Viper

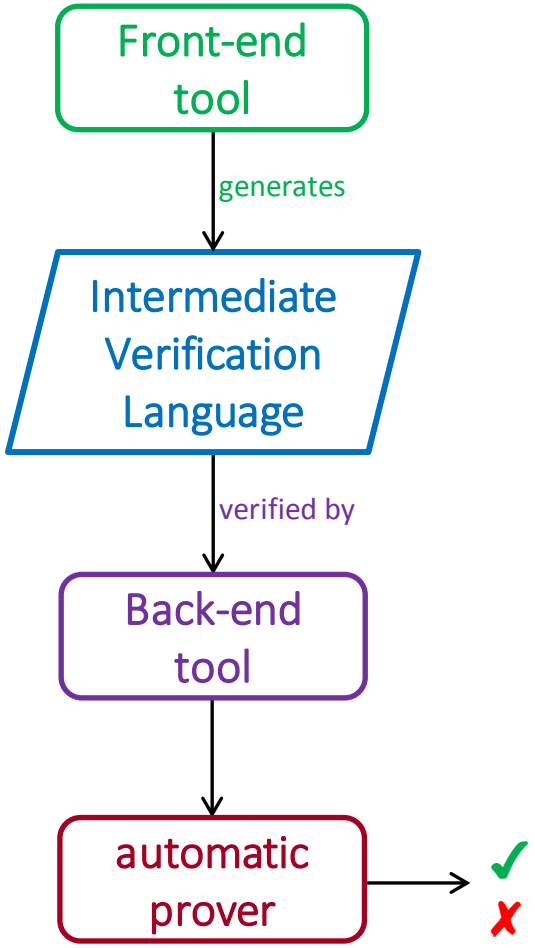
A Verification Infrastructure for Permission-Based Reasoning

Alex Summers, ETH Zurich

Joint work with Uri Juhasz, Ioannis Kassios,
Peter Müller, Milos Novacek, Malte Schwerhoff
(and many students)

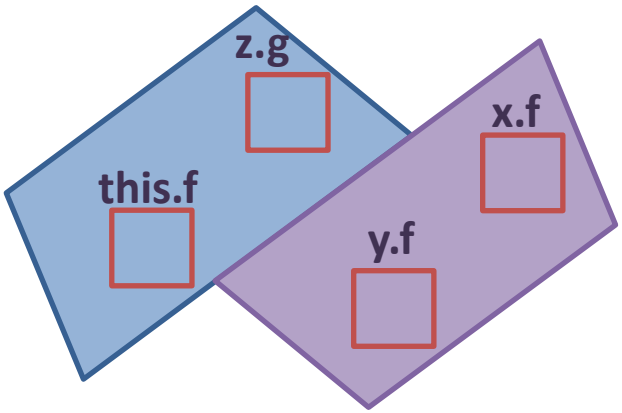
24th September 2015, Shonan Village Centre

Verification via Automatic Provers



- Last 10 years: rapid progress in automatic tools for first-order logics (SMT solvers, provers)
- Intermediate Verification Languages: e.g. Boogie and Why
- Provide common infrastructures for building program verifiers
- Many success stories and tools
  - Microsoft Hypervisor (VCC)
  - Device-drivers (Corral)
  - .. and many more, e.g., Why3, GPUVerify, Spec#, Dafny, Vericool, Krakatoa, etc….

Permission-Based Reasoning



- Modern program logics for heap reasoning + concurrency
  - e.g. separation logics
- control of ownership/sharing of *partial heaps* (heap fragments)
- First-order prover technology *difficult* to directly exploit
  - Custom verification engines (usually symbolic execution)
  - Lots of work to implement
  - Hard to reuse for new work
  - … fewer tools available ☹

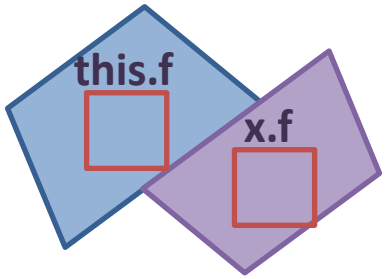
Framing with Permissions

```
void m(x:C)
requires
ensures
{

  this.f := 2;

  call increment(x);

  assert this.f == 2;

}
```

Framing with Permissions

- Tackle the problem of *framing* by splitting the heap into *partial heaps*
- Method call works on its own partial heap
  - *defined by precondition*

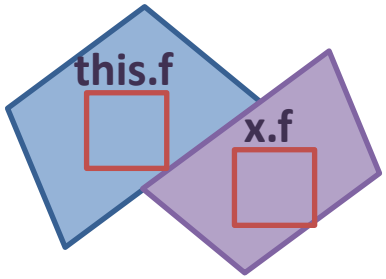
```
void m(x:C)
requires
ensures
{
  this.f := 2;

  call increment(x);

  assert this.f == 2;

}
```



Framing with Permissions

- Tackle the problem of *framing* by splitting the heap into *partial heaps*
- Method call works on its own partial heap
  - *defined by precondition*
- At each statement
  - split current heap into:
    - part needed by statement
    - left-over part (the *frame*)

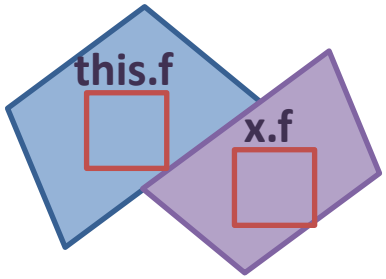
```
void m(x:C)
requires
ensures
{
    this.f := 2;

    call increment(x);

    assert this.f == 2;

}
```



Framing with Permissions

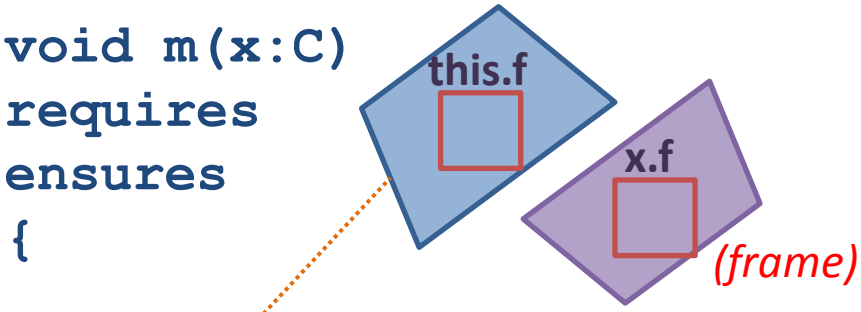
- Tackle the problem of *framing* by splitting the heap into *partial heaps*
- Method call works on its own partial heap
  - *defined by precondition*
- At each statement
  - split current heap into:
    - part needed by statement
    - left-over part (the *frame*)

```
void m(x:C)
requires
ensures
{
  this.f := 2;

  call increment(x);

  assert this.f == 2;

}
```



Framing with Permissions

- Tackle the problem of *framing* by splitting the heap into *partial heaps*
- Method call works on its own partial heap
  - *defined by precondition*
- At each statement
  - split current heap into:
    - part needed by statement
    - left-over part (the *frame*)

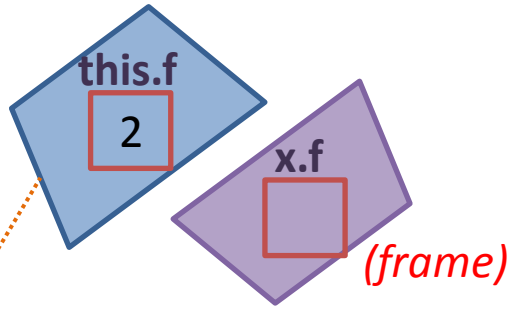



```
void m(x:C)
requires
ensures
{
  this.f := 2;

  call increment(x);

  assert this.f == 2;

}
```

Framing with Permissions

- Tackle the problem of *framing* by splitting the heap into *partial heaps*
- Method call works on its own partial heap
  - *defined by precondition*
- At each statement
  - split current heap into:
    - part needed by statement
    - left-over part (the *frame*)

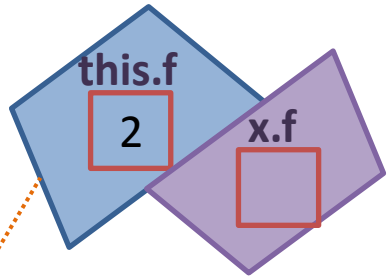
```
void m(x:C)
requires
ensures
{
  this.f := 2;

  call increment(x);

  assert this.f == 2;

}
```



Framing with Permissions

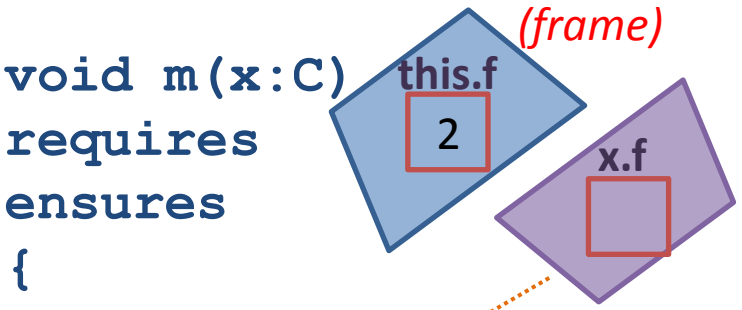
- Tackle the problem of *framing* by splitting the heap into *partial heaps*
- Method call works on its own partial heap
  - *defined by precondition*
- At each statement
  - split current heap into:
    - part needed by statement
    - left-over part (the *frame*)

```
void m(x:C)
requires
ensures
{
   this.f := 2;

   call increment(x);

   assert this.f == 2;

}
```



Framing with Permissions

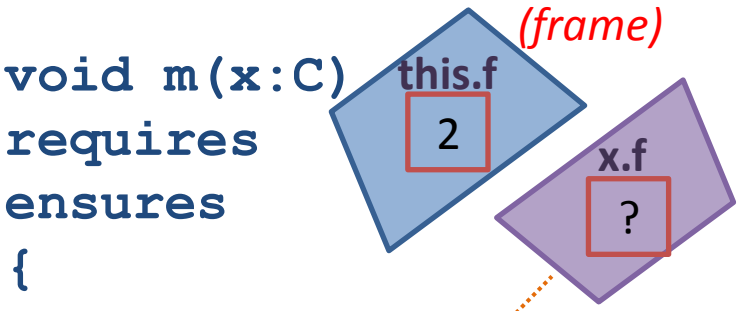
- Tackle the problem of *framing* by splitting the heap into *partial heaps*
- Method call works on its own partial heap
  - *defined by precondition*
- At each statement
  - split current heap into:
    - part needed by statement
    - left-over part (the *frame*)




```
void m(x:C)
requires
ensures
{
  this.f := 2;

  call increment(x);

  assert this.f == 2;

}
```

Framing with Permissions

- Tackle the problem of *framing* by splitting the heap into *partial heaps*
- Method call works on its own partial heap
  - *defined by precondition*
- At each statement
  - split current heap into:
    - part needed by statement
    - left-over part (the *frame*)

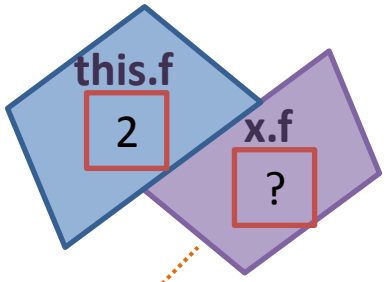



```
void m(x:C)
requires
ensures
{
  this.f := 2;

  call increment(x);

  assert this.f == 2;

}
```

Framing with Permissions

- Tackle the problem of *framing* by splitting the heap into *partial heaps*
- Method call works on its own partial heap
  - *defined by precondition*
- At each statement
  - split current heap into:
    - part needed by statement
    - left-over part (the *frame*)

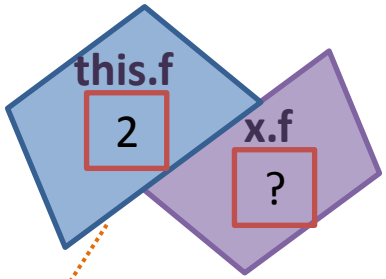
```
void m(x:C)
requires
ensures
{
  this.f := 2;

  call increment(x);

  assert this.f == 2;

}
```



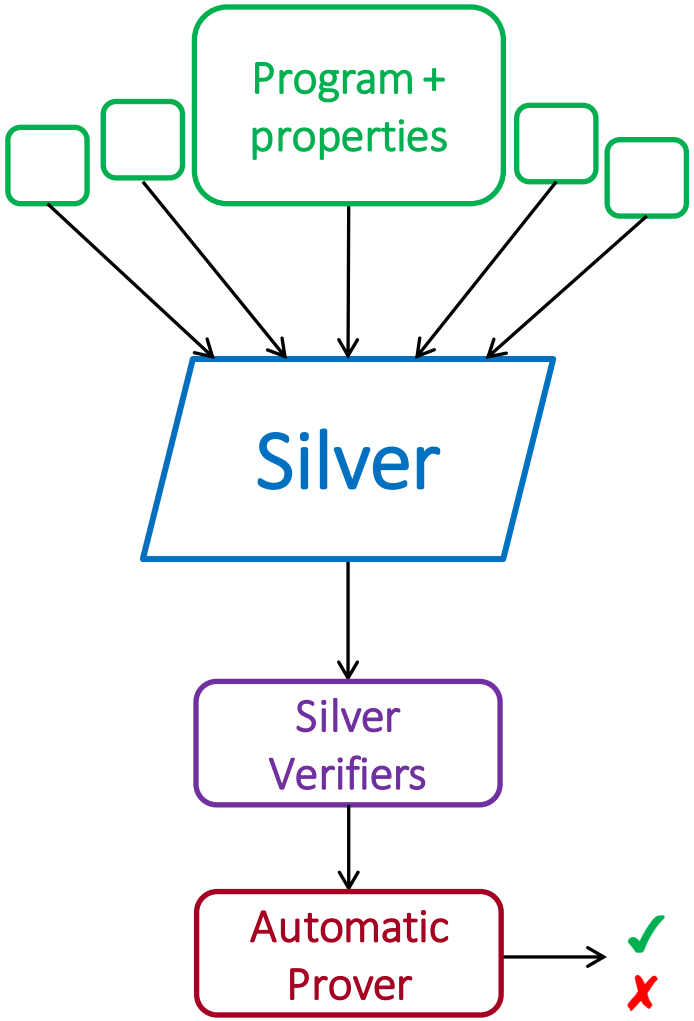
Framing with Permissions

- Tackle the problem of *framing* by splitting the heap into *partial heaps*
- Method call works on its own partial heap
  - *defined by precondition*
- At each statement
  - split current heap into:
    - part needed by statement
    - left-over part (the *frame*)

```
void m(x:C)
requires
ensures
{
   this.f := 2;

   call increment(x);

   assert this.f == 2;

}
```



The Viper Project



- We have designed **Silver**: a new *intermediate verification language*
- Idea: front-end tools that *translate* a problem into a **Silver** program
- We provide (two) **Silver Verifiers**
  - decide problems automatically
- The tool infrastructure is called Viper
  - in use for several projects (later)
- Native support for *permissions*
- Easy to encode many methodologies
  - separation logic, dynamic frames, invariants, type systems, etc…

Silver: Basic Assertion Language

- Based on *Implicit Dynamic Frames* [Smans et al. '09]
- Permission assertions: *accessibility predicates* acc(e.f)
  - exclusive: similar to $e.f \mapsto \_$ in separation logics
- Expressions e may depend directly on the heap
  - e.g. acc(x.f) && x.f > 0
- *Fractional permissions* [Boyland'03], e.g. acc(x.f, ½)
  - allow reading (and framing), not writing
- Conjunction && is *multiplicative* for permissions
  - e.g. acc(x.f, ½) && acc(x.f, ½) ≡ acc(x.f, 1)
  - e.g. acc(this.f, 1) && acc(x.f, 1) ⊨ this ≠ x

Silver: a tiny example

```
field f: Int

method increment(c: Ref)
  requires acc(c.f)
  ensures  acc(c.f) && c.f == old(c.f) + 1
 {
   c.f := c.f + 1
 }
```

`old(e)` evaluates *e* in the pre-heap of the method call

Silver: a tiny example

```
field f: Int

method increment(c: Ref)
  requires acc(c.f)
  ensures  acc(c.f) && c.f == old(c.f) + 1
 {
   c.f := c.f + 1
 }

method m(this: Ref, x:Ref)
  requires acc(this.f) && acc(x.f)
  ensures acc(this.f) && acc(x.f) && this.f == 2
 {
   this.f := 2;

   increment(x);


   assert this.f == 2
 }
```

`old(e)` evaluates *e* in the pre-heap of the method call

Silver primitives: Inhale and Exhale

- A statement **`inhale A`** means:
  - all permissions required by are **`A`** gained
  - all logical constraints (e.g. x.f > 0) are *assumed*
- A statement **`exhale A`** means:
  - check, and remove all permissions required by **`A`**
  - all logical constraints (e.g. x.f > 0) are *asserted*
  - any locations to which all permissions is lost are implicitly *havoced* (their values are no-longer known)
- Can be seen as the *permission-aware analogues* of assume/assert statements used in first-order verification
  - used to model appropriate permission transfers
  - verification semantics for high-level constructs
    - e.g. method call: **`exhale pre; inhale post`**

Silver: a tiny example

```
field f: Int

method increment(c: Ref)
  requires acc(c.f)
  ensures  acc(c.f) && c.f == old(c.f) + 1
 {
   c.f := c.f + 1
 }

method m(this: Ref, x:Ref)
  requires acc(this.f) && acc(x.f)
  ensures acc(this.f) && acc(x.f) && this.f == 2
 {
   this.f := 2;


   increment(x);


   assert this.f == 2
 }
```

Silver: a tiny example

```
field f: Int

method increment(c: Ref)
  requires acc(c.f)
  ensures  acc(c.f) && c.f == old(c.f) + 1
 {
   c.f := c.f + 1
 }

method m(this: Ref, x:Ref)
  requires acc(this.f) && acc(x.f)
  ensures acc(this.f) && acc(x.f) && this.f == 2
 {
   this.f := 2;

   var old_f : Int := x.f
   exhale acc(x.f)
   inhale acc(x.f) && x.f == old_f + 1

   assert this.f == 2
 }
```

Example : Encoding Locks

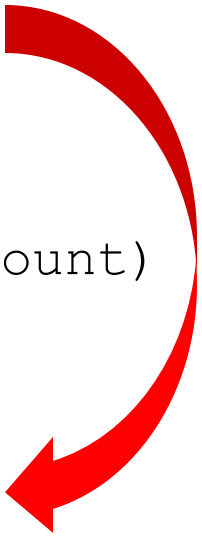
```
class C {
  int[ ] data; int count = 0;

  monitor invariant acc(this.data) && acc(this.count)


  void Foo( ) {
    acquire this;
    int i = data.length;
    while( 0 < i )
      invariant acc(this.data) && acc(this.count)
      invariant holds( this );
    { ... ; i = i - 1; }
    count = count + 1; release this;
  }
}
```

A few powerful Viper features

- *Paired assertions* **`[A,B]`**
  - **`A`** used when inhaled, **`B`** used when exhaled
  - mismatches: external justification / proof obligations
- *Quantification over local state* **`forallrefs[f] x ::`**
  - non-standard for separation logics (but handy)

Example : Two-state invariants

```
class C {
  int[ ] data; int count = 0;

  monitor invariant acc(this.data) && acc(this.count)
                    && this.count > old(this.count)

  void Foo( ) {
    acquire this;
    int i = data.length;
    while( 0 < i )
      invariant acc(this.data) && acc(this.count)
      invariant holds( this );
    { ... ; i = i - 1; }
    count = count + 1; release this;
  }
}
```



A few powerful Viper features

- *Paired assertions* **`[A,B]`**
  - **`A`** used when inhaled, **`B`** used when exhaled
  - mismatches: external justification / proof obligations
- *Quantification over local state* **`forallrefs[f] x ::`**
  - non-standard for separation logics
- *State snapshots, labelled "old" expressions*

Recursive assertions: predicates

```
predicate tree(t: Ref) {
  t != null ==> acc(t.val) && acc(t.left) && acc(tree(t.left))
                && acc(t.right) && acc(tree(t.right))
}
```

- Permission to unbounded recursive data structures
  - predicate definitions can take any number of parameters
- Predicate instances (e.g. tree(t)) are treated as a generalisation of permissions (inhaled/exhaled)
- Fold/unfold statements exchange predicate instances with their bodies (not automatic, due to recursion)
  - **unfold** `tree(t)` exchanges instance for body assertion
  - **fold** `tree(t)` exchanges body for predicate instance

Recursive assertions: functions

```
predicate tree(t: Ref) {
  t != null ==> acc(t.val) && acc(t.left) && acc(tree(t.left))
                && acc(t.right) && acc(tree(t.right))
}
function vals(t: Ref): Seq[Int]
  requires acc(tree(t))
{
  t == null ? Seq[Int]() : unfolding tree(t) in
                vals(t.left) ++ Seq(t.val) ++ vals(t.right)
}
```

- Silver also supports (recursive) *functions*
  - The body of a function is an *expression* (side-effect-free)
  - precondition must provide sufficient permissions to evaluate
  - Function invocations only allowed where precondition holds
    - e.g. **`tree(t) && vals(t) == Seq(1,2,3)`**
  - usable in both specifications and statements (*pure methods*)

A few powerful Viper features

- *Paired assertions* **`[A,B]`**
  - **`A`** used when inhaled, **`B`** used when exhaled
  - mismatches: external justification / proof obligations
- *Quantification over local state* **`forallrefs[f] x ::`**
  - non-standard for separation logics
- *State snapshots, labelled "old" expressions*
- *Custom predicates, heap-dependent functions* [ECOOP'13]
  - fold/unfold for predicates, functions mostly automatic
- *Custom domains, sets and sequences, quantifiers*
- *Constrainable permissions* [VMCAI'13, FTfJP'14]
  - Alternative to fractional permissions (angelic amounts)
- *"Magic wand" support* [ECOOP'15]
  - Powerful connective from separation logic

Magic Wands

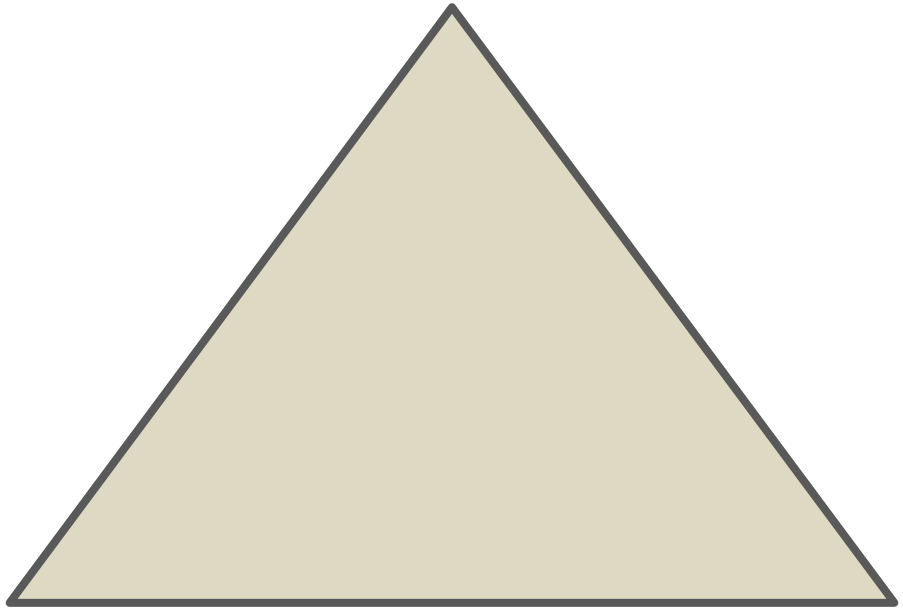
$$A \mathrel{-\!\!*} B$$

describes potential *exchange of verification states*

Read as a promise: "In any state, if you combine $A \mathrel{-\!\!*} B$ with A, then you can exchange them for B"

Tree Challenge

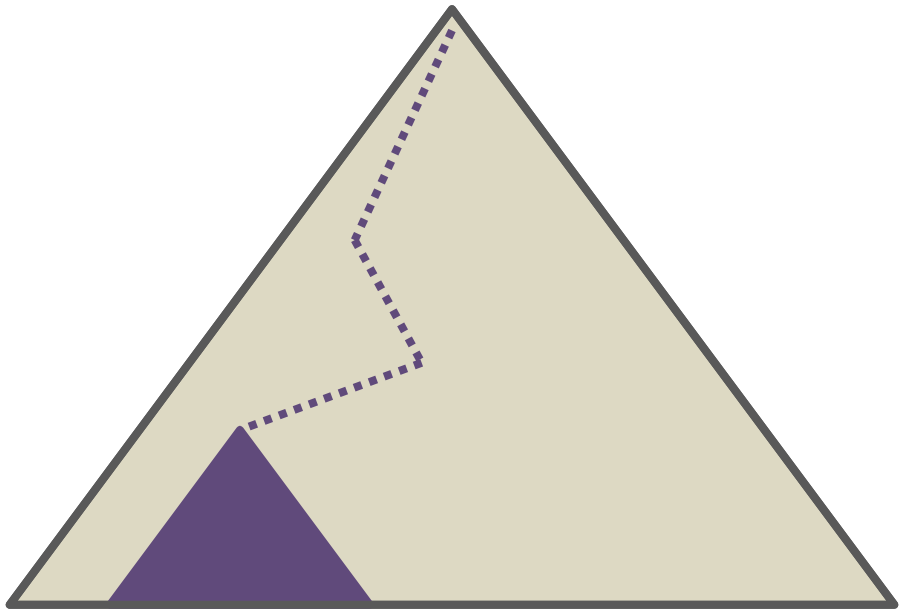
Scenario: Iteratively traverse a recursively defined tree
(Verification Challenge at VerifyThis@FM'12)



Tree Challenge

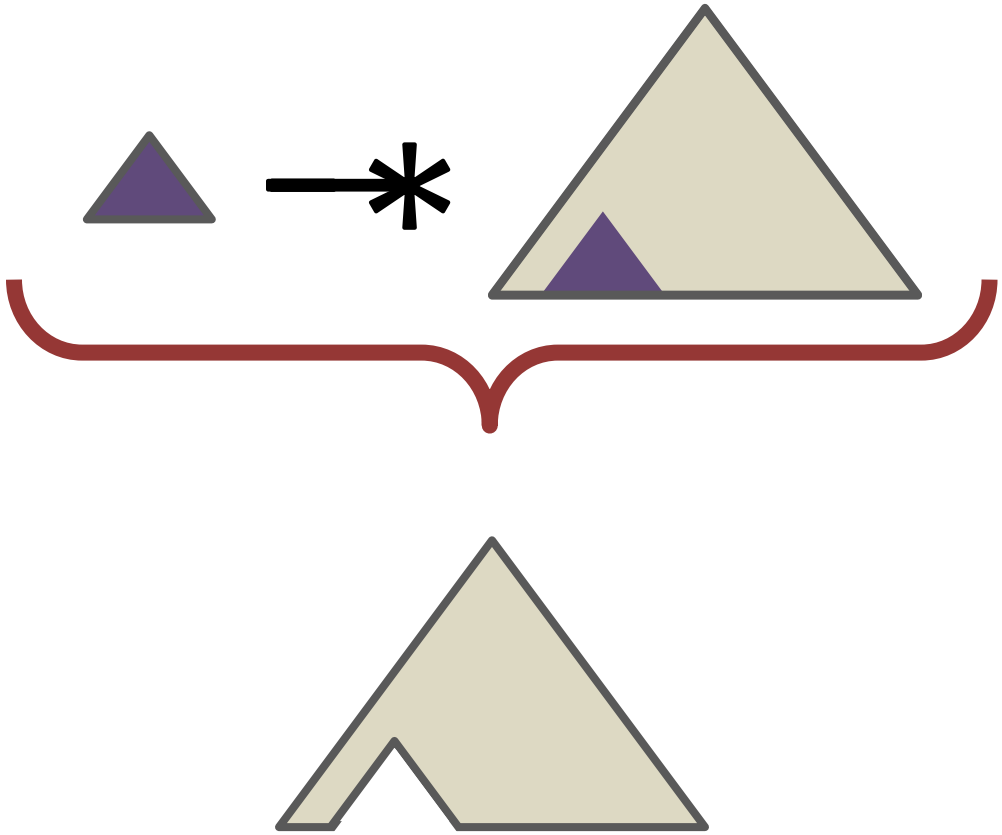
Scenario: Iteratively traverse a recursively defined tree

↳Loop invariant: Describe partial data structure



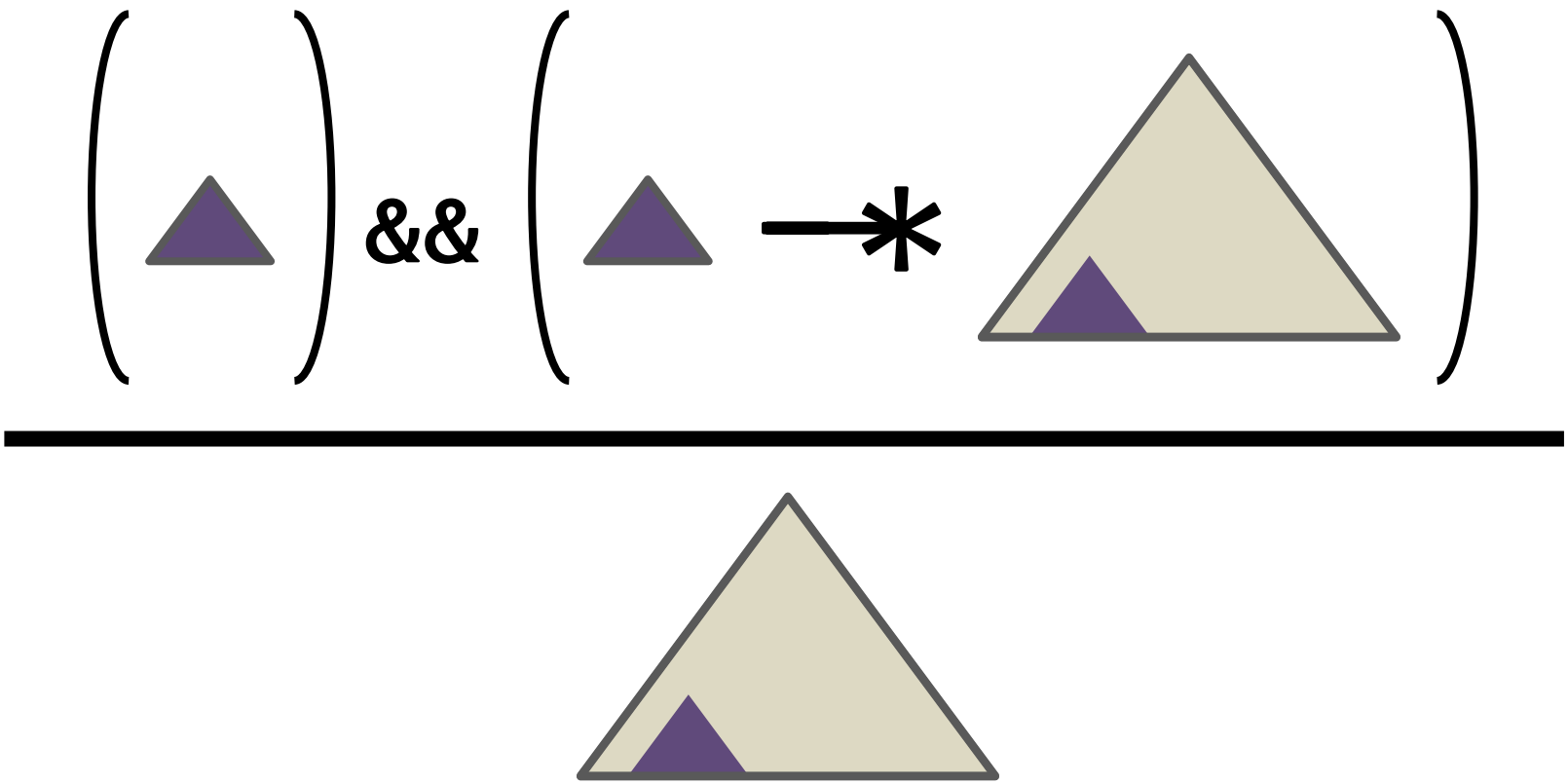
Partial Data Structures as Magic Wands

Indirectly describe partial data structure as a promise



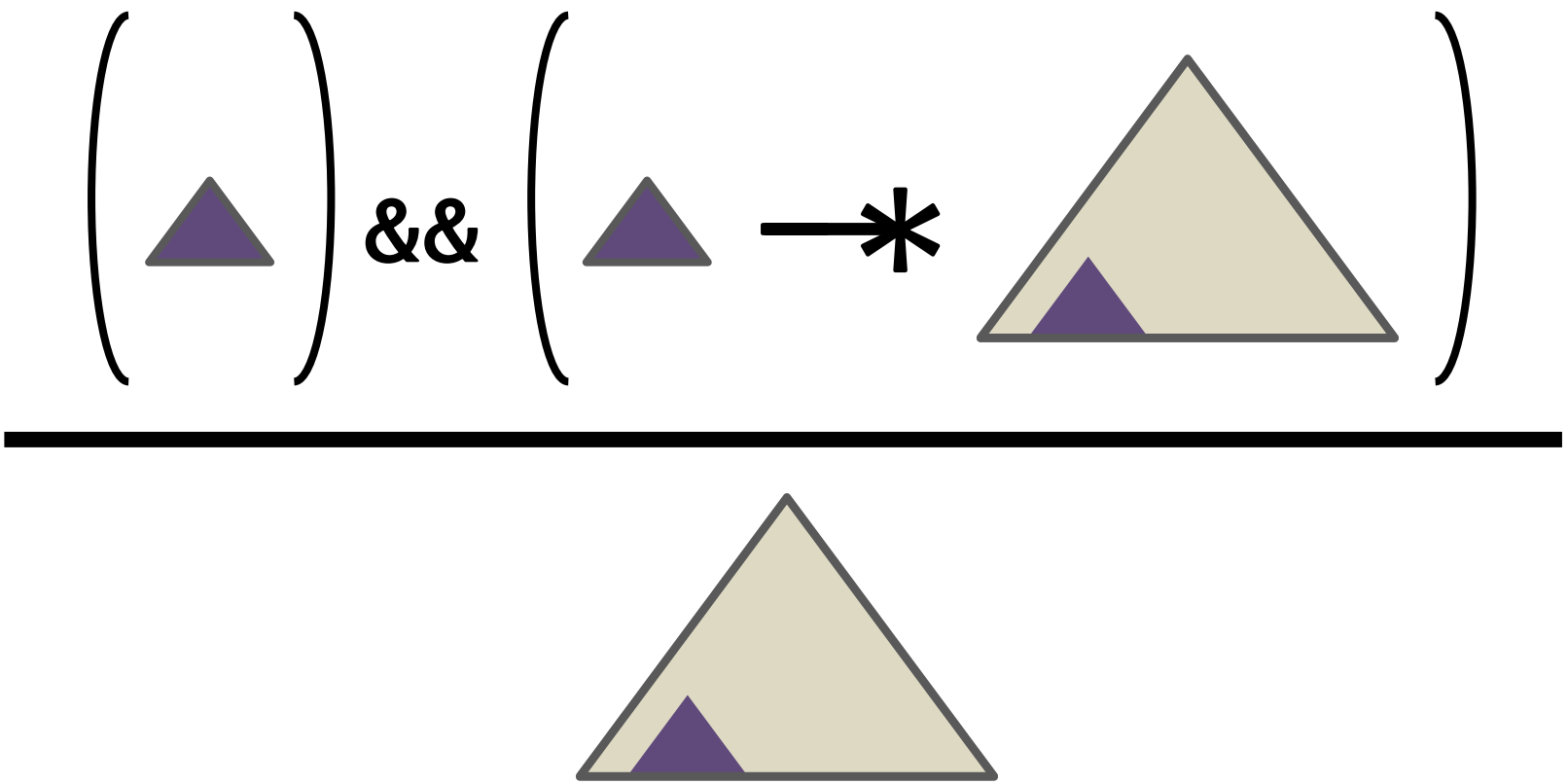
Partial Data Structures as Magic Wands

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 (separation logic)

- Partial data structures
- Usage protocols for data structures (e.g. iterators)
- Synchronisation barriers
- …

Typically* not supported in automatic verifiers

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

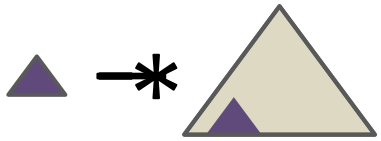
Entailment of magic wand formulas is undecidable

↳ Lightweight user guidance to direct verification

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

Guidance: Ghost Operations + Specifications

Make a promise

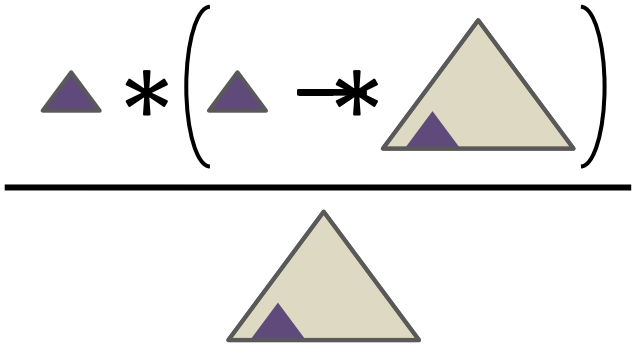


`package` $A \mathrel{-\!\!*} B$

Pass it around

Use $A \mathrel{-\!\!*} B$ in specifications

Use it

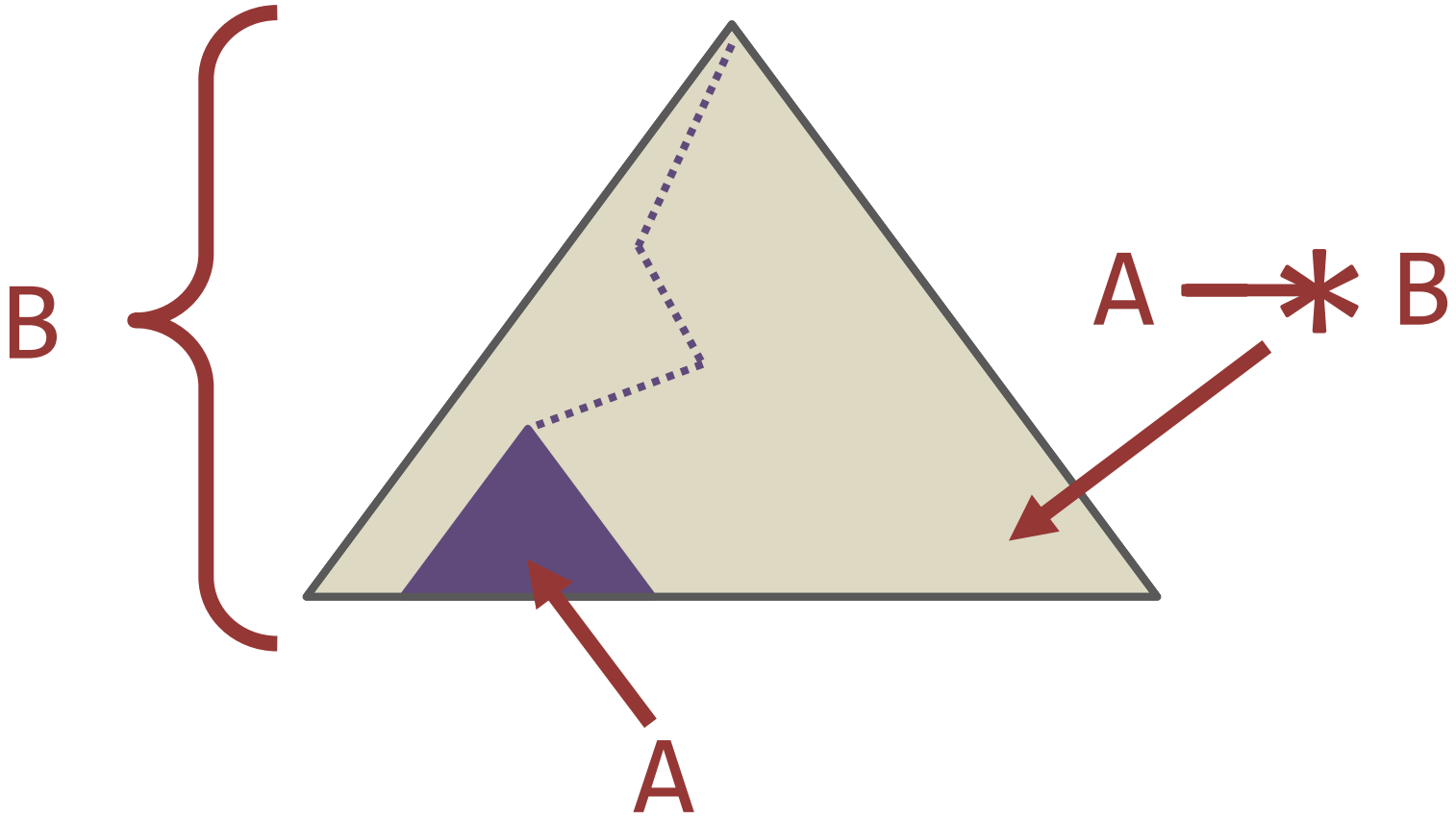


`apply` $A \mathrel{-\!\!*} B$

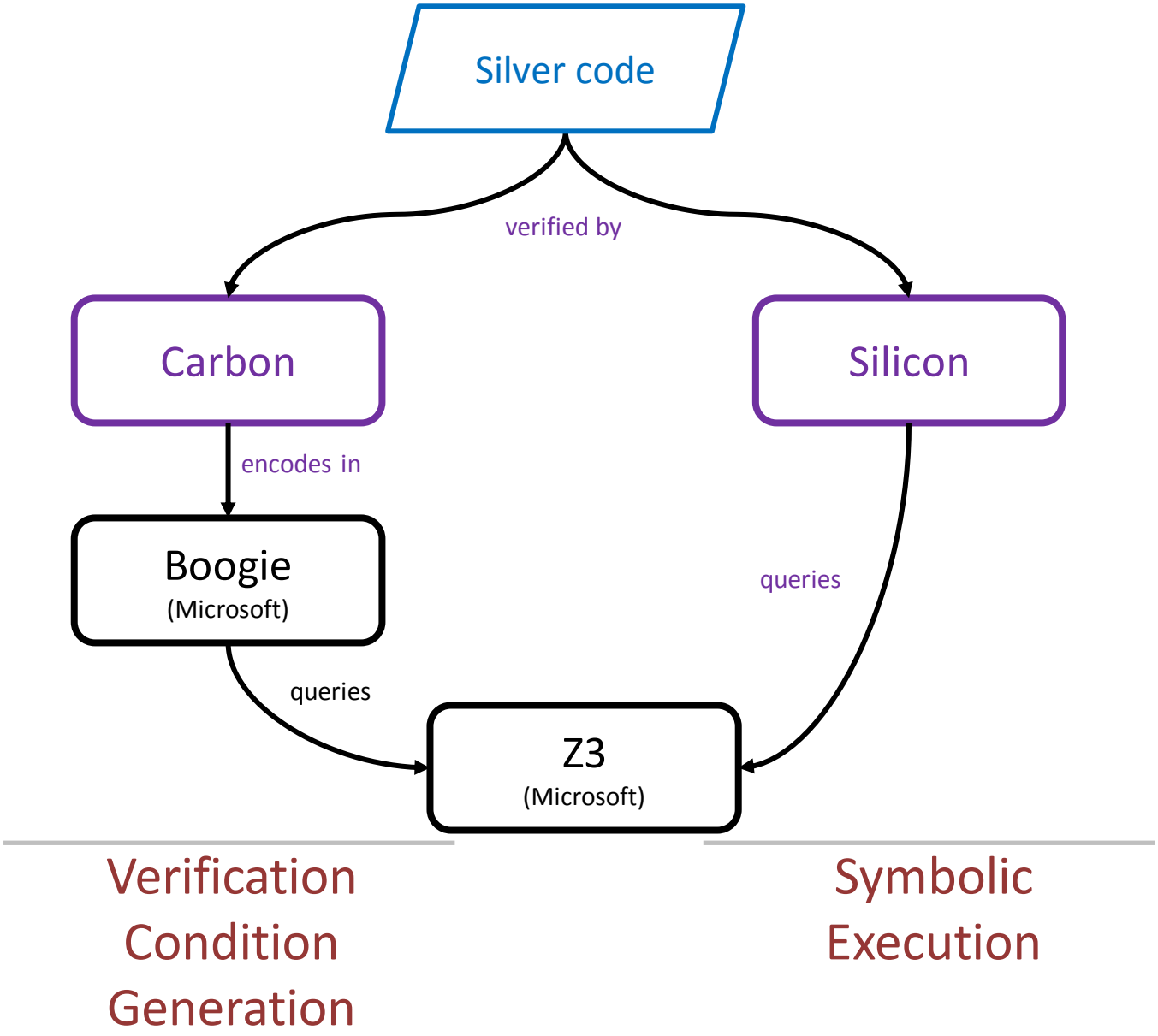
VerifyThis'12 Challenge Revisited

Scenario: Iteratively traverse a recursively defined tree

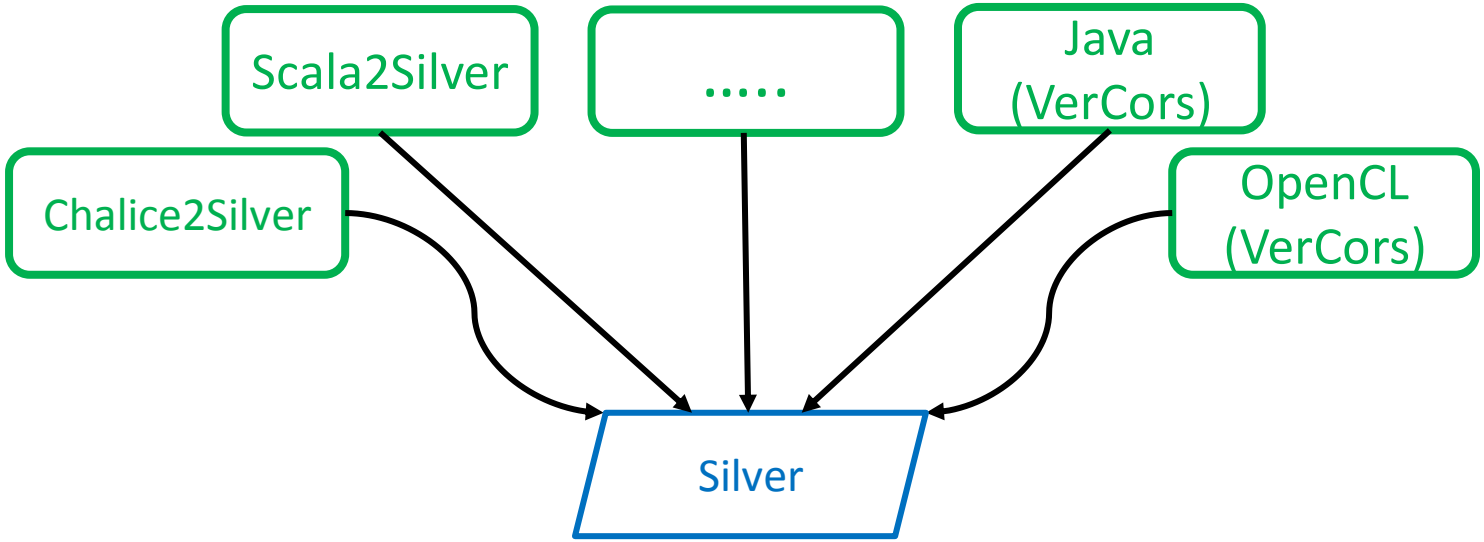
↳Loop invariant: Describe partial data structure



Verification of Silver Code (back-ends)



Current (and Ongoing) Applications



Chalice (alternative verifier)
- concurrency verification
- [Leino&Müller'09] (later + Smans)

Scala (small fragment)

Finite blocking verification
- [Boström&Müller'15]

Automating TaDa logic (W.I.P.)
- [da Rocha Pinto et al.'14]

VerCors (ERC) project
- Marieke Huisman (UTwente)
- Java, GPU code, kernel code

Javascript verification
- Philippa Gardner (IC London)

SCION router verification
- Adrian Perrig (ETH Zurich)

[Your tool name here? ☺]

Tool Availability and Future Work

- Core tools released (open-source) in September 2014:

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

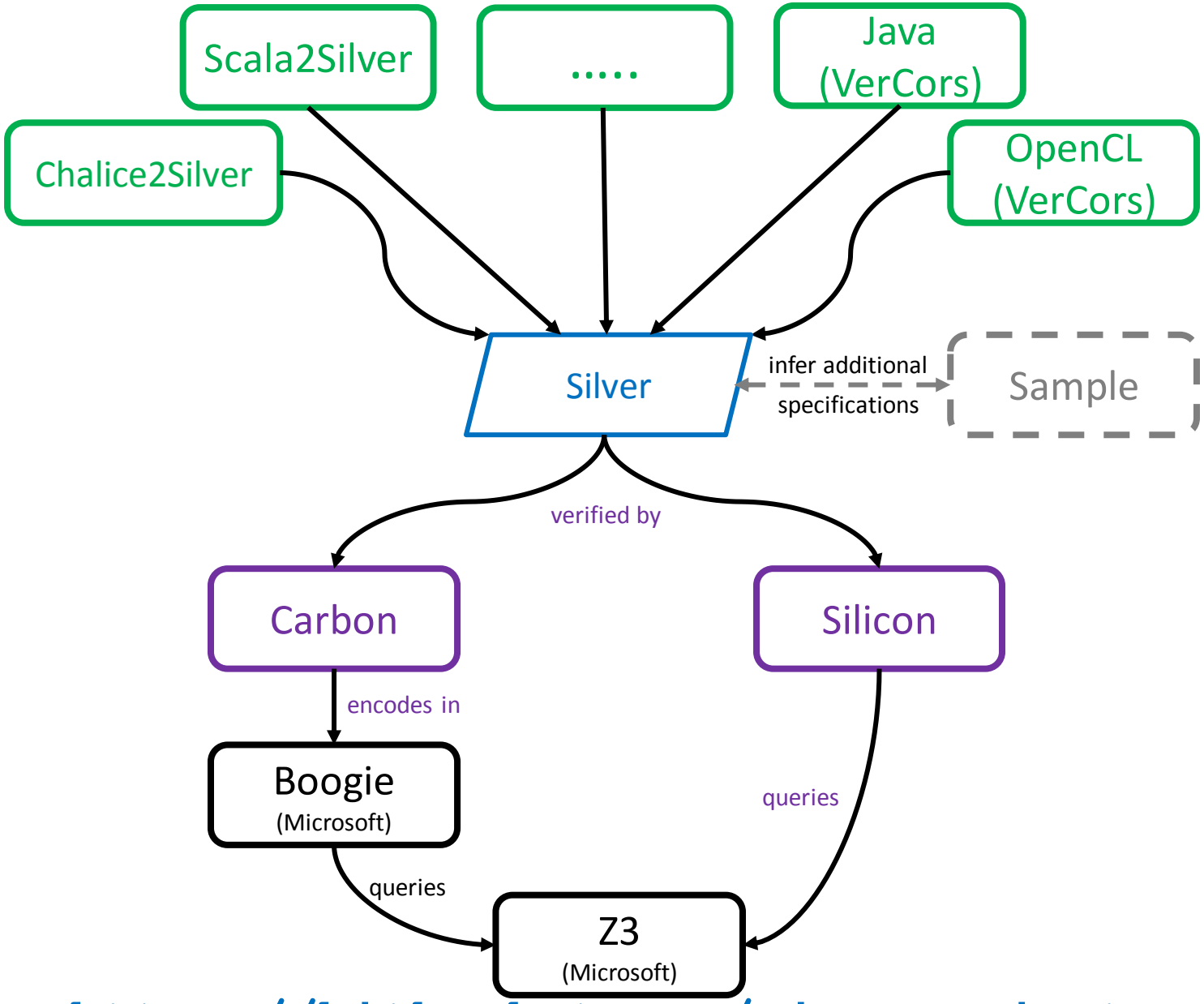
  https://bitbucket.org/viperproject

  - we have (public!) issue trackers for known problems
  - Some advanced features are in the pipeline (but ask)

- Building / supporting new tools by translations into Silver
  - SL, dynamic frames, rely-guarantee, type systems
  - More-advanced program logics? Weak memory? …
  - Also interested in work we *cannot* encode (yet …)

- A good platform to experiment with and build on ☺
  - coalesces much formal and practical past research
  - users can focus on the aspects relevant to their work

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



https://bitbucket.org/viperproject