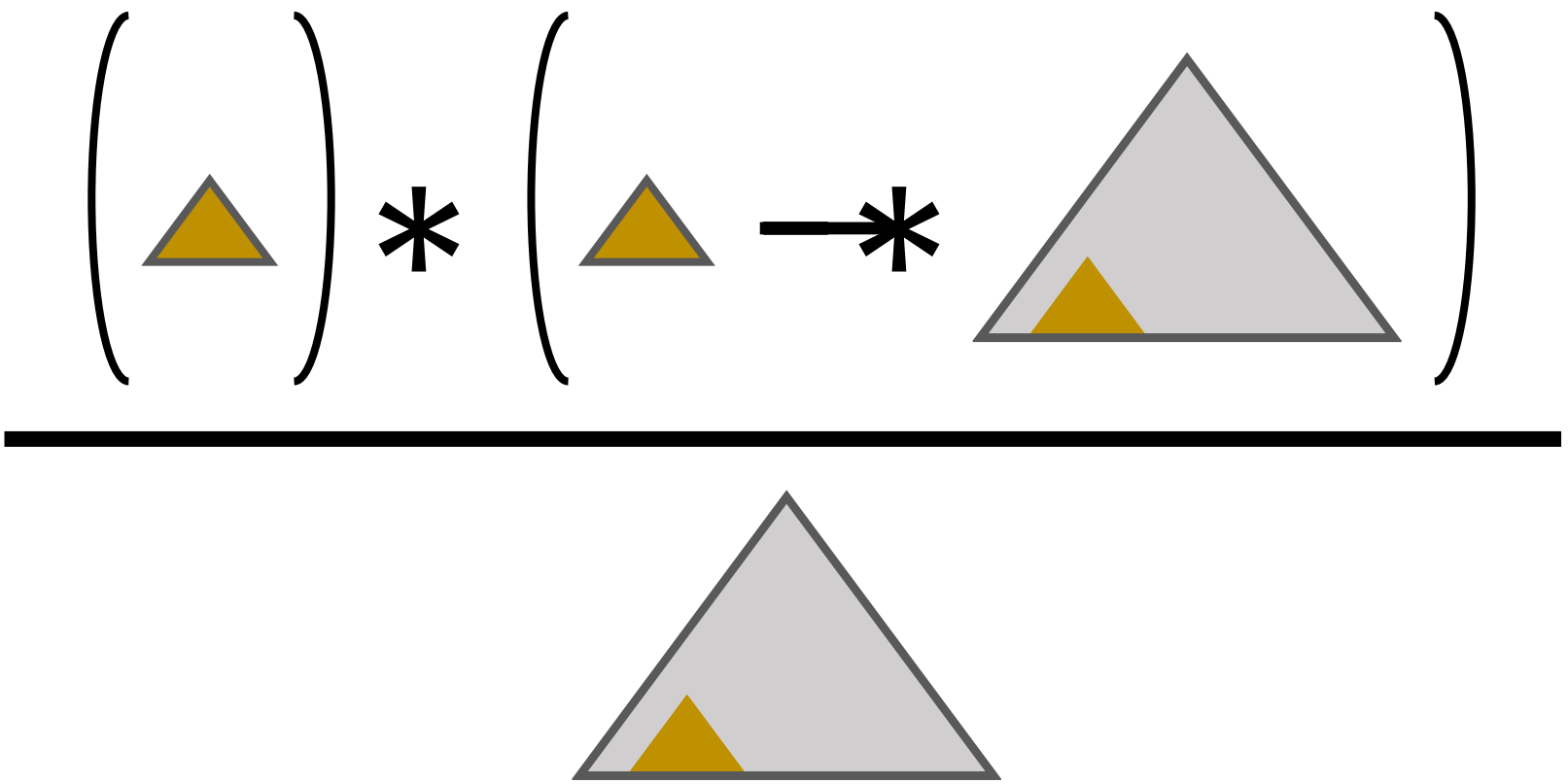
Partial Data Structures as Magic Wands

Modus-Ponens-like rule makes promise applicable



Partial Data Structures as Magic Wands

$$\sigma \vDash A \mathrel{-\!\!*} B \quad \Leftrightarrow \quad \forall\, \sigma' \cdot (\sigma' \vDash A \;\Rightarrow\; \sigma \uplus \sigma' \vDash B)$$



Magic Wands in Proofs and Tools

Used in various pen & paper proofs

- Partial data structures
- Usage protocols for data structures
- Synchronisation barriers
- …

Typically* not supported in automatic verifiers

$$\sigma \vDash A \mathrel{-\!\!*} B \quad \Leftrightarrow \quad \forall\, \sigma' \cdot (\sigma' \vDash A \;\Rightarrow\; \sigma \uplus \sigma' \vDash B)$$

* Only exception we are aware of is VerCors; developed in parallel

Automating Magic Wand Reasoning

Entailment of magic wand formulas is undecidable

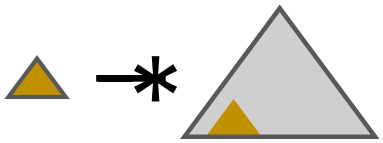
↳ Lightweight user guidance to direct verification

Guidance: Ghost Operations + Specifications

Make a promise

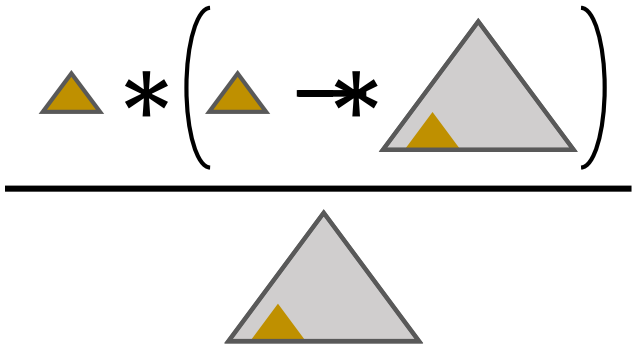
Pass it around

Use it



`package` $A \mathrel{-\!\!*} B$

Opaque resource; Specifications

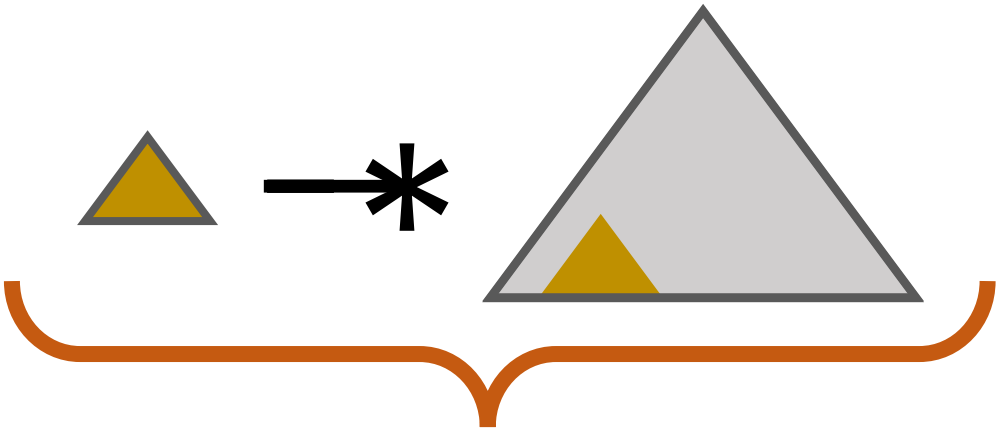


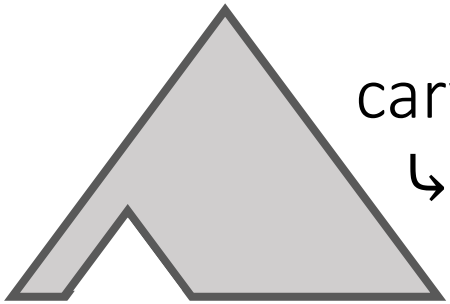
`apply` $A \mathrel{-\!\!*} B$

Guidance: Ghost Operations + Specifications

Footprints of $A \mathrel{-\!\!*} B$

Permissions guaranteeing that giving up $A * (A \mathrel{-\!\!*} B)$ and obtaining B is sound



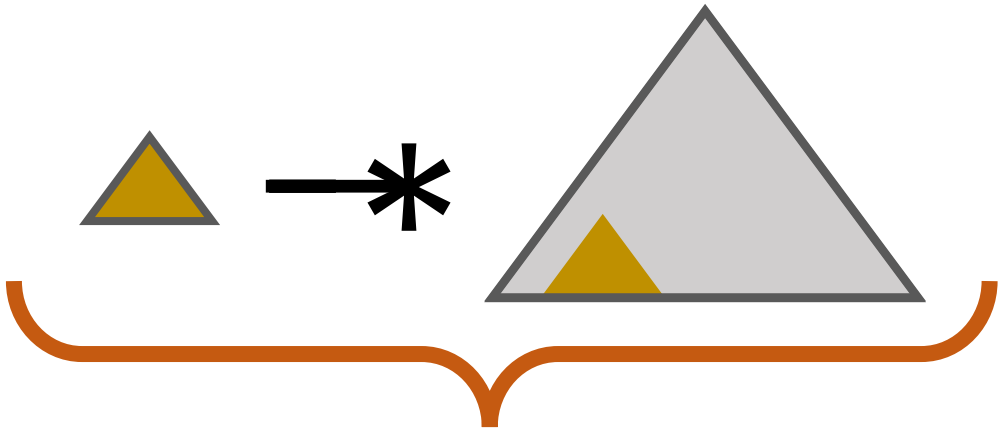


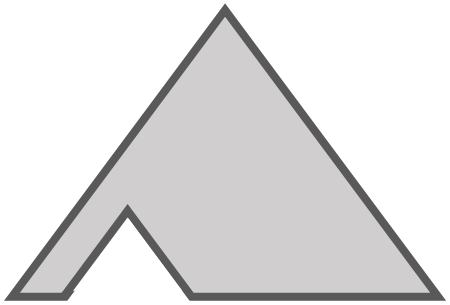
carved out

↳ effectively immutable

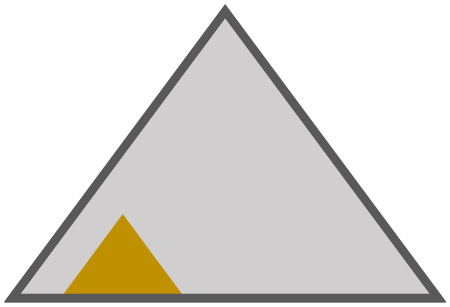
Footprints of $A \twoheadrightarrow B$

Footprints are not unique





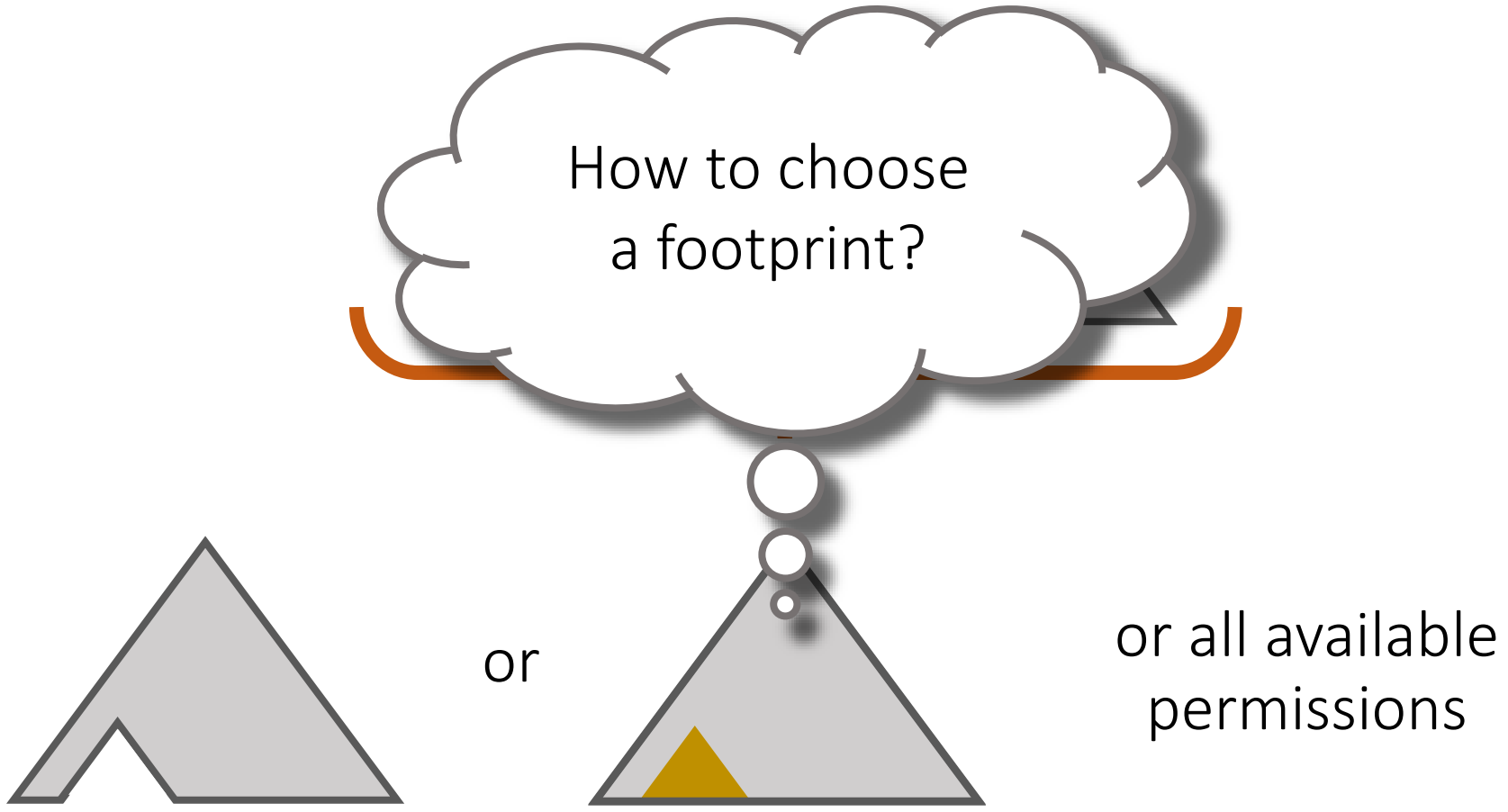
or



or all available permissions

Footprints of $A \twoheadrightarrow B$

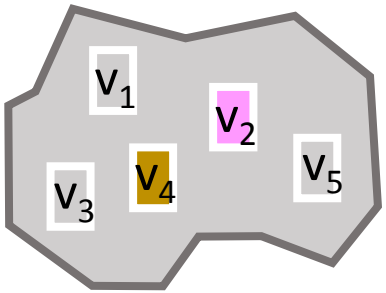
Footprints are not unique



Footprint Computation Algorithm: Setup

$$\texttt{package}\ \ A \twoheadrightarrow (B_1 * B_2 * \ldots * B_n)$$

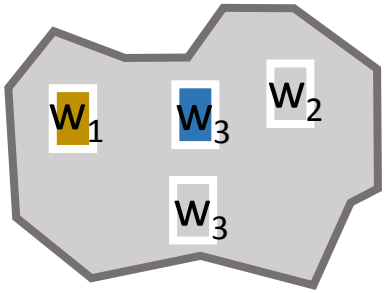
0. given current state



Footprint Computation Algorithm: Setup

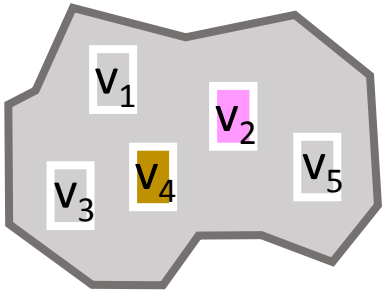
$$\texttt{package}\ \ A \twoheadrightarrow (B_1 * B_2 * \ldots * B_n)$$

1. create LHS state



$\models A$

current state

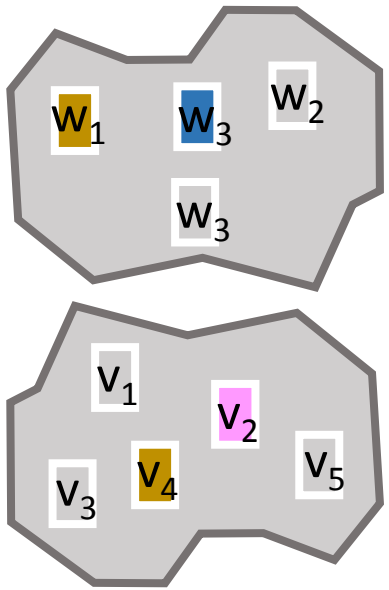


Footprint Computation Algorithm: Setup

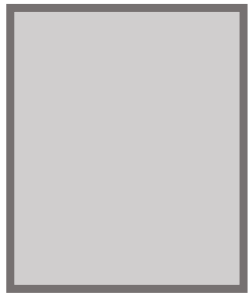
$$\texttt{package}\ \ A \twoheadrightarrow (B_1 * B_2 * \ldots * B_n)$$

LHS state

current state



$\models A$

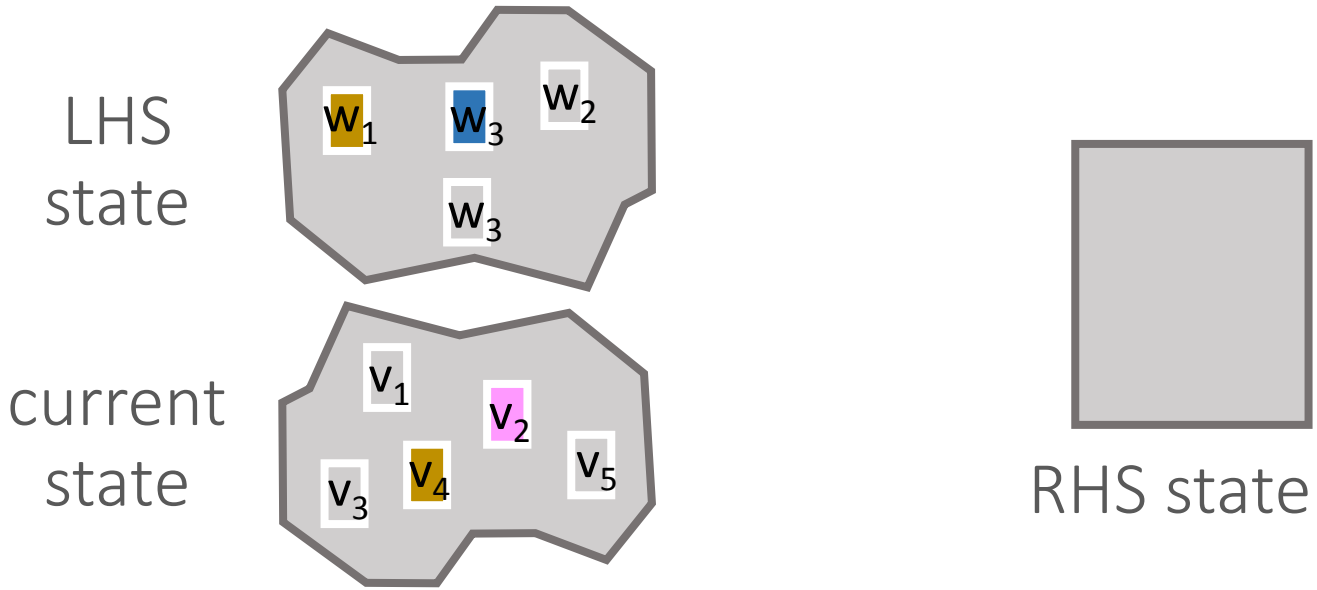


2. create empty RHS state

Footprint Computation Algorithm: Execution

$$\texttt{package}\ \ A \twoheadrightarrow (B_1 * B_2 * \ldots * B_n)$$

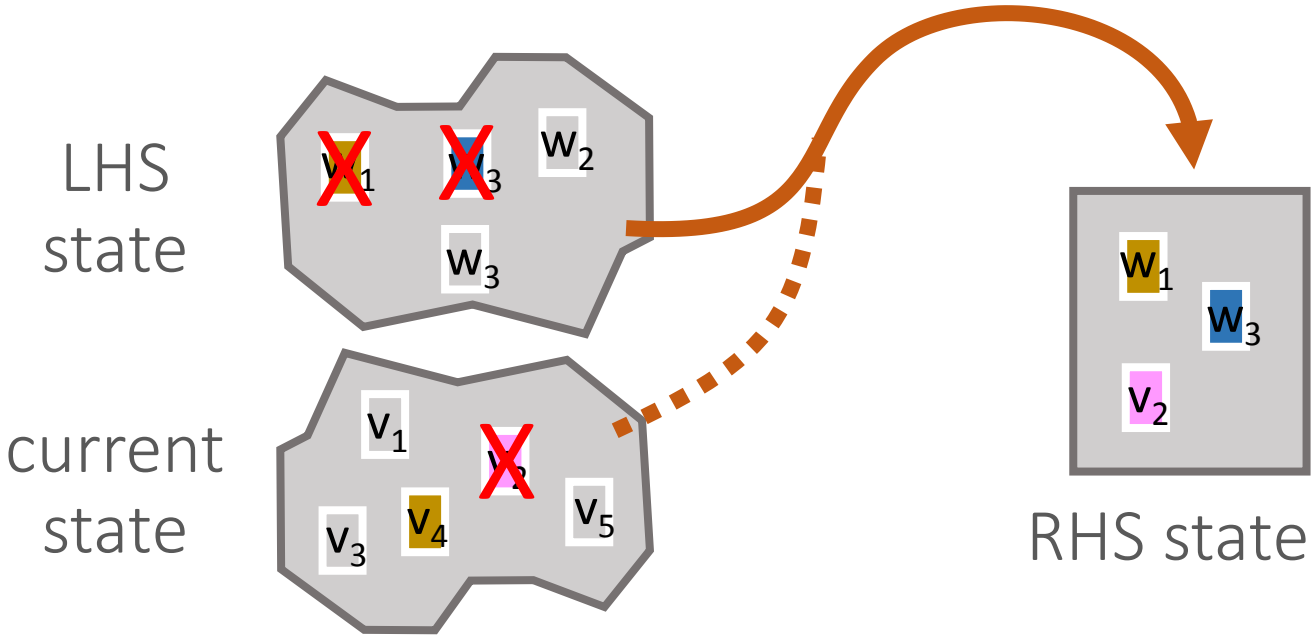
3. iterate over B_i's:



Footprint Computation Algorithm: Execution

$$\texttt{package}\ \ A \twoheadrightarrow (B_1 * B_2 * \ldots * B_n)$$

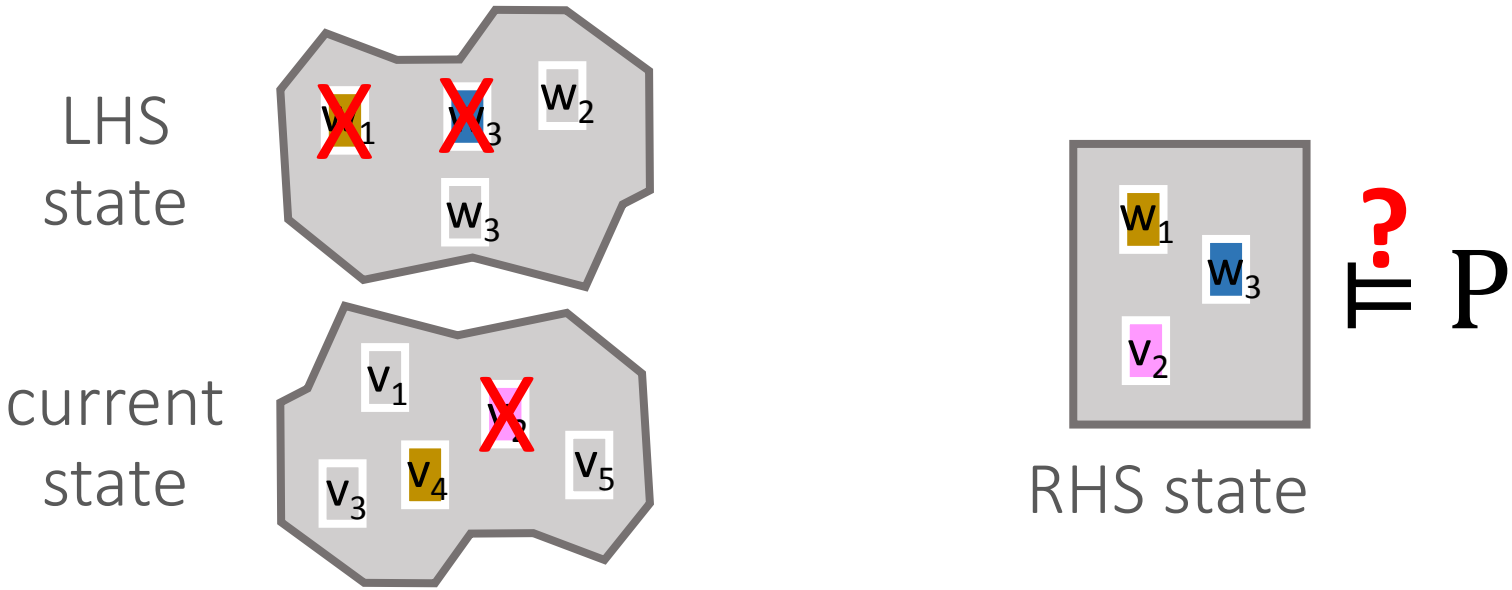
3. iterate over B_i's: If B_i is `acc(x.f)` then transfer permissions and assumptions



Footprint Computation Algorithm: Execution

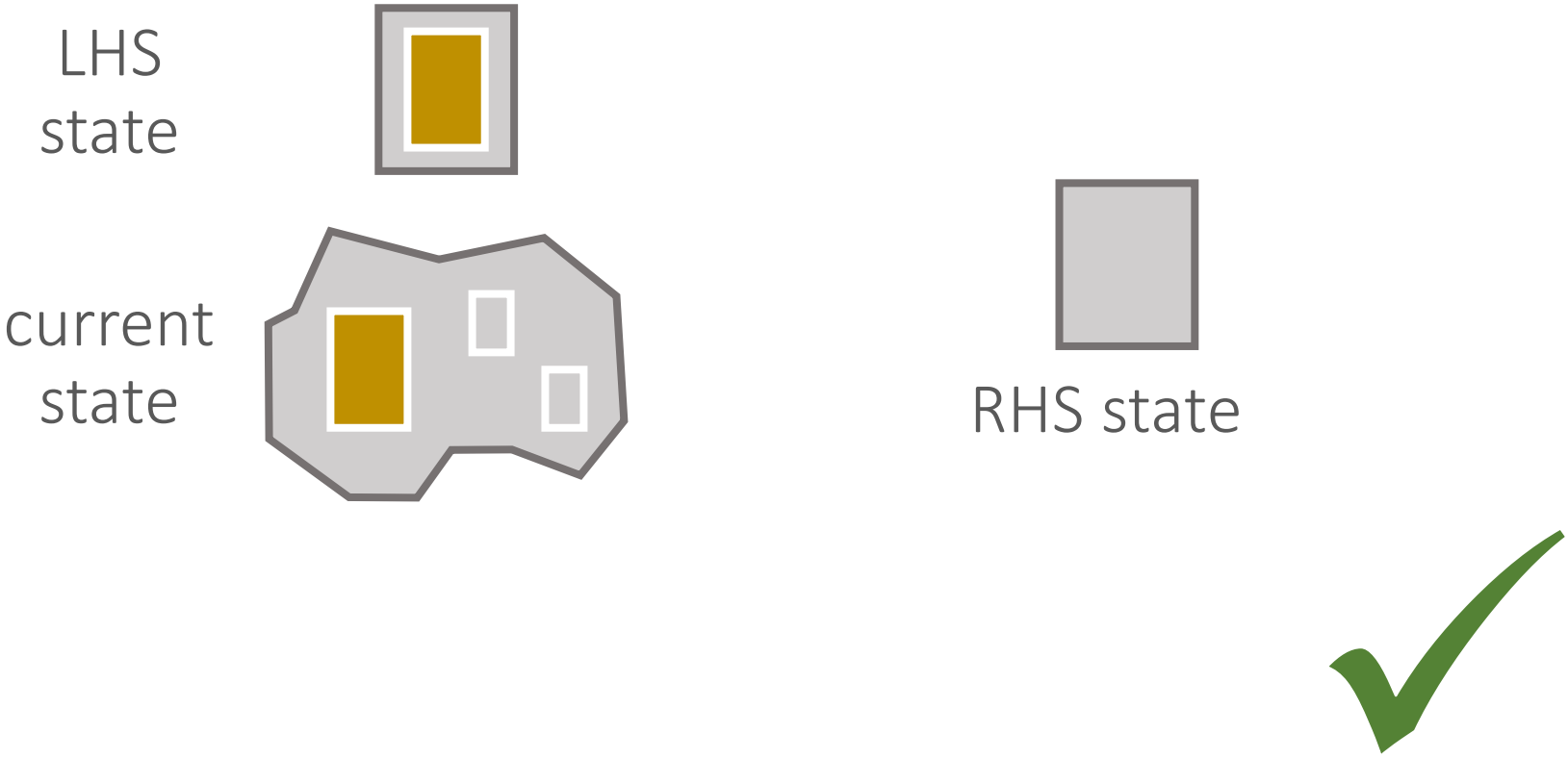
$$\texttt{package}\ \ A \twoheadrightarrow (B_1 * B_2 * \ldots * B_n)$$

3. iterate over B_i's: If B_i is a logical property P,
e.g. `x.f == 0`, then check P



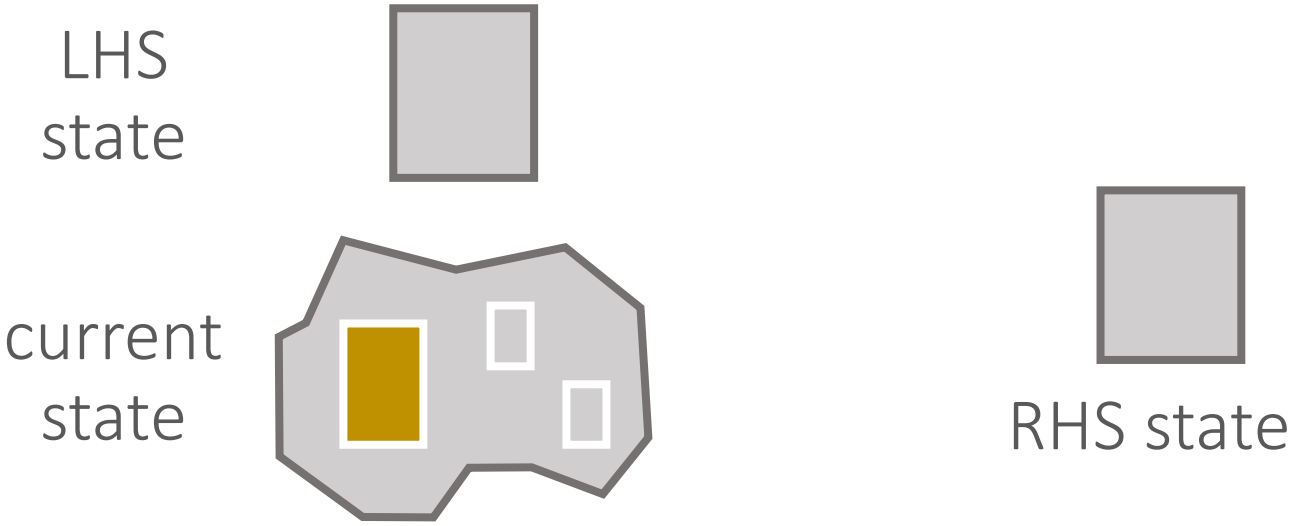
Examples

```
package acc(x.f) --* acc(x.f)
```



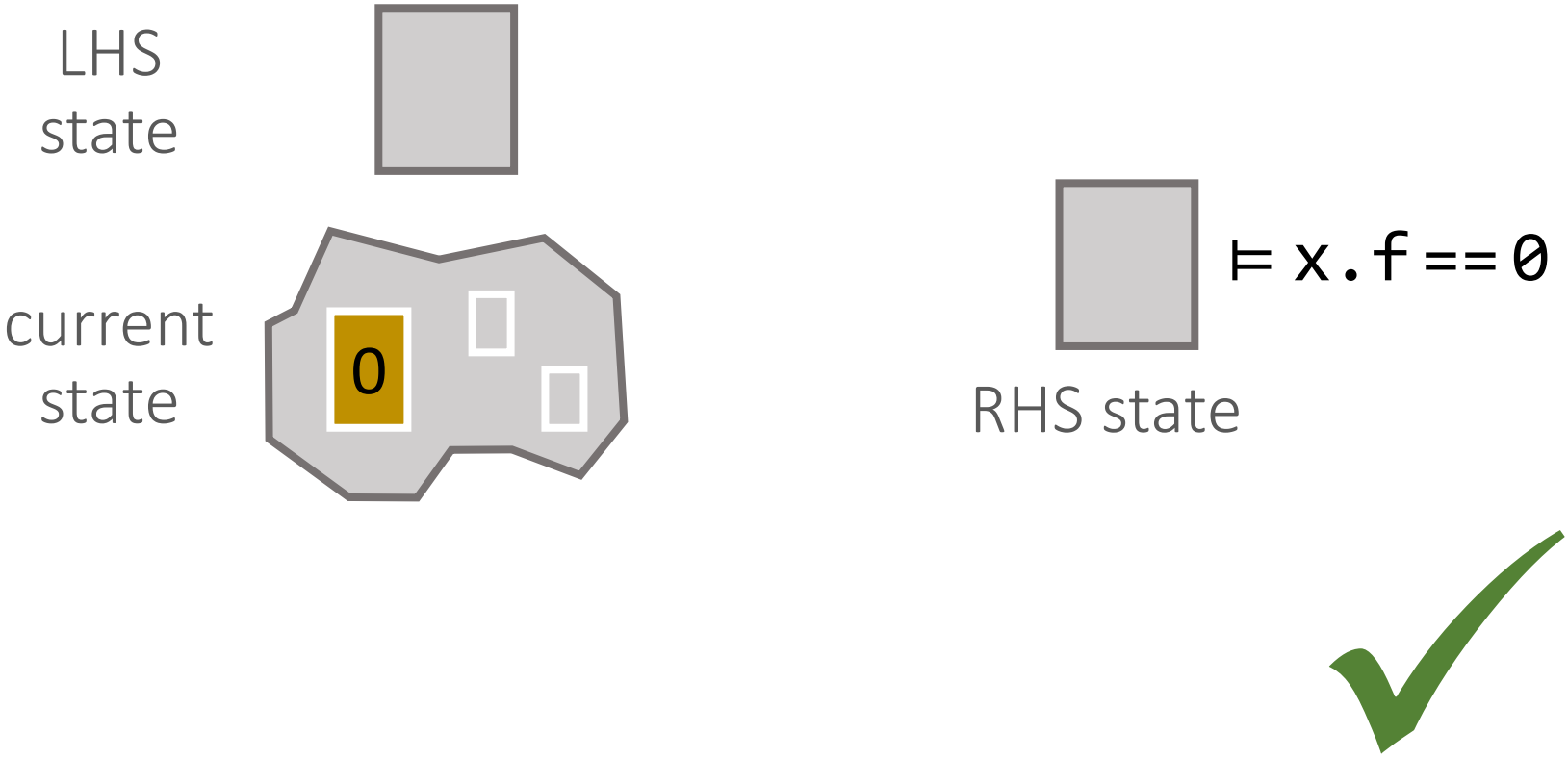
Examples

```
package true —∗ acc(x.f)
```



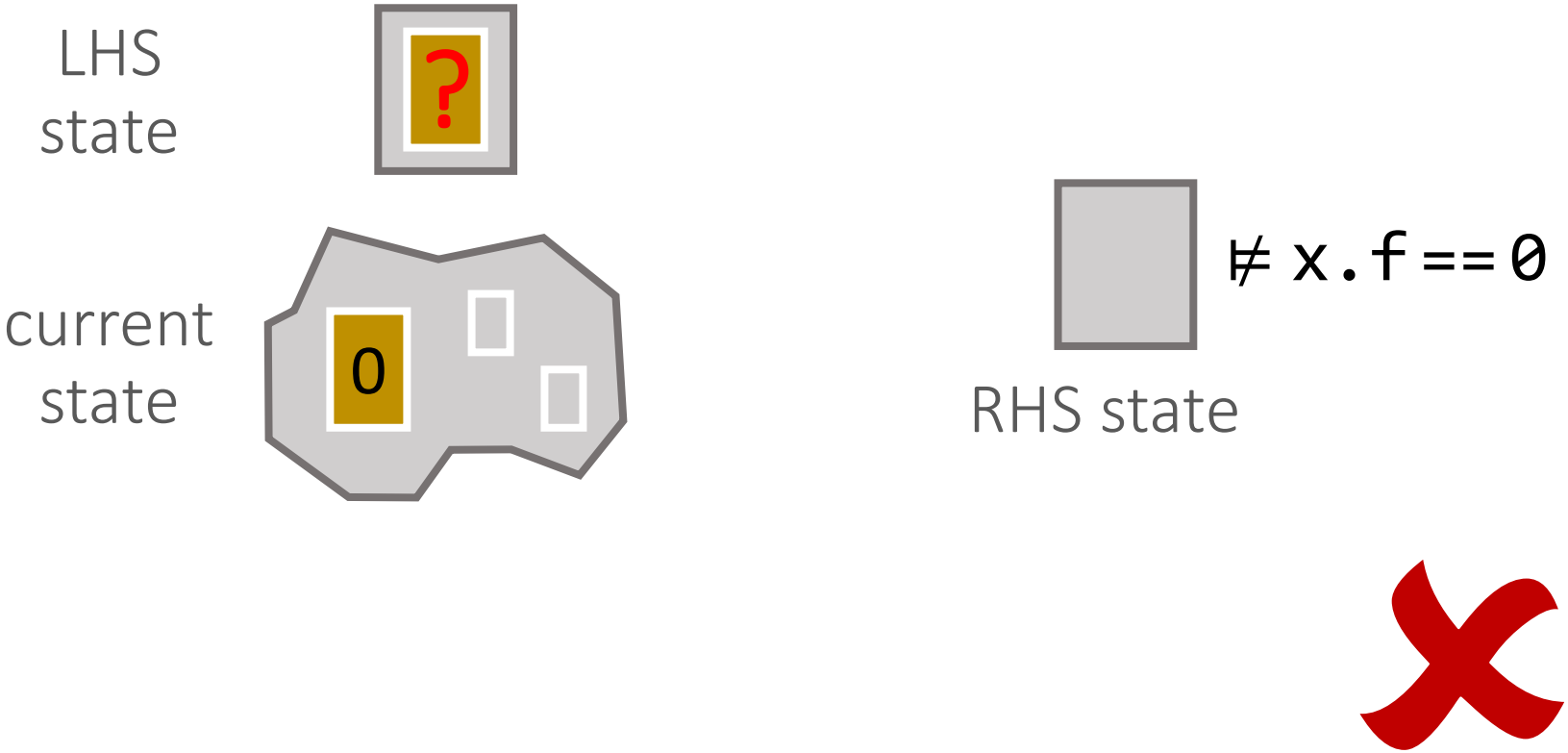
Examples

```
package true —∗ acc(x.f) ∗ x.f == 0
```



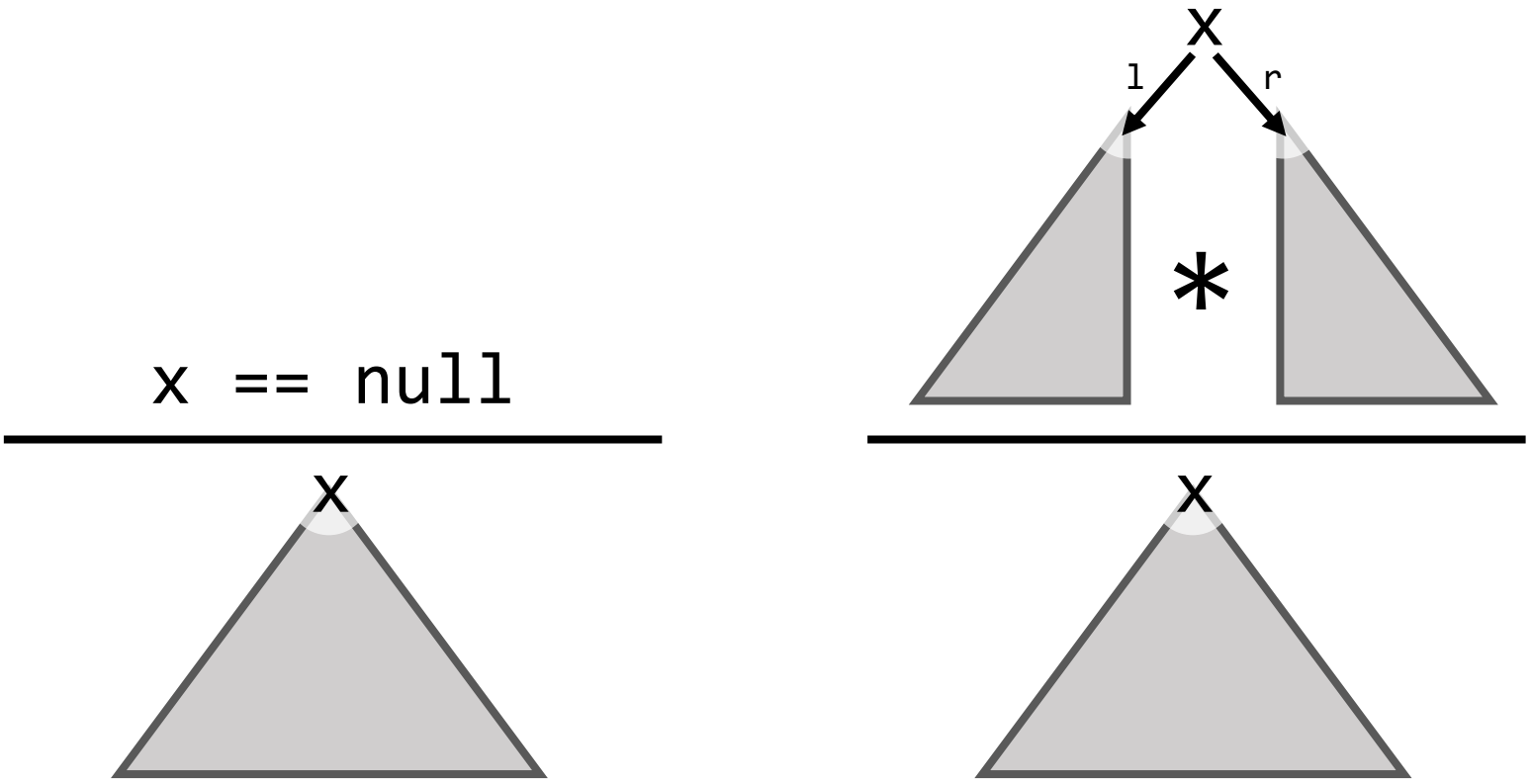
Examples

```
package acc(x.f) —∗ acc(x.f) ∗ x.f == 0
```



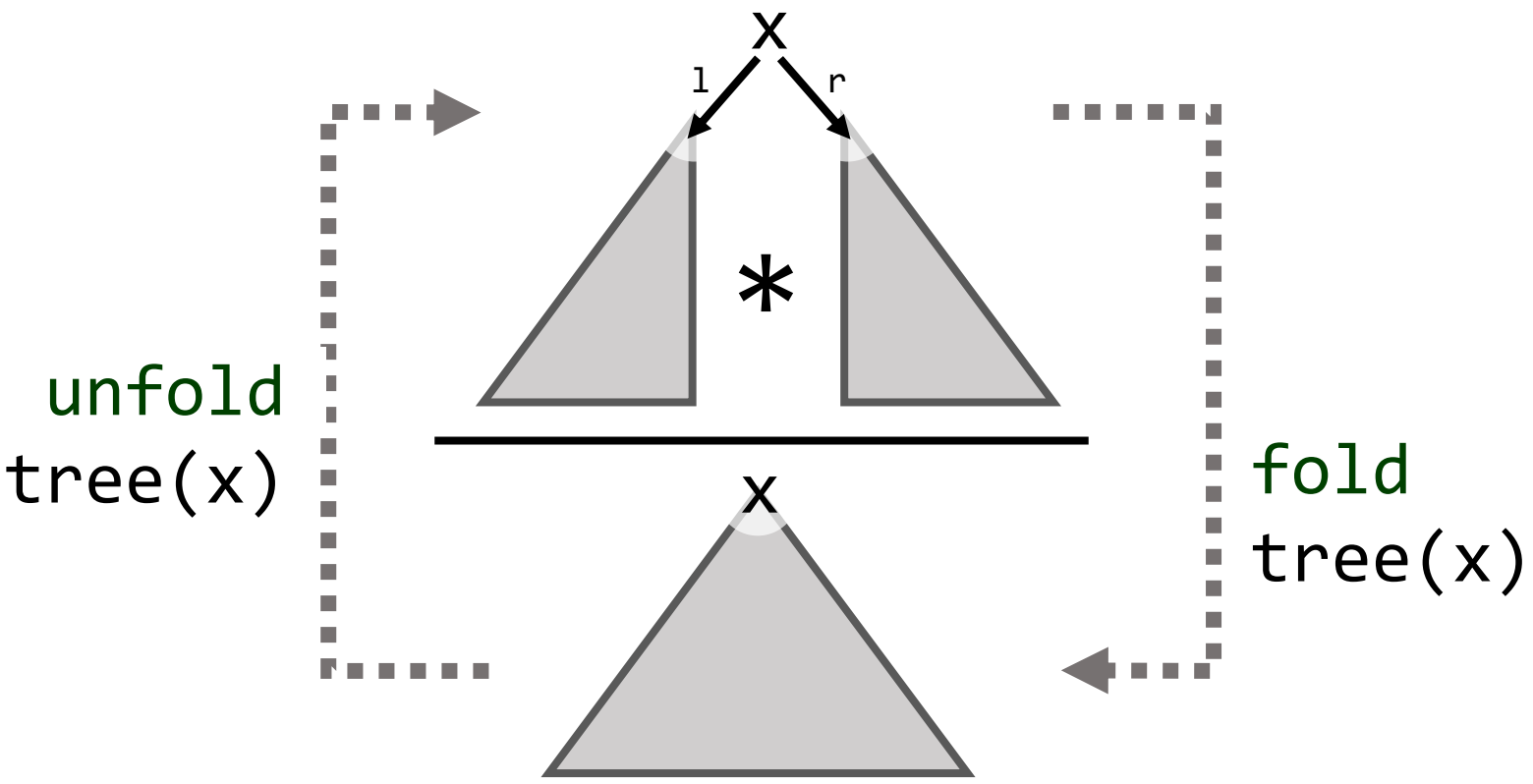
Existing Features

Abstract predicates for recursive data structures

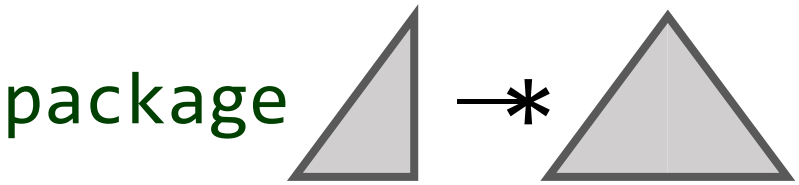


Existing Ghost Operations

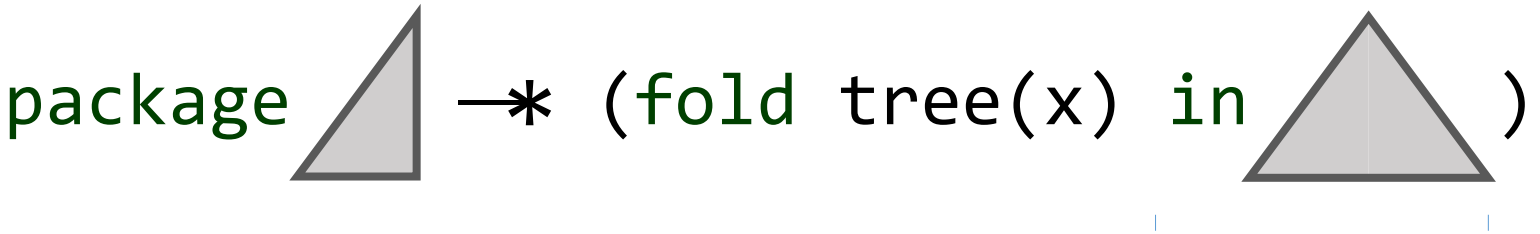
Abstract predicates plus ghost operations



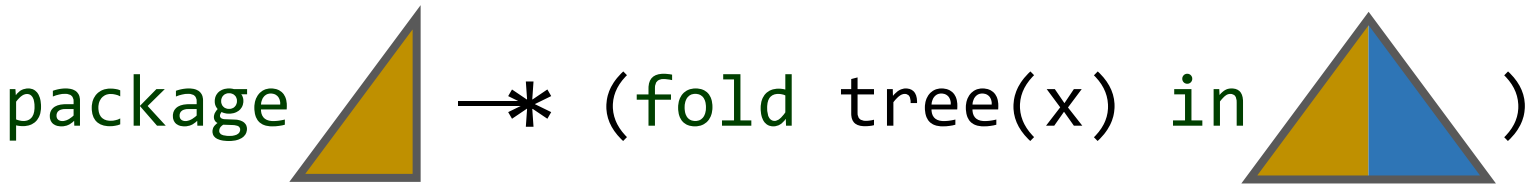
Integrating Existing Ghost Operations

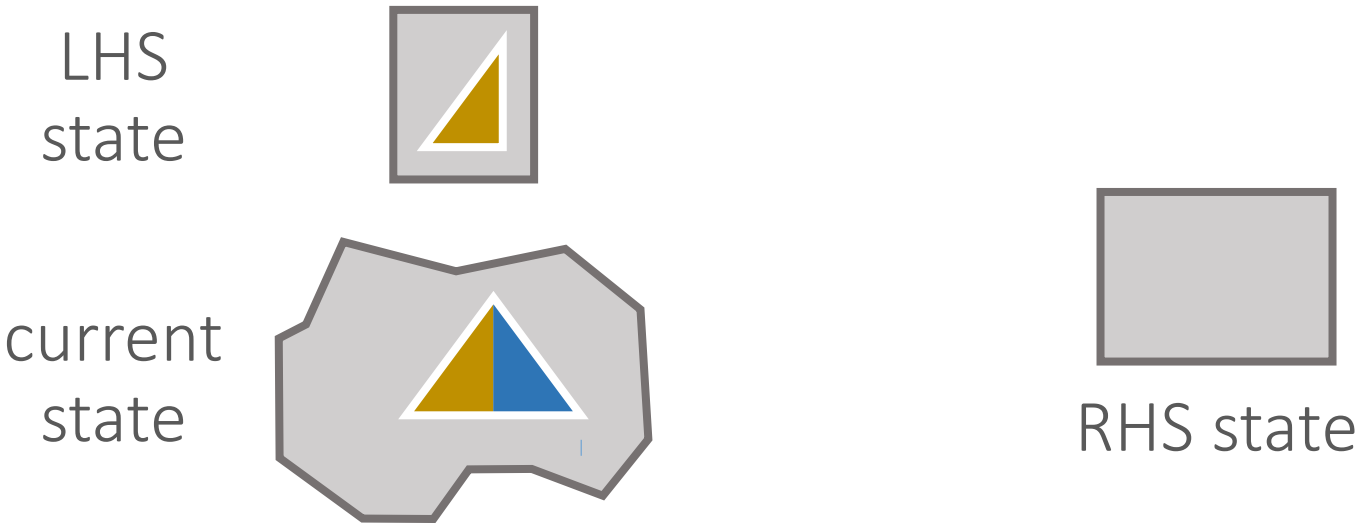


Integrating Existing Ghost Operations

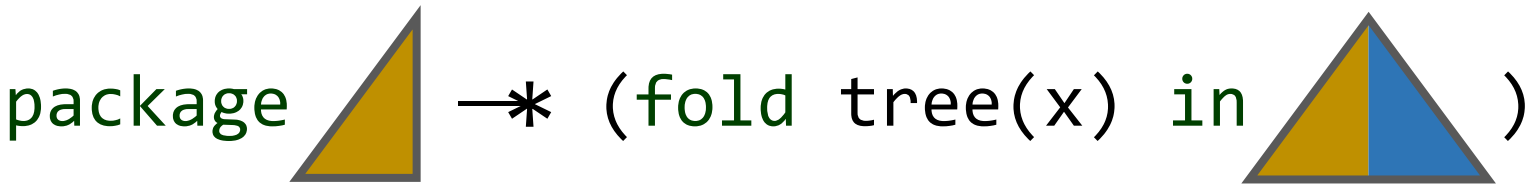


Integrating Existing Ghost Operations





Integrating Existing Ghost Operations



LHS state

current state

RHS state

Implementation

Part of Viper verification infrastructure

- Implementation based on symbolic execution
- Rich logic: unrestricted abstract predicates, abstraction functions, quantifiers, sets, sequences, custom mathematical domains, flexible permission model, …

Set of interesting examples; 1.6 to 3 seconds

Verification challenge from VerifyThis'12

- Verifies in 3s
- VerCors:
  - 6 minutes (originally, using Chalice/Boogie)
  - 60 seconds (currently, using Viper)

Annotation Inference Heuristics

Simple heuristics to infer package and apply statements

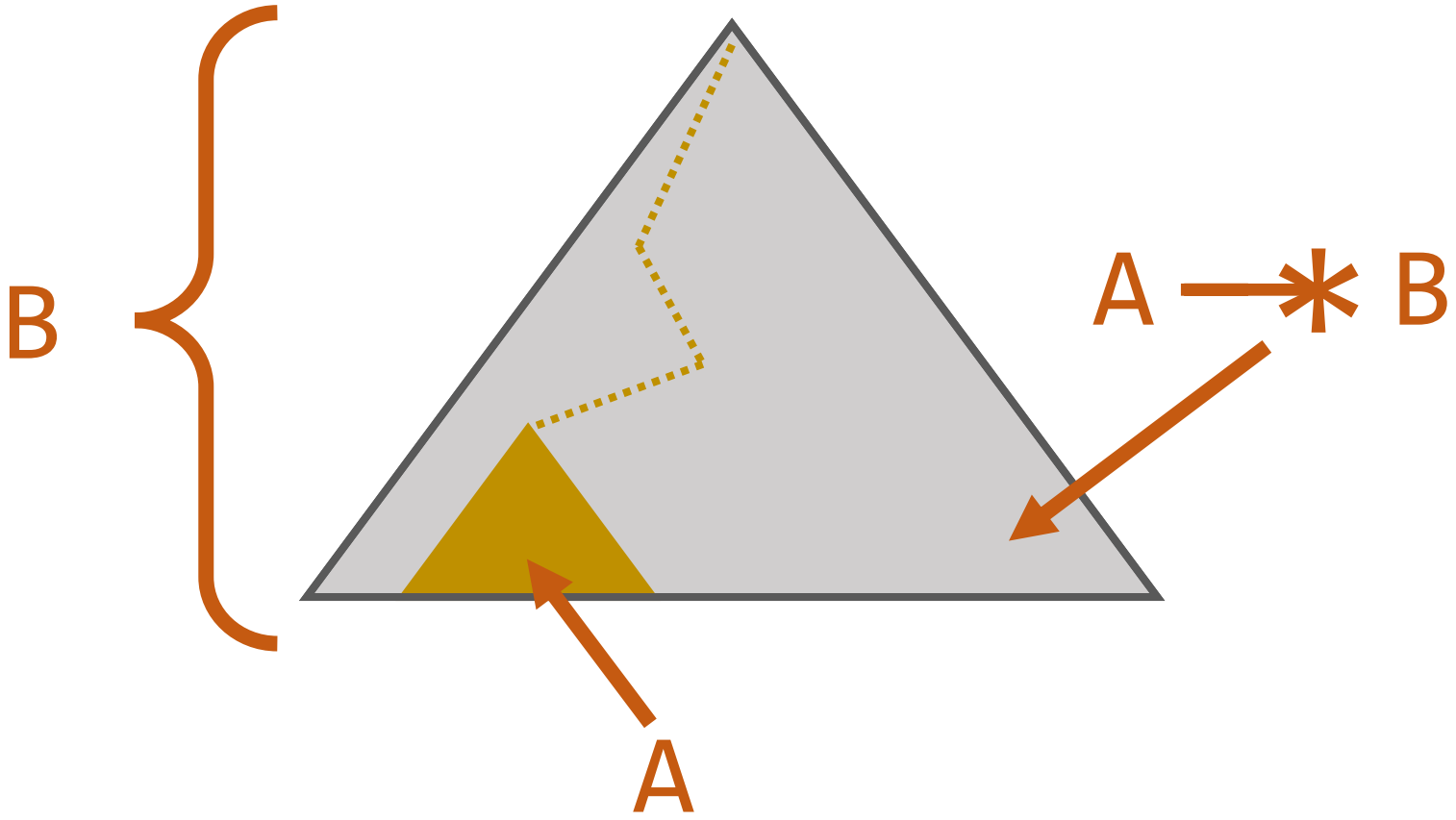
Infers all package and apply statements in our examples

Verification time: +0.5s or less

VerifyThis'12 Challenge Revisited

Scenario: Iteratively traverse a recursively defined tree

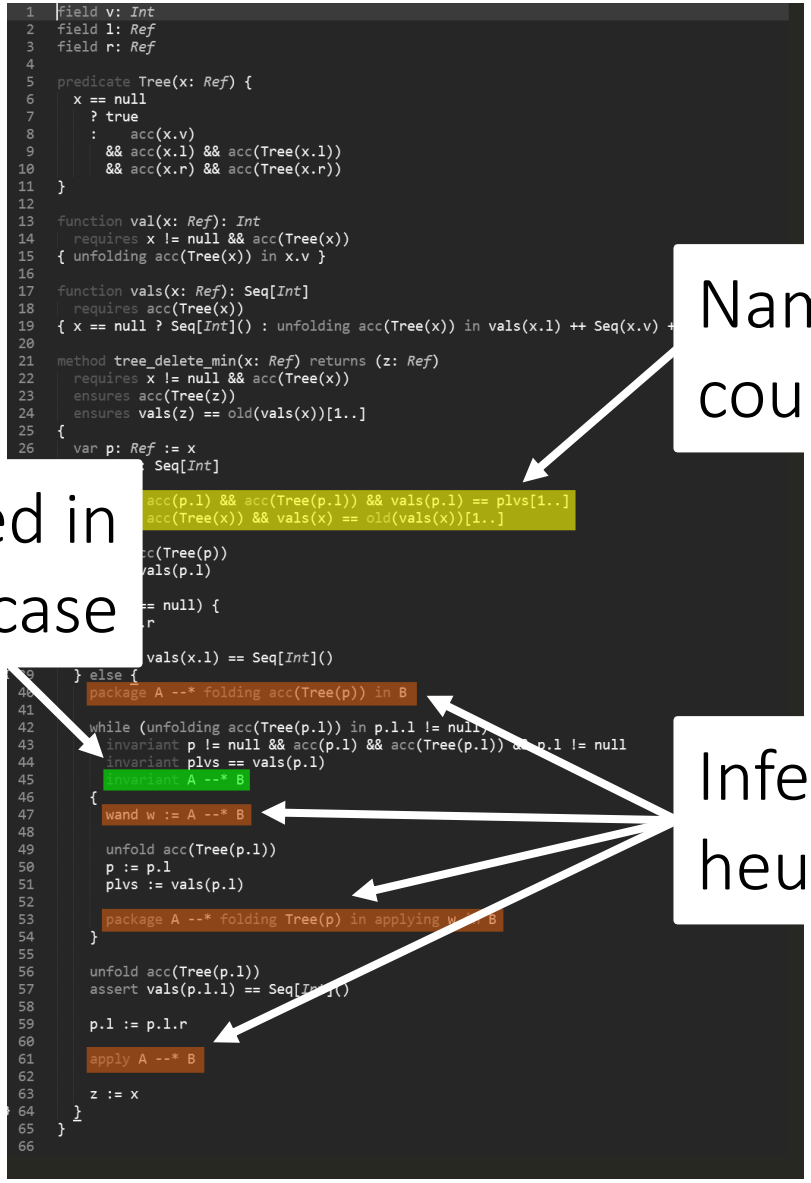
↳Loop invariant: Describe partial data structure



VerifyThis'12 Challenge Encoded

```
field v: Int
field l: Ref
field r: Ref

predicate Tree(x: Ref) {
  x == null
    ? true
    :     acc(x.v)
      && acc(x.l) && acc(Tree(x.l))
      && acc(x.r) && acc(Tree(x.r))
}

function val(x: Ref): Int
  requires x != null && acc(Tree(x))
{ unfolding acc(Tree(x)) in x.v }

function vals(x: Ref): Seq[Int]
  requires acc(Tree(x))
{ x == null ? Seq[Int]() : unfolding acc(Tree(x)) in vals(x.l) ++ Seq(x.v) ++ vals(x.r) }

method tree_delete_min(x: Ref) returns (z: Ref)
  requires x != null && acc(Tree(x))
  ensures acc(Tree(z))
  ensures vals(z) == old(vals(x))[1..]
{
  var p: Ref := x
  var plvs: Seq[Int]

  define A acc(p.l) && acc(Tree(p.l)) && vals(p.l) == plvs[1..]
  define B acc(Tree(x)) && vals(x) == old(vals(x))[1..]

  unfold acc(Tree(p))
  plvs := vals(p.l)

  if (p.l == null) {
    z := p.r

    assert vals(x.l) == Seq[Int]()
  } else {
    package A --* folding acc(Tree(p)) in B

    while (unfolding acc(Tree(p.l)) in p.l.l != null)
      invariant p != null && acc(p.l) && acc(Tree(p.l)) && p.l != null
      invariant plvs == vals(p.l)
      invariant A --* B
    {
      wand w := A --* B

      unfold acc(Tree(p.l))
      p := p.l
      plvs := vals(p.l)

      package A --* folding Tree(p) in applying w in B
    }

    unfold acc(Tree(p.l))
    assert vals(p.l.l) == Seq[Int]()

    p.l := p.l.r

    apply A --* B

    z := x
  }
}
```

VerifyThis'12 Challenge Encoded



Named shorthand, could be inlined

Required in either case

Inferred by heuristics

www.pm.inf.ethz.ch/research/viper.html

Algorithm for computing wand footprints

- Sound (proof sketch)
- Permissive and predictable

Formalised verifier-independently

Implementation

- Co-first* to support magic wands in an automatic verifier
- Lightweight user annotations
- Convincing initial results (expressiveness, performance)

* VerCors