An Introduction to Automated Program Verification with Permission Logics

Uri Juhasz, Ioannis Kassios, Peter Müller, Milos Novacek, Malte Schwerhoff, Alex Summers (and several students)

15th May 2015, Systems Group, ETH Zurich

Initial Example (Pseudo Java)

```
class Cell {
  int v

  void add(Cell c)
  { v = v + c.v }
}
```

```
void client() {
  Cell c1 = new Cell()
  c1.v = 1
  Cell c2 = new Cell()
  c2.v = 2

  c1.add(c2)

  assert c1.v == 3
  assert c2.v == 2
}
```

Goal: Check assertions **statically**

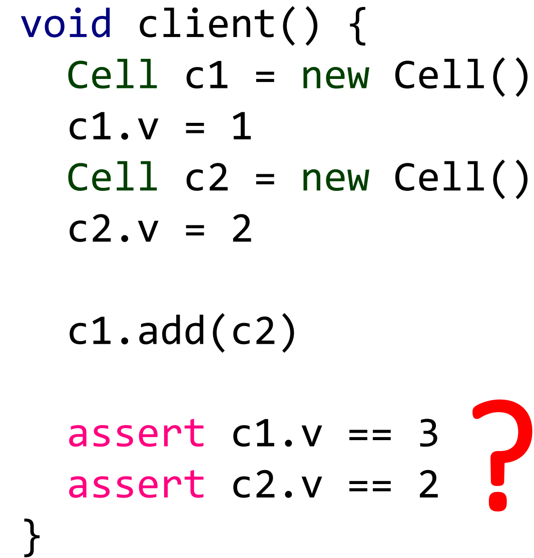
Challenges:

- Whole-code analysis is **expensive**
- **Dynamic dispatch** (inheritance; open-world assumption)

Modularity

```
class Cell {
  int v

  void add(Cell c)
  { ███ }
}
```

```
void client() {
  Cell c1 = new Cell()
  c1.v = 1
  Cell c2 = new Cell()
  c2.v = 2

  c1.add(c2)

  assert c1.v == 3
  assert c2.v == 2
}
```



Specifications

```
class Cell {
  int v

  void add(Cell c)
    requires c != null
    ensures v == old(v) + old(c.v)
  { v = v + c.v }
}
```

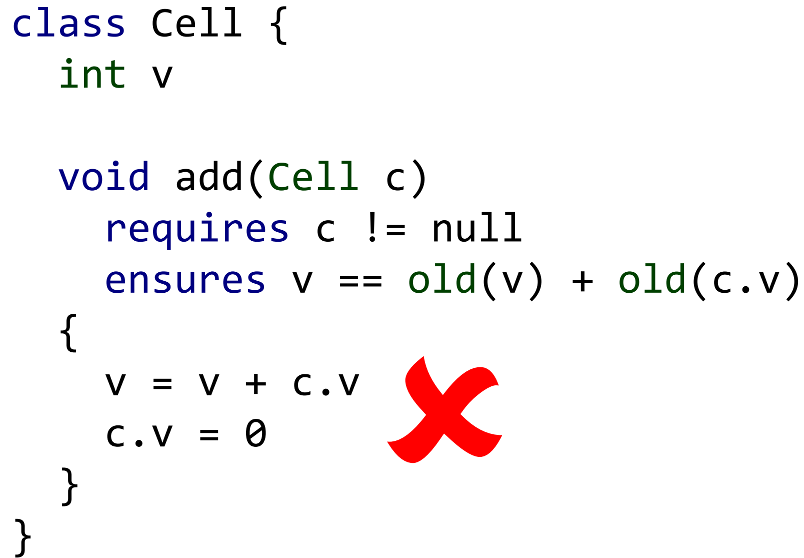
```
void client() {
  Cell c1 = new Cell()
  c1.v = 1
  Cell c2 = new Cell()
  c2.v = 2

  c1.add(c2)

  assert c1.v == 3
  assert c2.v == 2
}
```

Reasoning with Specifications

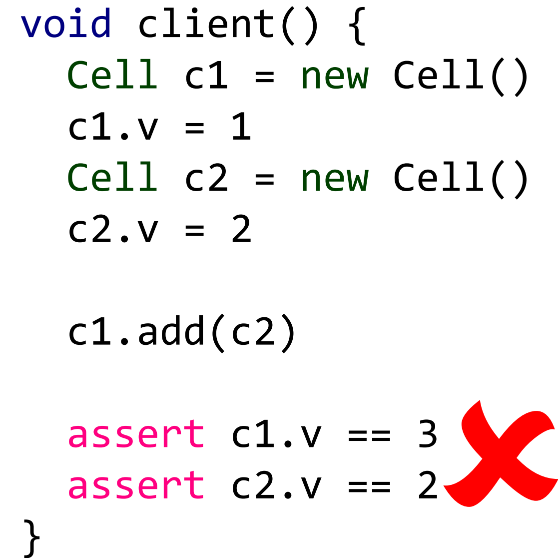
```
class Cell {
  int v

  void add(Cell c)
    requires c != null
    ensures v == old(v) + old(c.v)
  { ████████████ }
}
```

```
void client() {
  Cell c1 = new Cell()
  c1.v = 1
  Cell c2 = new Cell()
  c2.v = 2

  c1.add(c2)

  assert c1.v == 3
  assert c2.v == 2
}
```

An Incorrect Implementation

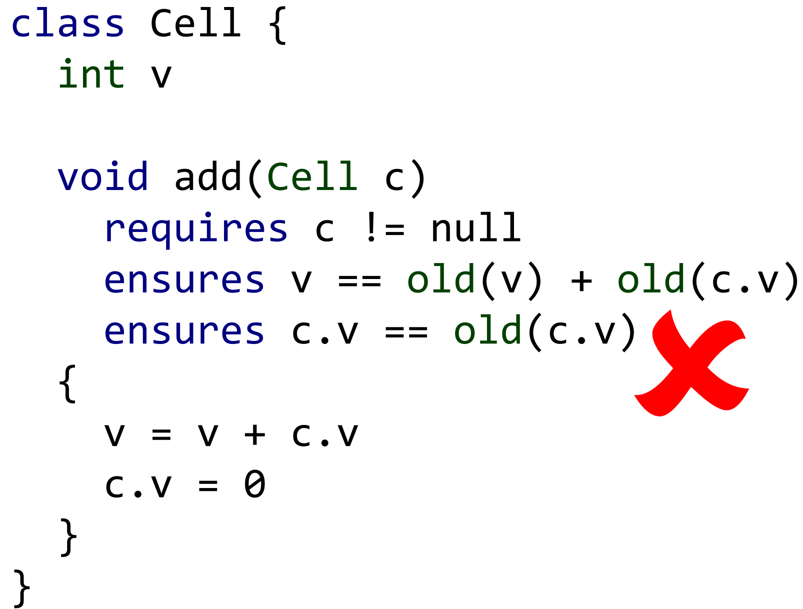


```
class Cell {
  int v

  void add(Cell c)
    requires c != null
    ensures v == old(v) + old(c.v)
  {
    v = v + c.v
    c.v = 0
  }
}
```



```
void client() {
  Cell c1 = new Cell()
  c1.v = 1
  Cell c2 = new Cell()
  c2.v = 2

  c1.add(c2)

  assert c1.v == 3
  assert c2.v == 2
}
```

Strengthening Specifications




```
void client() {
  Cell c1 = new Cell()
  c1.v = 1
  Cell c2 = new Cell()
  c2.v = 2

  c1.add(c2)

  assert c1.v == 3
  assert c2.v == 2
}
```

Strengthening Specifications

```
class Cell {
  int v

  void add(Cell c)
    requires c != null
    ensures v == old(v) + old(c.v)
    ensures c.v == old(c.v)
  { v = v + c.v }
}
```

?

```
void client() {
  Cell c1 = new Cell()
  c1.v = 1
  Cell c2 = new Cell()
  c2.v = 2

  c1.add(c2)

  assert c1.v == 3
  assert c2.v == 2
}
```

Aliasing

```
class Cell {
  int v

  void add(Cell c)
    requires c != null
    ensures v == old(v) + old(c.v)
    ensures c.v == old(c.v)
  { v = v + c.v }
}
```

```
void client() {
  Cell c1 = new Cell()
  c1.v = 1
  Cell c2 := new Cell()
  c2.v = 2

  c1.add(c1)

  assert c1.v == 3
  assert c2.v == 2
}
```

Challenges

Reason about Shared State (Including Data Races)

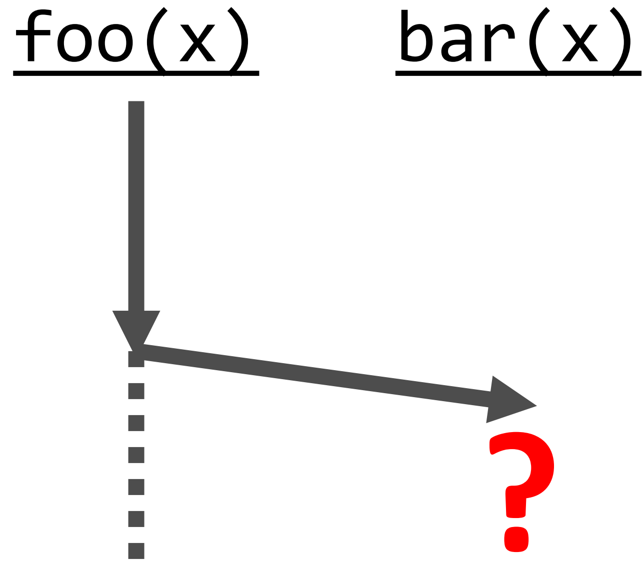
and

Control Aliasing

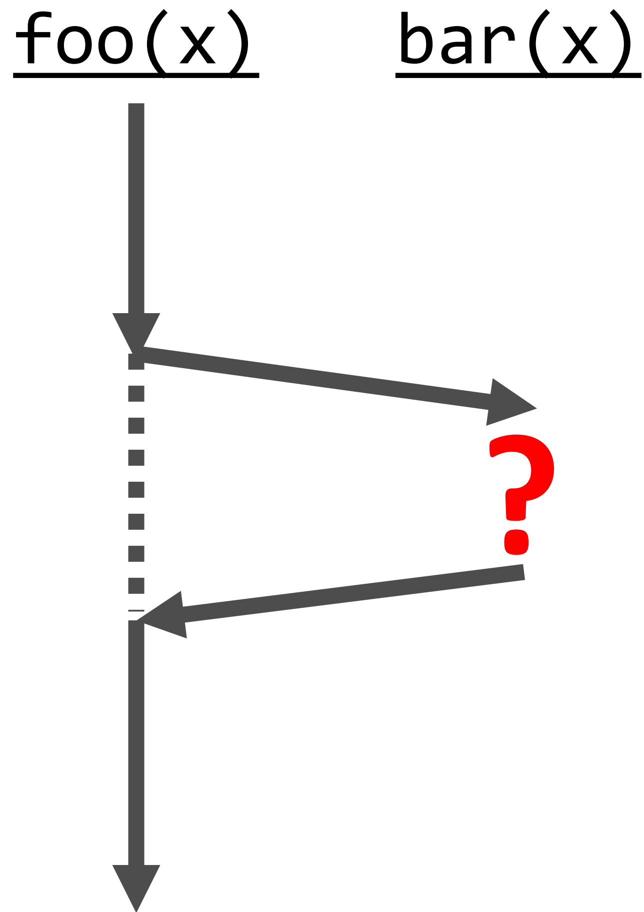
Modular Static Verification + Shared State

foo(x)   bar(x)

Modular Static Verification + Shared State

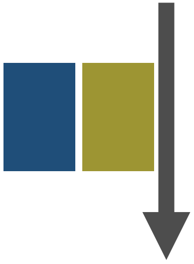


Modular Static Verification + Shared State

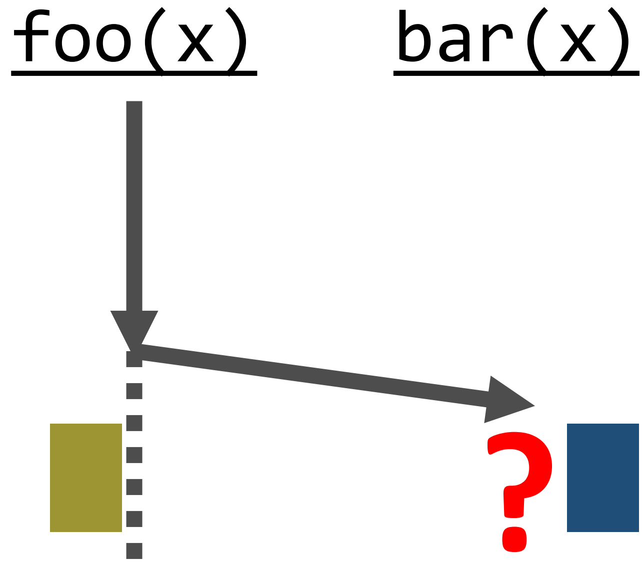


Permissions

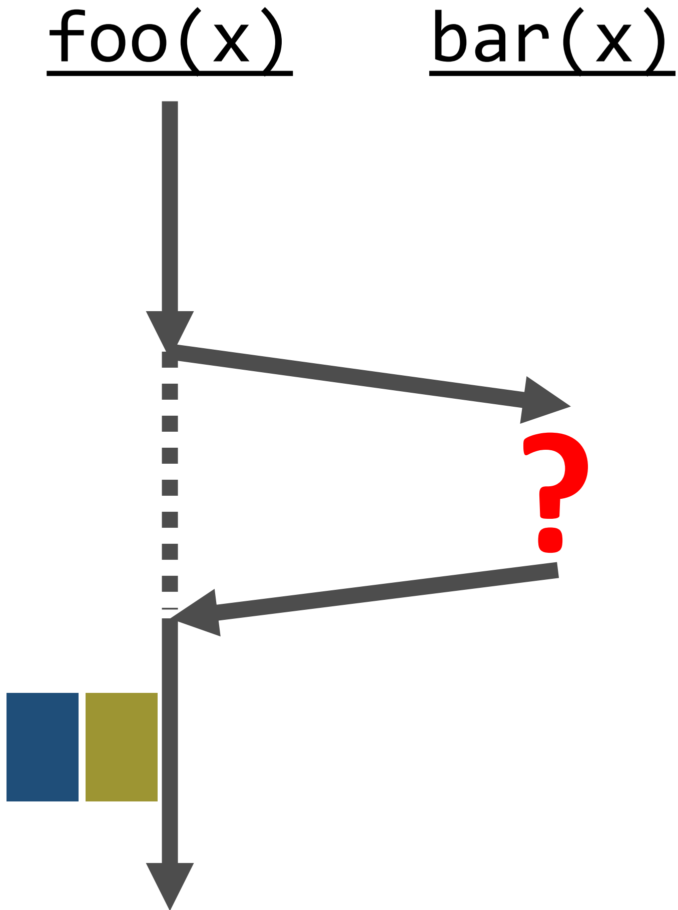
foo(x)   bar(x)



Permission Transfer

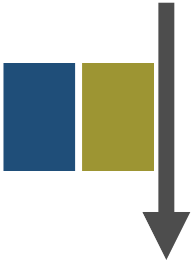


Permission Transfer

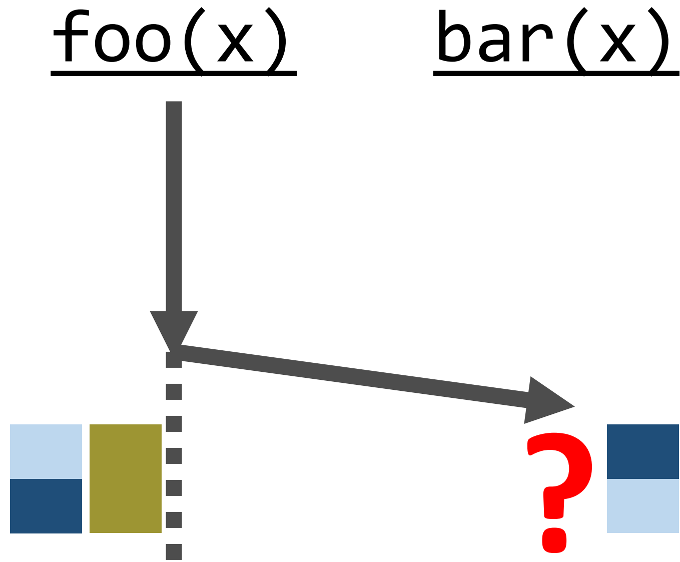


Fractional Permissions

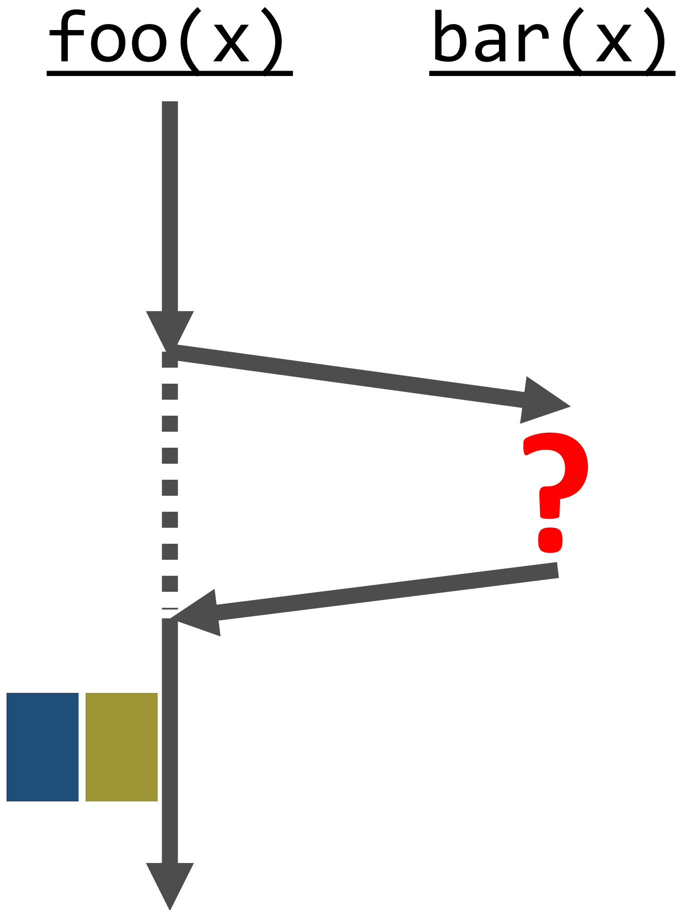
foo(x) bar(x)



Splitting Fractional Permissions



Merging Fractional Permissions



Silver: Assertion Language Basics

Accessibility predicates denote permissions

```
acc(c.f)
```

Assertions may be *heap-dependent*

```
acc(c.f) && c.f == 0
```

Fractional permissions

```
acc(c.f, ½)
```

Conjunction *sums up* permissions (similar to ∗ in separation logic)

```
acc(c.f,½) && acc(c.f,½)
```

Write permission is *exclusive* (similar to ∗ in separation logic)

```
acc(c1.f) && acc(c2.f,ε)
  ⇒ c1 != c2
```

Demo

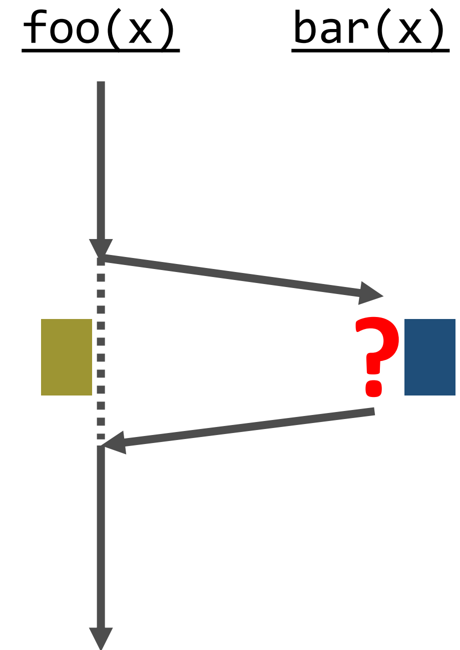
Permission Transfer Reloaded

Idea of *permission transfer* generalises

- Fork-join (transfer between threads)
- Locks (transfer to/from lock invariant)
- Message passing (pass permissions)

Common operations

- Gain permissions
- Lose permissions



Silver: Inhale and Exhale Statements

Statement `inhale A` means

- Gain permissions required by `A` (e.g. `acc(x.f)`)
- Assume logical constraints in `A` (e.g. `x.f != 0`)

Statement `exhale A` means

- Assert and remove permissions required by `A`
- Assert logical constraints in `A`
- Havoc locations to which all permissions were removed (i.e. forget their values)

Concurrency Examples

Fork-Join Concurrency (Pseudo-Java)

```
class Cell {
  int v

  void add(Cell c) {
    v = v + c.v
  }
}
```

```
void client() {
  Cell c1 = new Cell()
  c1.v = 1
  Cell c2 = new Cell()
  c2.v = 2

  Token tk = fork c1.add(c2)
  // ...
  join tk

  assert c1.v == 3
  assert c2.v == 2
}
```

Locks and @GuardedBy Annotations (Pseudo-Java)

```
class SharedPair {
  @GuardedBy(“this”)
  int x, y
```

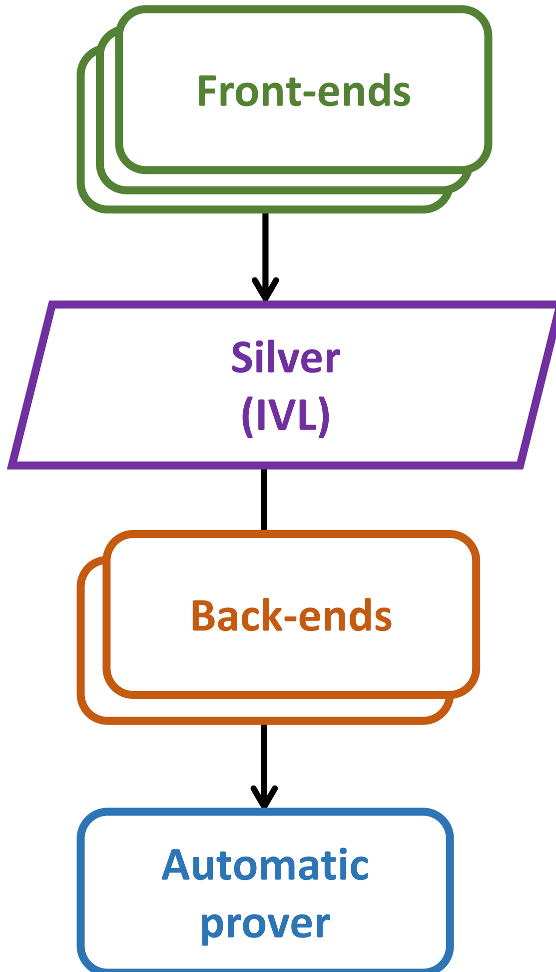
Lock Invariants (Pseudo-Java)

```
class SharedPair {
  @GuardedBy(“this”, “x < y”)
  int x, y









}
```

Lock Invariants (Pseudo-Java)

```
class SharedPair {
  @GuardedBy("this", "x < y")
  int x, y

  void inc(int dx, int dy) {
    assert dx <= dy
    synchronized(this) {
      assert x < y
      x = x + dx
      y = y + dy
      assert x < y
    }
  }
}
```

Viper: Our Verification Infrastructure



Silver:

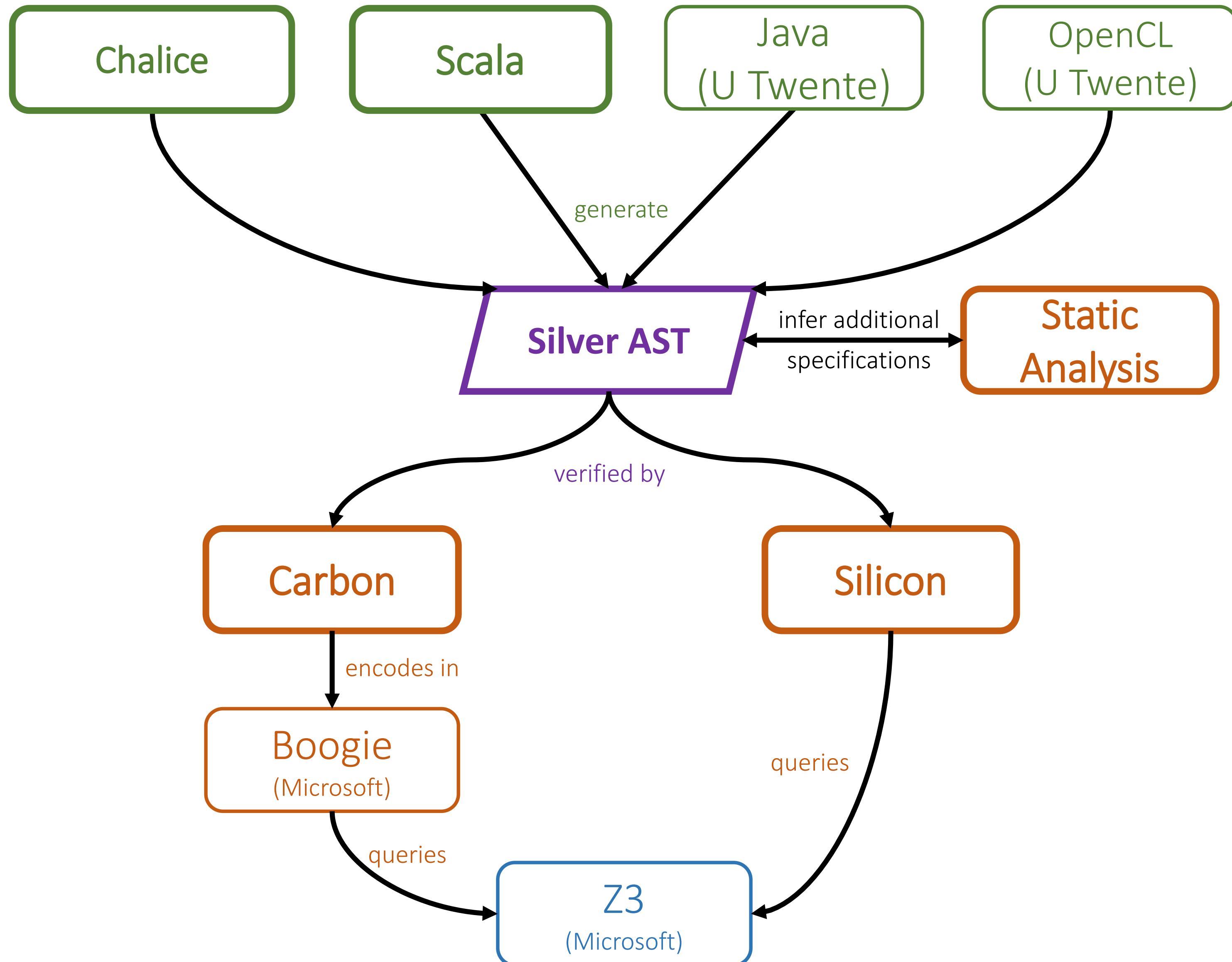
- Intermediate Verification Language
- Few (but expressive) constructs
- Designed with verification and inference in mind

Back-ends: Two verifiers; plans to develop inference, slicer

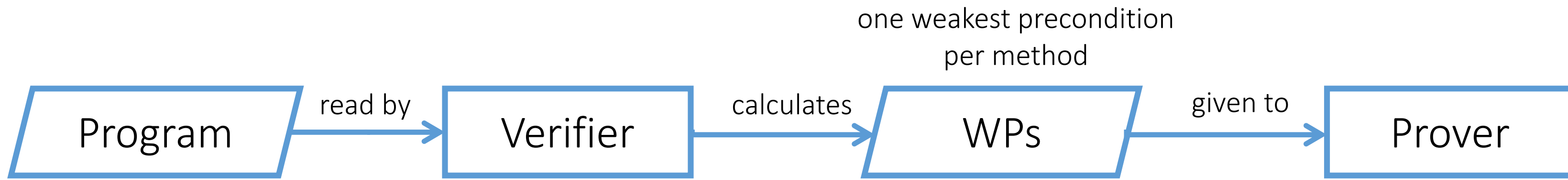
Front-ends (proof of concept):

- *Chalice* (concurrency research)
- *Scala* (very small subset)
- *Java* (VerCors, U Twente)
- *OpenCL* (VerCors, U Twente)

Viper: Our Verification Infrastructure

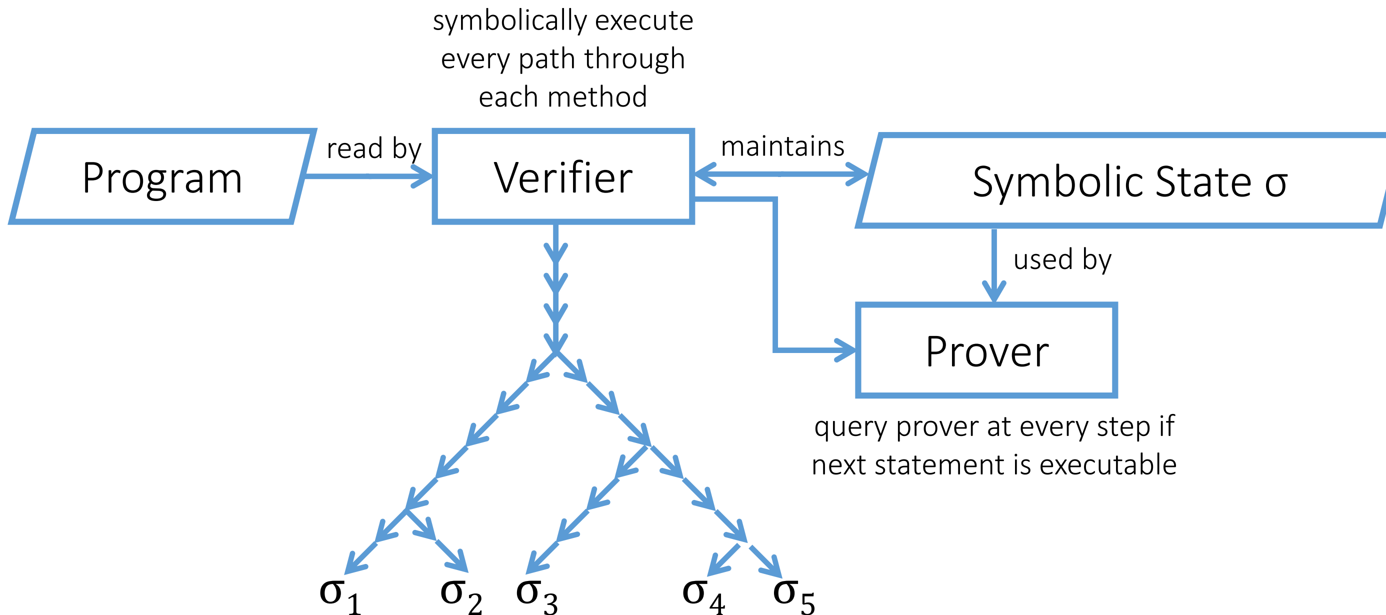


Verification Condition Generation vs. Symbolic Execution



Query prover once with full information (Carbon)

VCG



Query prover often with limited information (Silicon)

SE

Outlook

Information hiding, abstraction and inheritance

Unbounded (recursive) data structures

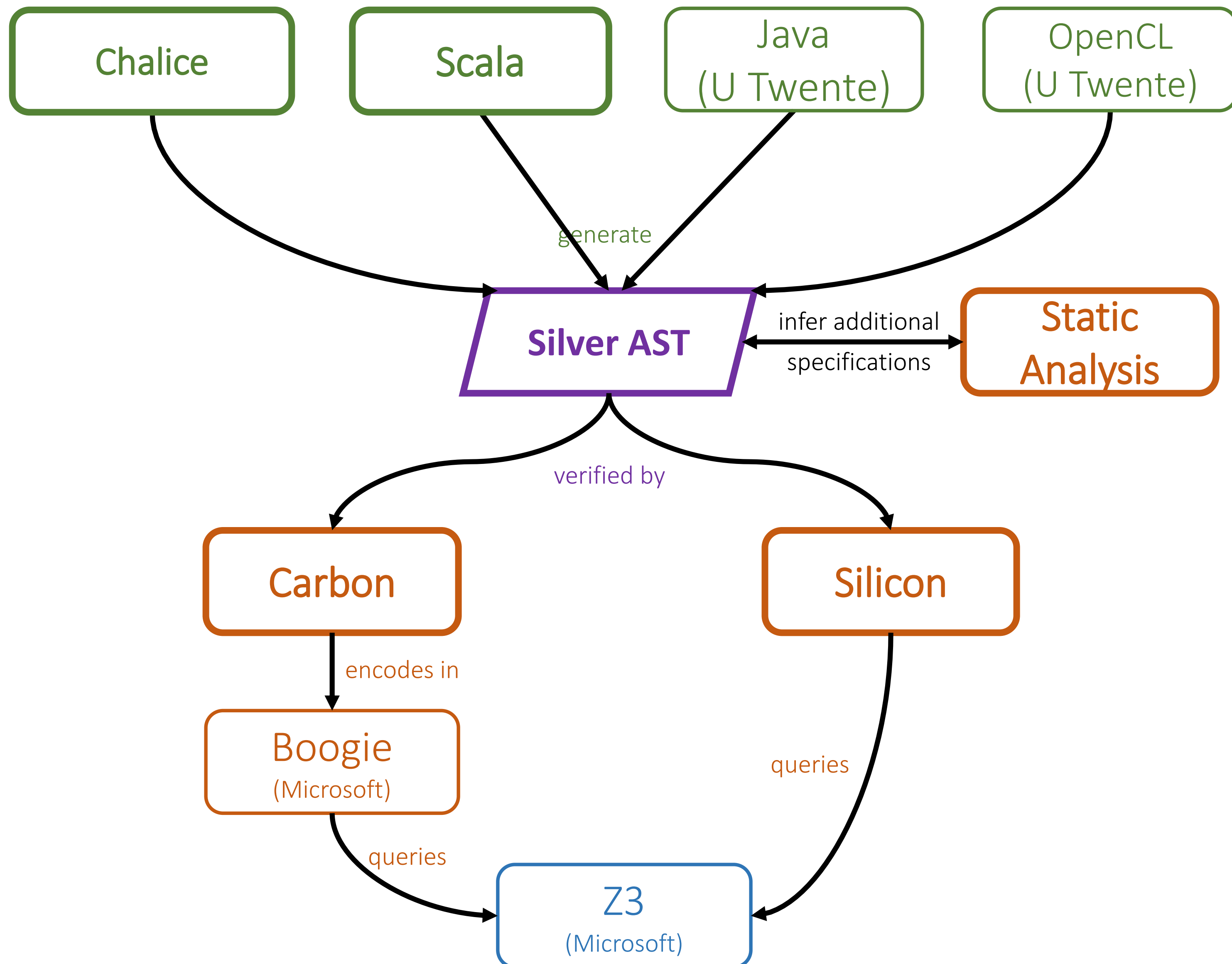
Obligations – the dual to permissions

Specification inference

Encoding of high-level features

- Immutable data (vs. permissions)
- Lazy evaluation (vs. permissions)
- Closures/higher order functions
- Actor-based concurrency
- Fine-grained locking, lock-free algorithms

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



You shouldn't even be here!

Fork-Join Concurrency (Pseudo-Java)

```
class Cell {
  int v

  void add(Cell c) {
    v = v + c.v
  }
}
```

```
void client() {
  Cell c1 = new Cell()
  c1.v = 1
  Cell c2 = new Cell()
  c2.v = 2

  Token tk = fork c1.add(c2)
  // ...
  join tk

  assert c1.v == 3
  assert c2.v == 2
}
```

Fork-Join Concurrency (Silver)

```
field v: Int

method add(this: Ref, c: Ref)
  requires acc(this.v) && acc(c.v, ½)
  ensures  acc(this.v) && acc(c.v, ½)
  ensures  this.v == old(this.v) + old(c.v)
{
  this.v := this.v + c.v
}
```

Locks and @GuardedBy Annotations (Pseudo-Java)

```
class SharedPair {
  @GuardedBy(“this”)
  int x, y
```

Locks and @GuardedBy Annotations (Silver)

```
field x: Int
field y: Int

define inv(this) acc(this.x) && acc(this.y)
```

Lock Invariants (Pseudo-Java)

```
class SharedPair {
  @GuardedBy(“this”, “x < y”)
  int x, y

}
```

Lock Invariants (Pseudo-Java)

```
class SharedPair {
  @GuardedBy("this", "x < y")
  int x, y

  void inc(int dx, int dy) {
    assert dx <= dy

    synchronized(this) {
      assert x < y

      x = x = dx
      y = y = dy

      assert x < y
    }
  }
}
```

Lock Invariants (Silver)

```
field x: Int
field y: Int

define inv(this)    acc(this.x) && acc(this.y)
                 && this.x <= this.y
```