Viper

A Verification Infrastructure for Permission-Based Reasoning

Quantified Permissions

Dynamic-Frames-Style Specifications in Permission Logics

ETH zürich

Malte Schwerhoff

3rd November 2016, Bad Herrenalb

Frame Problem

$$\frac{\{\,P\,\}\ C\ \{\,Q\,\}}{\{\,P \wedge R\,\}\ C\ \{\,Q \wedge R\,\}}$$

Framing Methodologies

$$\frac{\{ P \}\ C\ \{ Q \}}{\{ P \wedge R \}\ C\ \{ Q \wedge R \}}$$

Dynamic Frames
(no permissions)

$$\frac{\{ P \}\ C\ \{ Q \} \quad \text{modifies}(C) \cap \text{reads}(R) = \emptyset}{\{ P \wedge R \}\ C\ \{ Q \wedge R \}}$$

Framing Methodologies

~~$$\frac{\{\,P\,\}\ C\ \{\,Q\,\}}{\{\,P \wedge R\,\}\ C\ \{\,Q \wedge R\,\}}$$~~

Dynamic Frames
(no permissions)

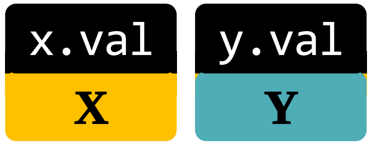
$$\frac{\{\,P\,\}\ C\ \{\,Q\,\} \qquad \mathrm{modifies}(C) \cap \mathrm{reads}(R) = \emptyset}{\{\,P \wedge R\,\}\ C\ \{\,Q \wedge R\,\}}$$

Separation Logic
(permissions)

$$\frac{\{\,P\,\}\ C\ \{\,Q\,\}}{\{\,P \ast R\,\}\ C\ \{\,Q \ast R\,\}}$$

Permissions

```
method mutate()
  requires acc(this.val)
  ensures  acc(this.val)
```




```
method client(x, y)
  requires acc(x.val) * acc(y.val)
{
  var tmp := y.val
  x.mutate()
  assert tmp == y.val
}
```

Permissions

```
method mutate()
  requires acc(this.val)
  ensures  acc(this.val)
```

x.val	y.val
X	Y

```
method client(x, y)
  requires acc(x.val) * acc(y.val)
{
  var tmp := y.val
  x.mutate()
  assert tmp == y.val
}
```

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$

Permissions

```
method mutate()
  requires acc(this.val)
  ensures  acc(this.val)
```

x.val
X

y.val
Y

```
method client(x, y)
  requires acc(x.val) * acc(y.val)
{
  var tmp := y.val
  x.mutate()
  assert tmp == y.val
}
```

$$\frac{\{\,P\,\}\ C\ \{\,Q\,\}}{\{\,P * R\,\}\ C\ \{\,Q * R\,\}}$$

Permissions

```
method mutate()
  requires acc(this.val)
  ensures  acc(this.val)
```

x.val
?

y.val
Y

```
method client(x, y)
  requires acc(x.val) * acc(y.val)
{
  var tmp := y.val
  x.mutate()
  assert tmp == y.val
}
```

$$\frac{\{\,P\,\}\ C\ \{\,Q\,\}}{\{\,P * R\,\}\ C\ \{\,Q * R\,\}}$$

Permissions

```
method mutate()
  requires acc(this.val)
  ensures  acc(this.val)
```

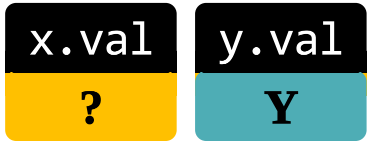
x.val	y.val
?	Y

```
method client(x, y)
  requires acc(x.val) * acc(y.val)
{
  var tmp := y.val
  x.mutate()
  assert tmp == y.val
}
```

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$

Permissions

```
method mutate()
  requires acc(this.val)
  ensures  acc(this.val)
```

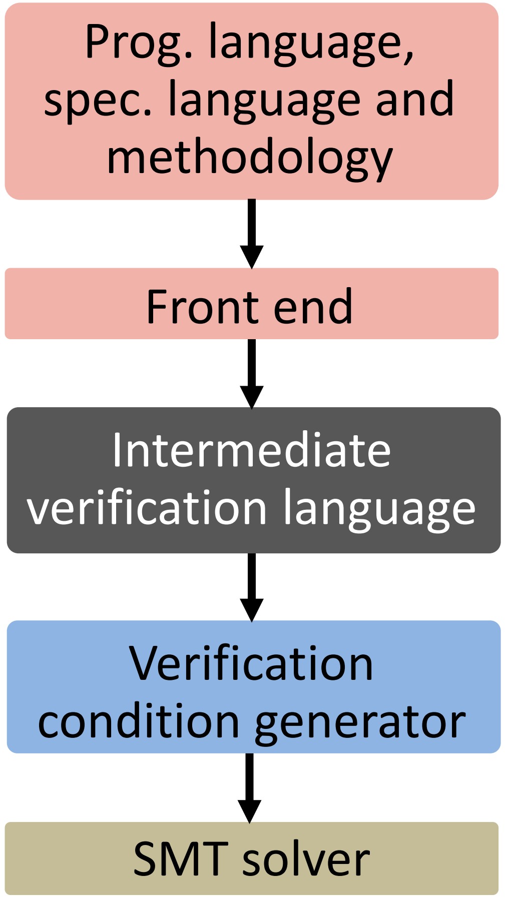



```
method client(x, y)
  requires acc(x.val) * acc(y.val)
{
  var tmp := y.val
  x.mutate()
  assert tmp == y.val
}
```

$$\frac{\{\,P\,\}\ C\ \{\,Q\,\}}{\{\,P * R\,\}\ C\ \{\,Q * R\,\}}$$

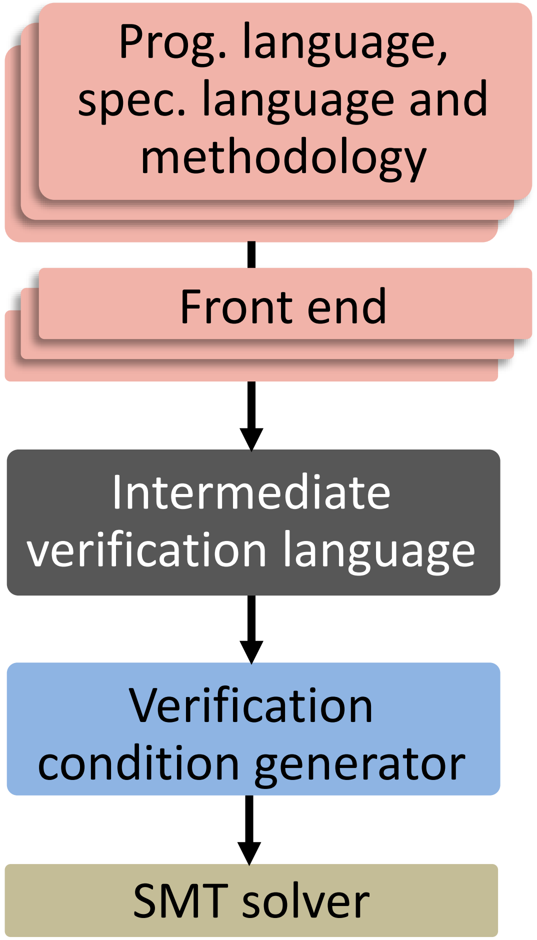
Common Tool Infrastructures

No Permissions

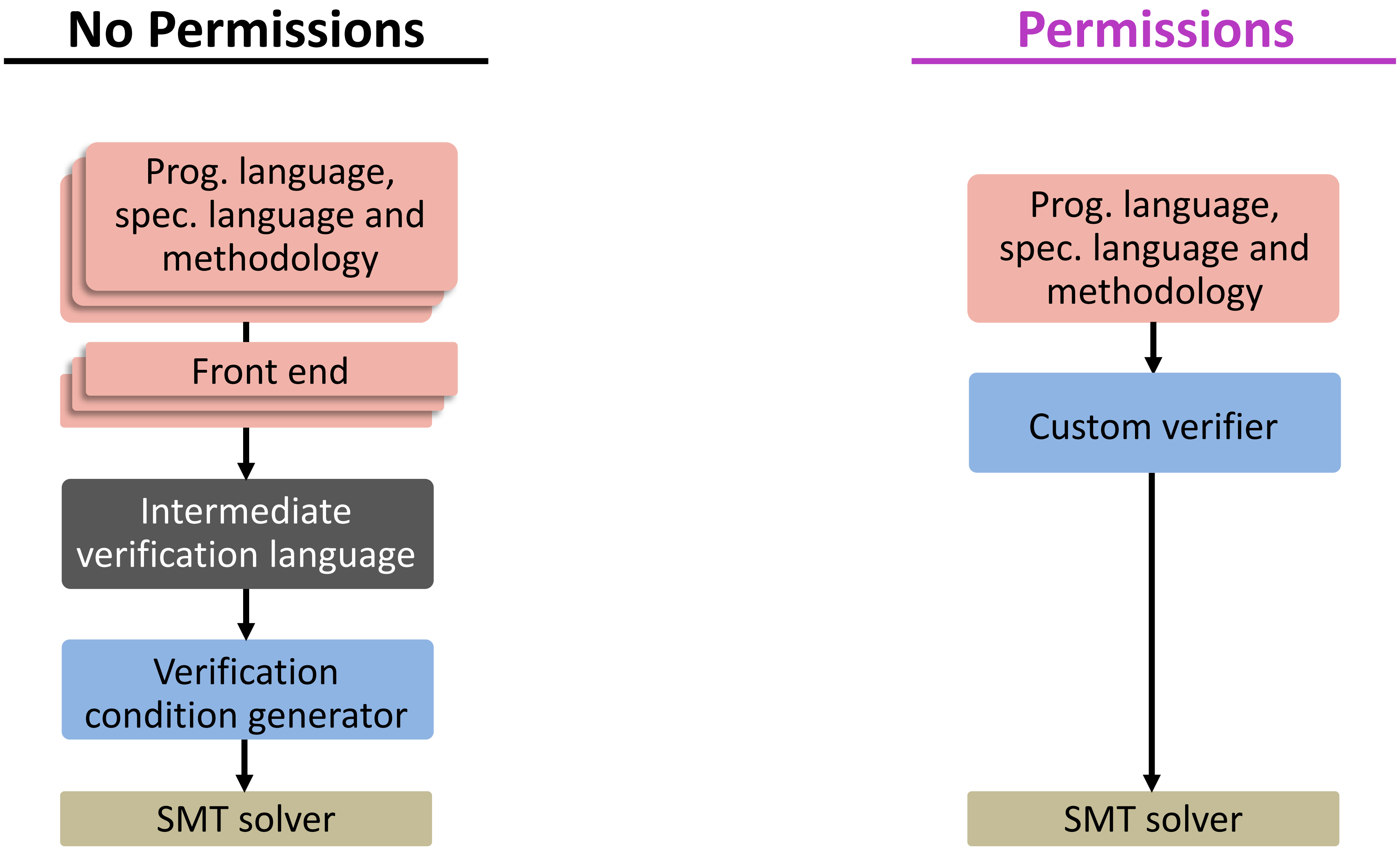


Common Tool Infrastructures

No Permissions

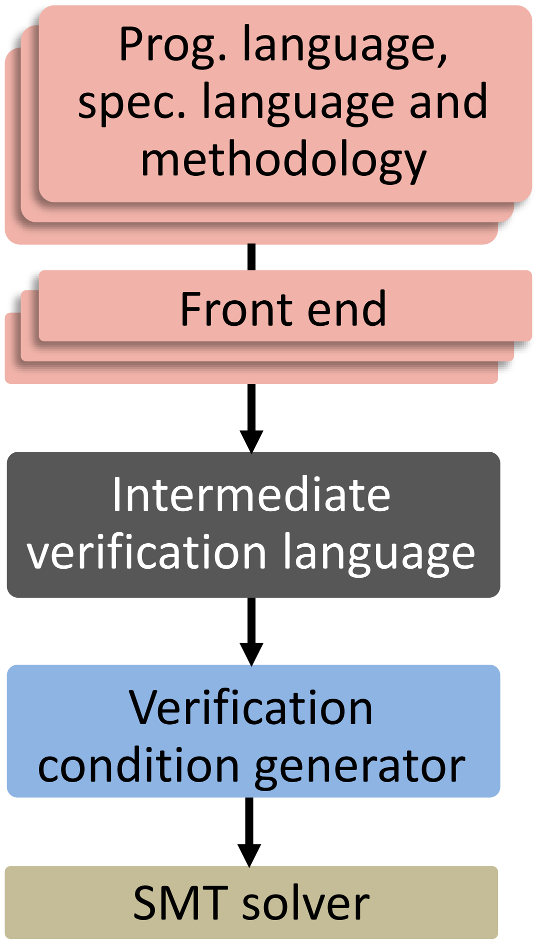


Common Tool Infrastructures

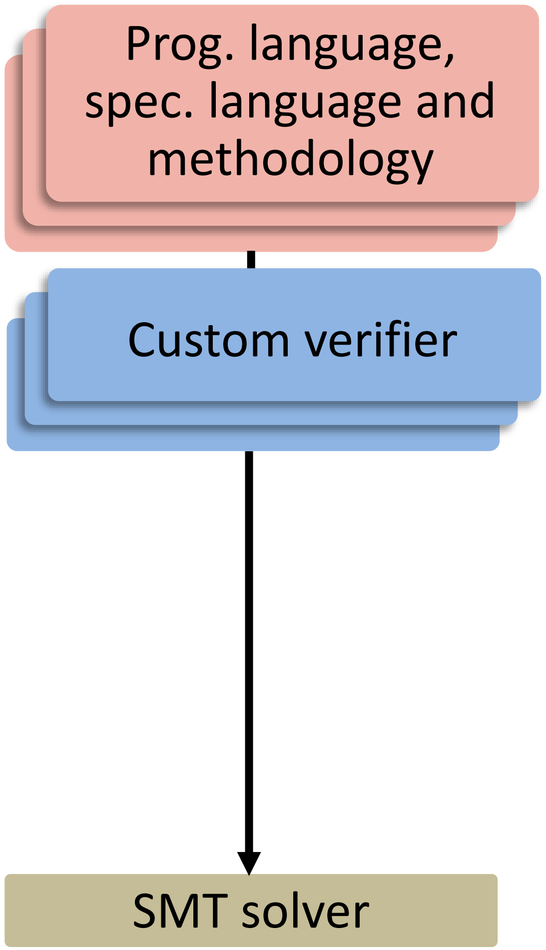


Common Tool Infrastructures

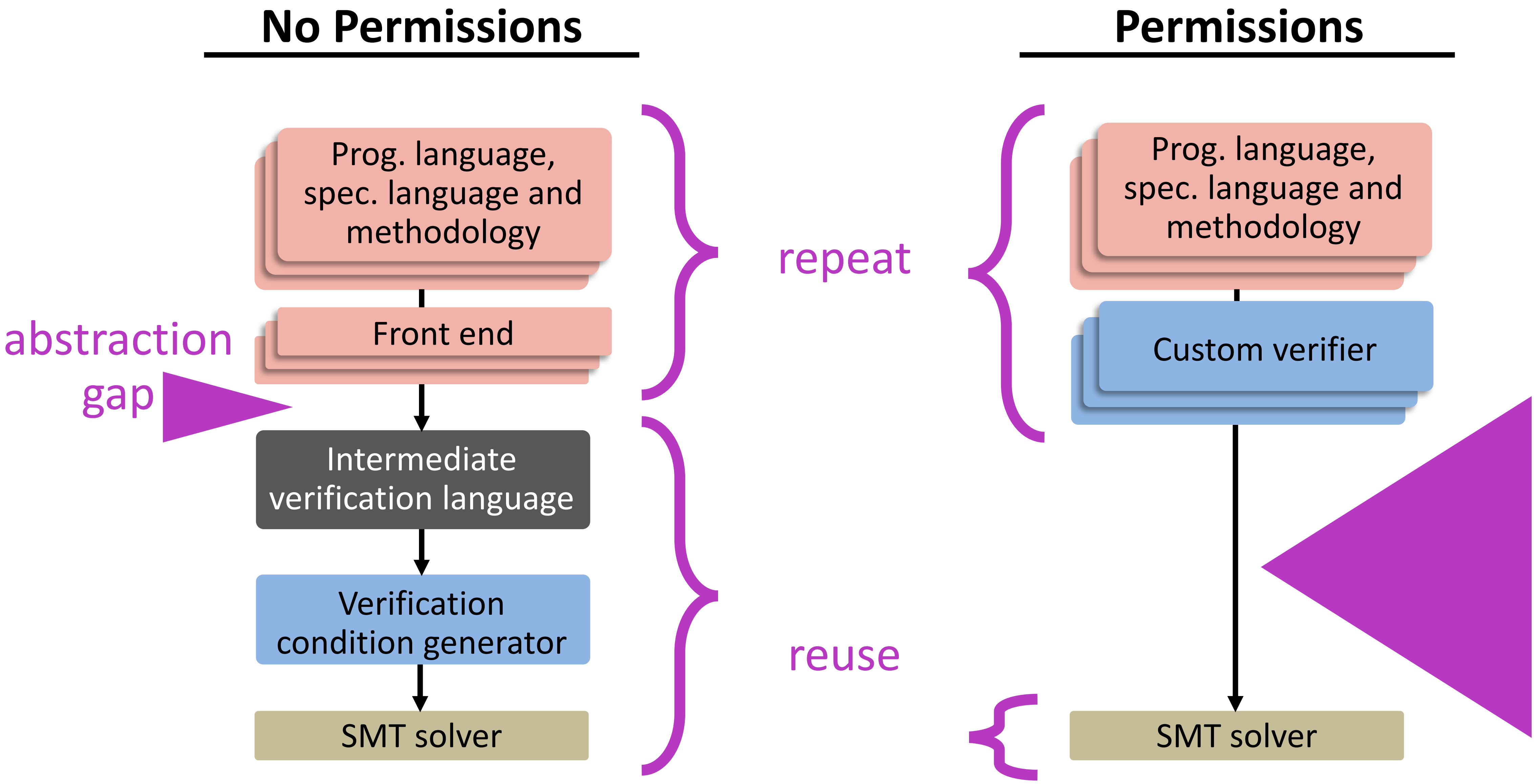
No Permissions



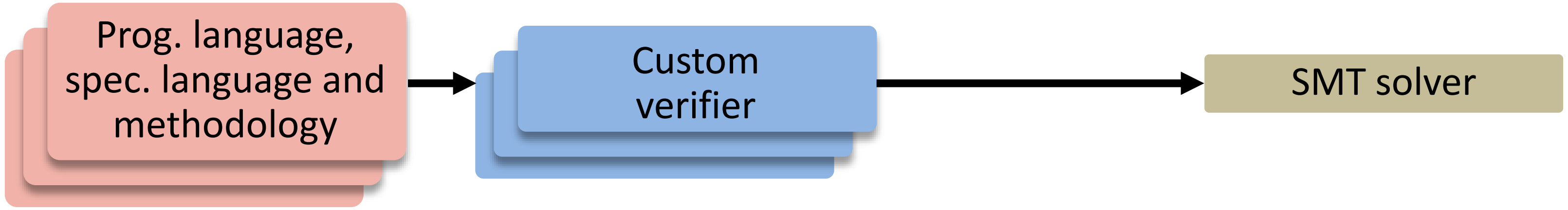
Permissions

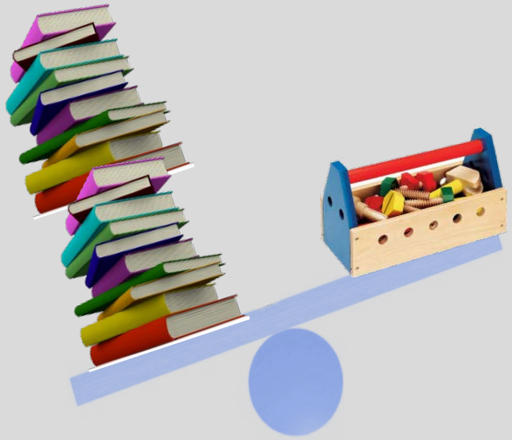


Common Tool Infrastructures



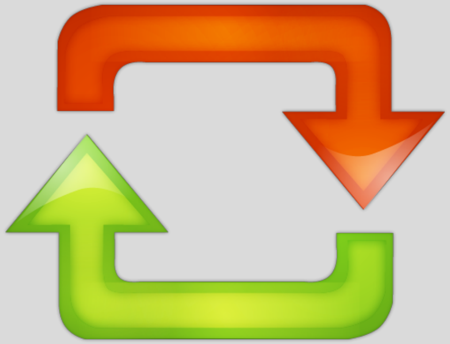
Insufficient Tool Support for Permission Logics





Verification efforts do not benefit fully from advances in theory

Theory does not receive feedback from applications



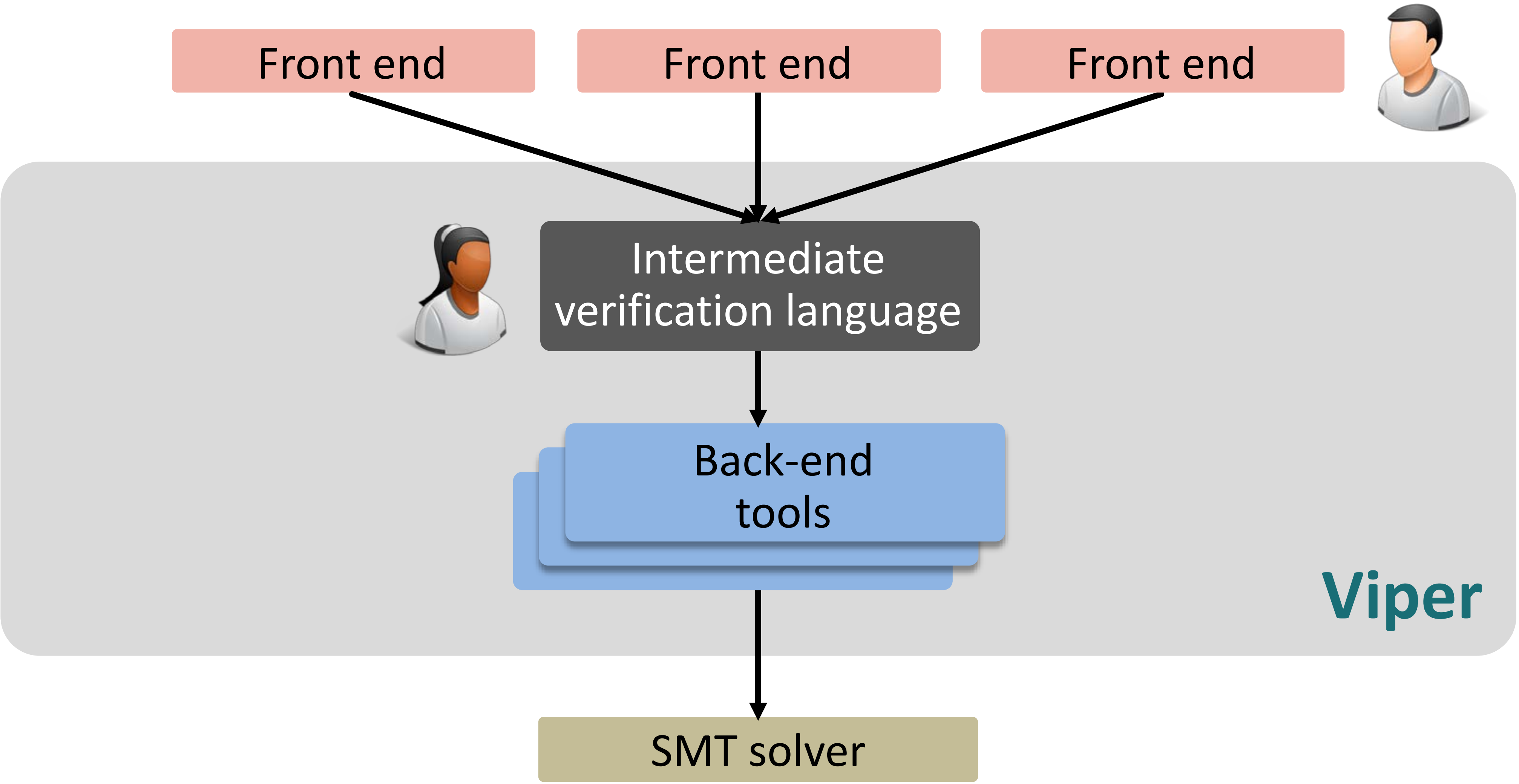
Images: https://www.flickr.com/photos/61423903@N06/7357608430
https://www.flickr.com/photos/rustedhammer/2298322271
http://www.iconarchive.com/show/misc-icons-by-tooschee/Sync-icon.html

Facilitate the

1. development of tools
2. prototyping of encodings

for permission-based verification

Viper



Icons: http://www.iconarchive.com/show/vista-people-icons-by-icons-land.html
http://www.icons-land.com/

Permission Transfer

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$

$$\frac{\text{callee}}{\text{caller}}$$

Viper Features: Inhale and Exhale

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}} \qquad \frac{\text{callee}}{\text{caller}}$$

`exhale P`

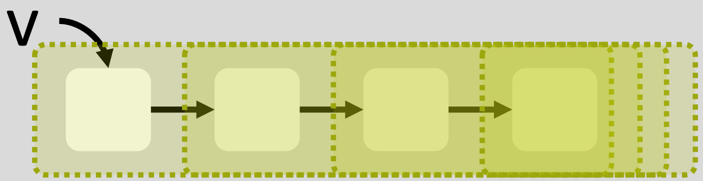
- assert value constraints
- check and remove permissions
- havoc newly-inaccessible locations

`inhale Q`

- obtain permissions
- assume value constraints

Demo

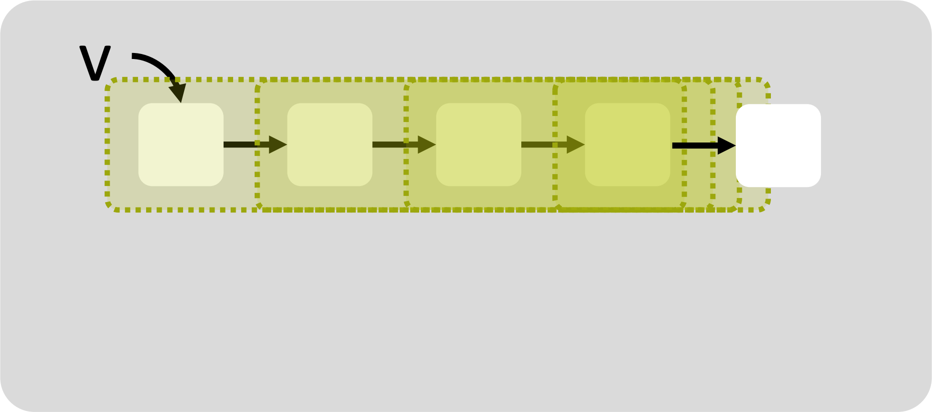
Recursive Predicates




```
predicate list(this: Ref) {
  this != null ==>
    acc(this.data) &&
    acc(this.next) &&
    list(this.next)
}
```

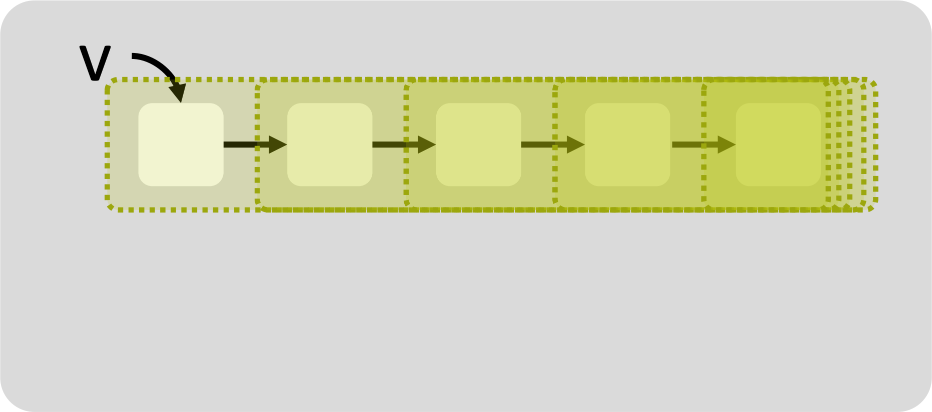
```
unfold list(this)
// access this.data
// and this.next
fold list(this)
```

Recursive Predicates: Limitations



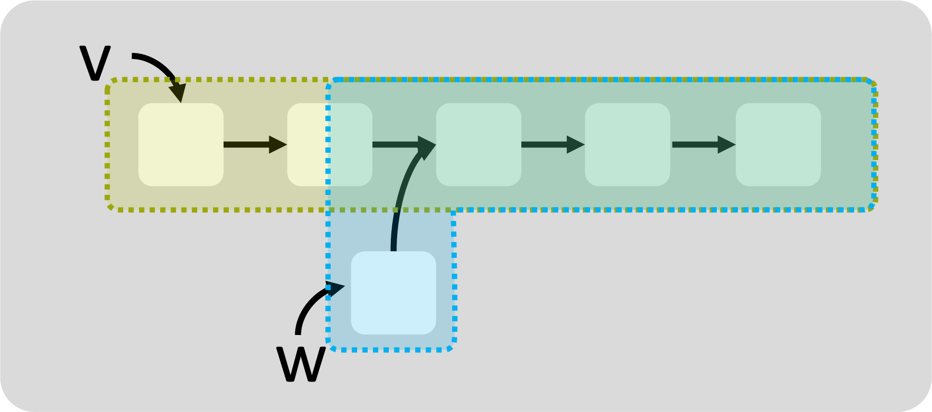
1. Extending

Recursive Predicates: Limitations



1. Extending

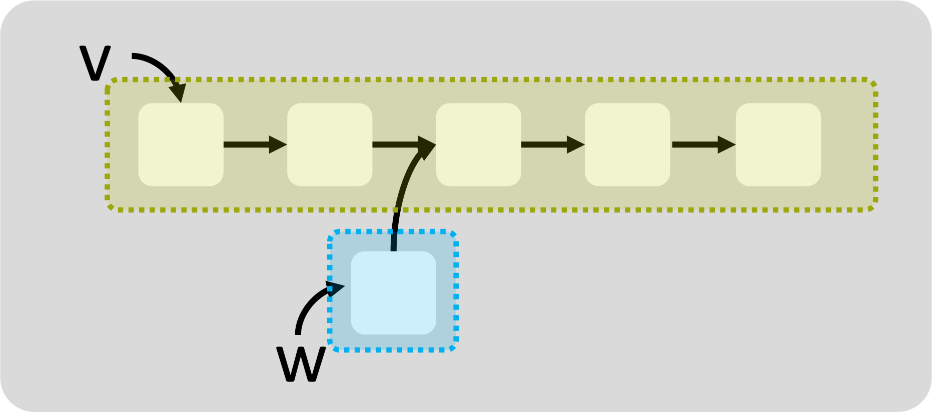
Recursive Predicates: Limitations



1. Extending

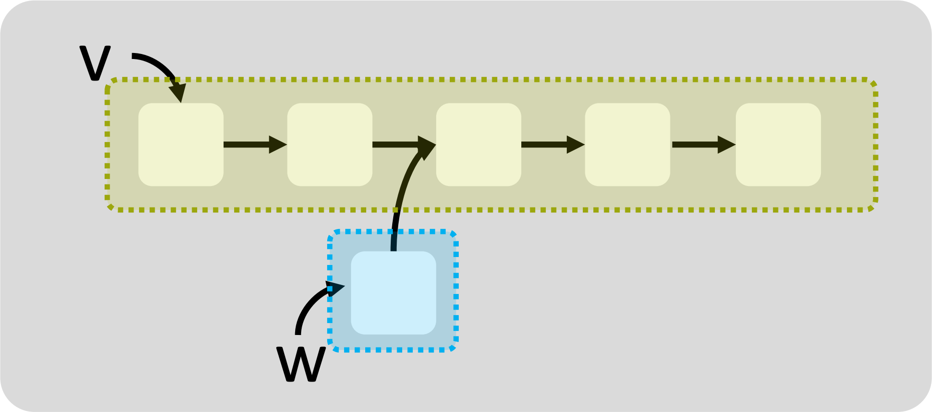
2. Sharing

Recursive Predicates: Limitations



1. Extending
2. Sharing

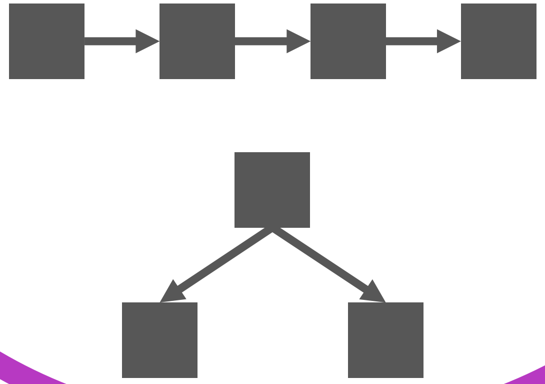
Recursive Predicates: Limitations



1. Extending
2. Sharing
3. Traversing

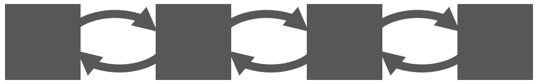
Unbounded Data Structures

Unidirectional



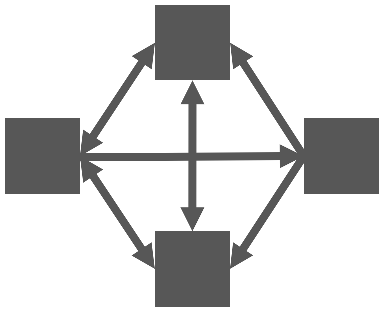
recursive predicates
are often a suitable
specification mechanism

Multidirectional



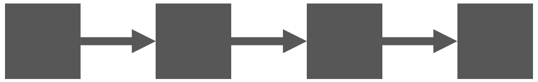
Random Access

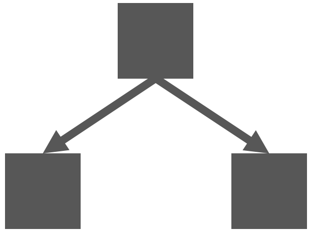
Unstructured



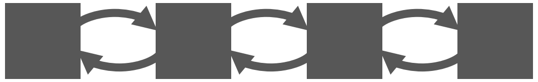
Unbounded Data Structures

Unidirectional



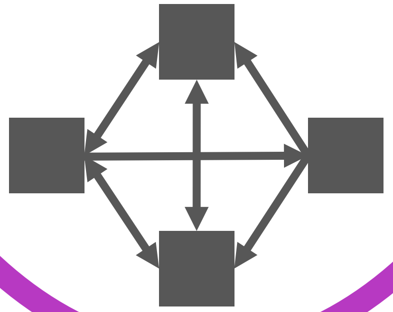


Multidirectional



Random Access

Unstructured

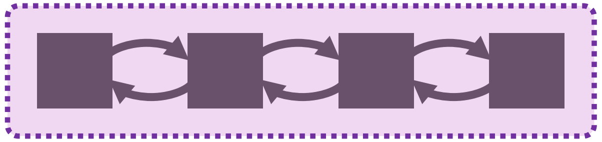


need for an alternative
specification mechanism

Quantified Permissions

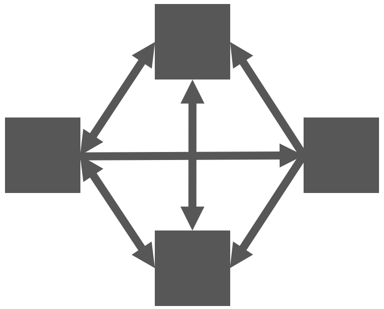
```
forall n in nodes ::
  acc(n.next) && acc(n.prev)
```

Multidirectional



Random Access

Unstructured

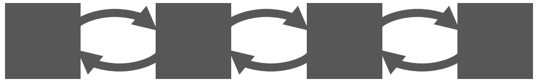


Quantified Permissions

```
forall n in nodes ::
  acc(n.next) && acc(n.prev)
```

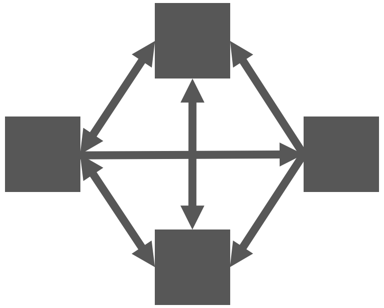
```
forall i in [0..5] ::
  acc(arr[i])
```

Multidirectional



Random Access

Unstructured



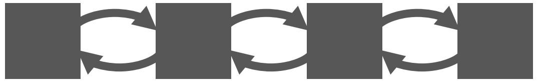
Quantified Permissions

```
forall n in nodes ::
  acc(n.next) && acc(n.prev)
```

```
forall i in [0..5] ::
  acc(arr[i])
```

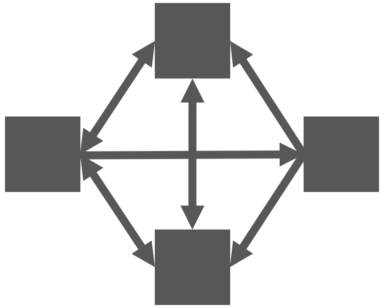
```
forall i in [0..5] ::
  i % 2 == 1 ==> acc(arr[i])
```

Multidirectional



Random Access

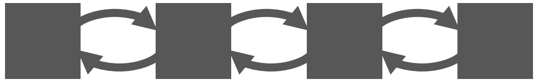
Unstructured



Quantified Permissions

```
forall n in nodes ::
  acc(n.next) && acc(n.prev)
```

Multidirectional



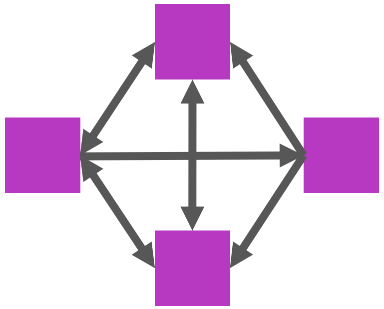
```
forall i in [0..5] ::
  acc(arr[i])
```

```
forall i in [0..5] ::
  i % 2 == 1 ==> acc(arr[i])
```

Random Access

```
forall n in nodes ::
  acc(n.succs) && acc(n.marked)
```

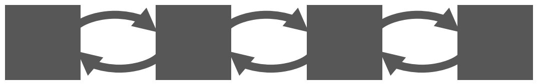
Unstructured



Quantified Permissions

```
forall n in nodes ::
  acc(n.next) && acc(n.prev)
```

Multidirectional

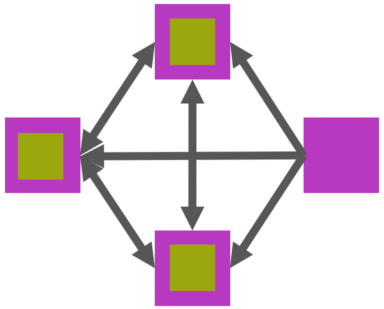


```
forall i in [0..5] ::
  acc(arr[i])
```

Random Access

```
forall i in [0..5] ::
  i % 2 == 1 ==> acc(arr[i])
```

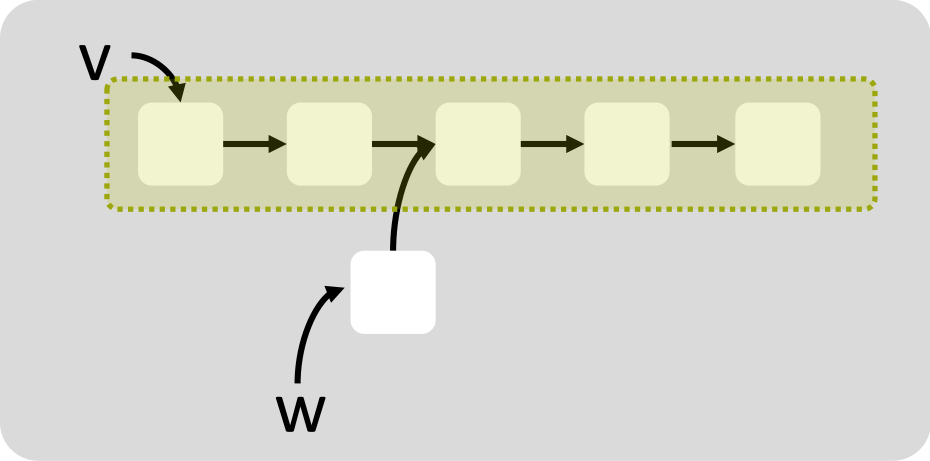
Unstructured



```
forall n in nodes ::
  acc(n.succs) && acc(n.marked) &&
    (n.marked ==>
      forall m in n.succs :: m.marked)
```

List Tail Sharing Revisited

```
predicate list(nodes: Set[Ref]) {
  forall n ∈ nodes ::
    acc(n.data) &&
    acc(n.next) &&
    (n.next != null ==>
        n.next ∈ nodes)
}
```




```
list(nodes) &&
v ∈ nodes &&
w.next ∈ nodes
```

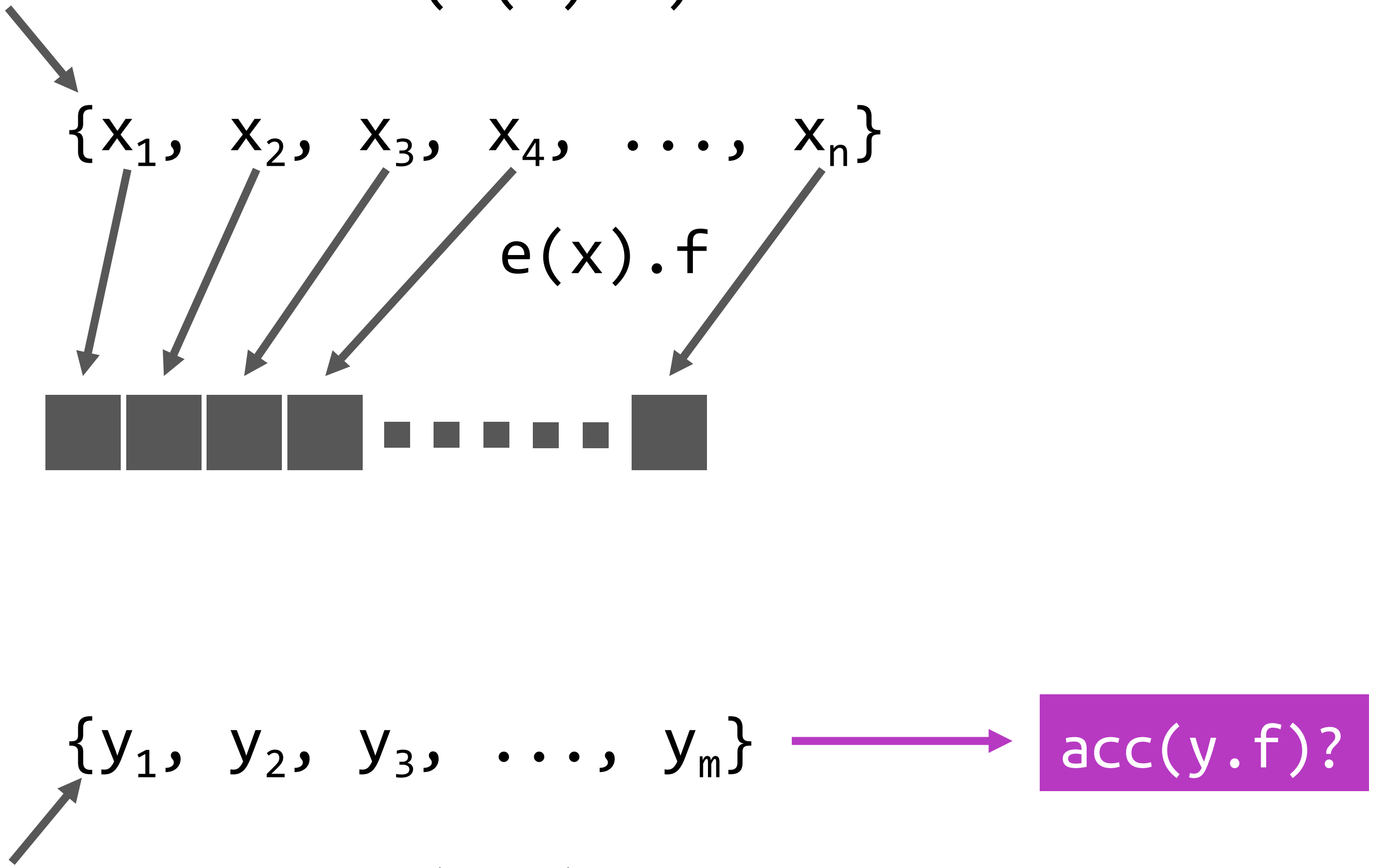
General Receiver Expressions

```
inhale ∀ x ∈ S :: acc(e(x).f)
```

```
exhale ∀ y ∈ R :: acc(y.f)
```

General Receiver Expressions: Challenge

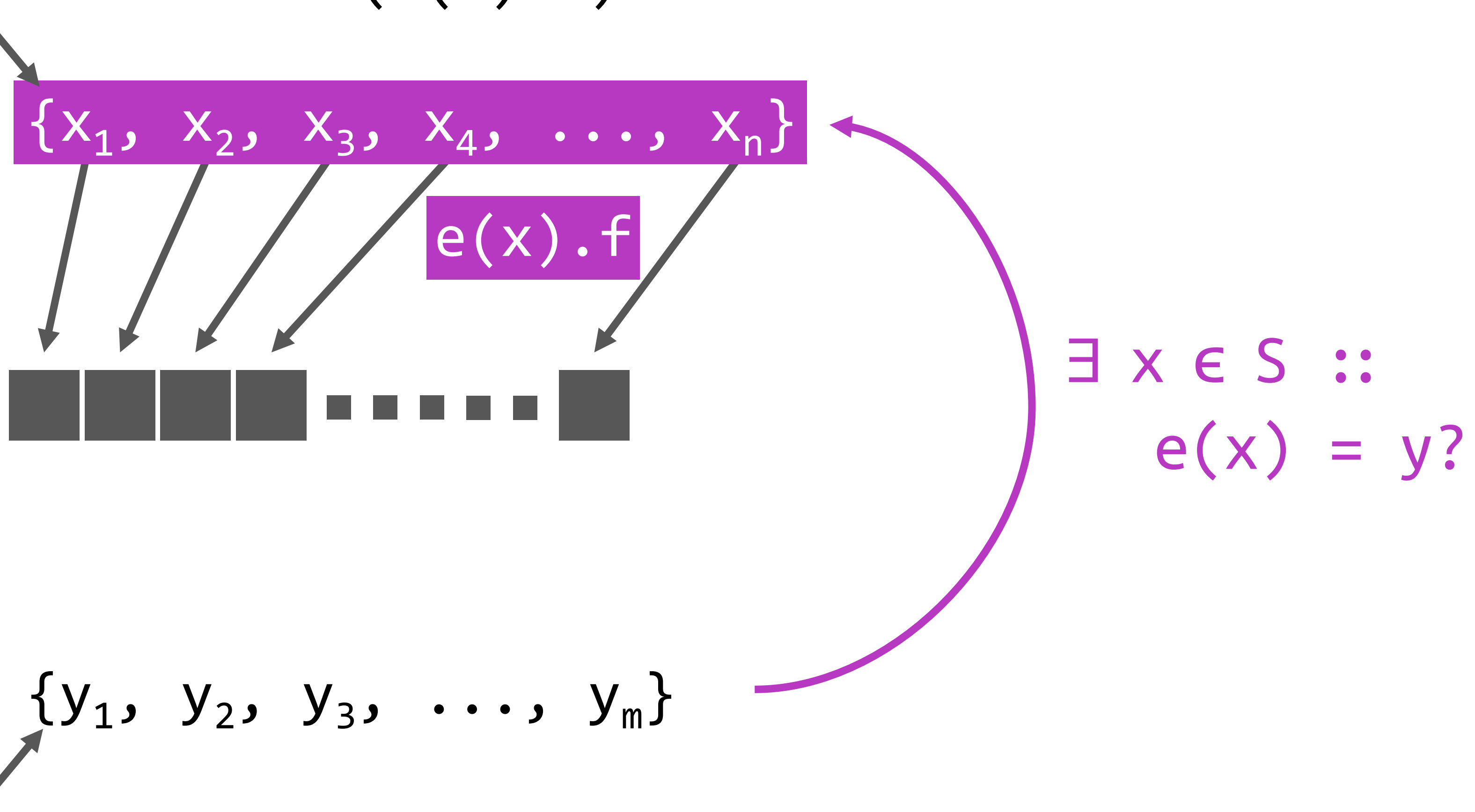
inhale ∀ x ∈ S :: acc(e(x).f)



exhale ∀ y ∈ R :: acc(y.f)

General Receiver Expressions: Challenge

inhale ∀ x ∈ S :: acc(e(x).f)



exhale ∀ y ∈ R :: acc(y.f)

General Receiver Expressions: Injectivity

```
inhale ∀ x ∈ S :: acc(e(x).f)
```



```
exhale ∀ y ∈ R :: acc(y.f)
```

1. Require e(x) to be injective

   (naturally satisfied by e.g. arrays and graphs)

General Receiver Expressions: Inverse Functions

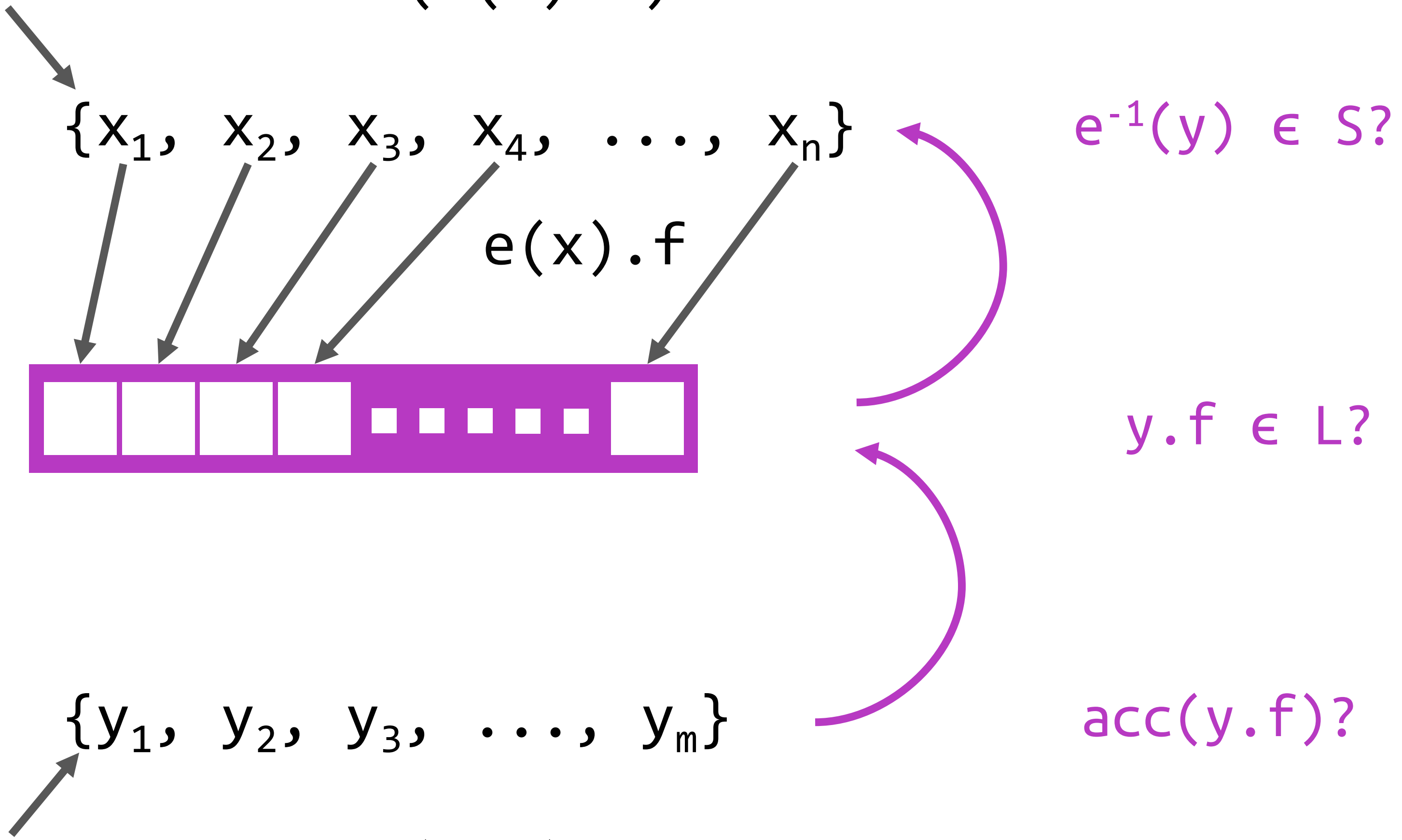
`inhale ∀ x ∈ S :: acc(e(x).f)`

`exhale ∀ y ∈ R :: acc(y.f)`

1. Require e(x) to be injective
2. Axiomatise inverse function $e^{-1}(x)$ to SMT solver

General Receiver Expressions: Challenge

inhale ∀ x ∈ S :: acc(e(x).f)



exhale ∀ y ∈ R :: acc(y.f)

Demo

Dynamic Frames vs. Permissions

Permission Logics: disjointness per default

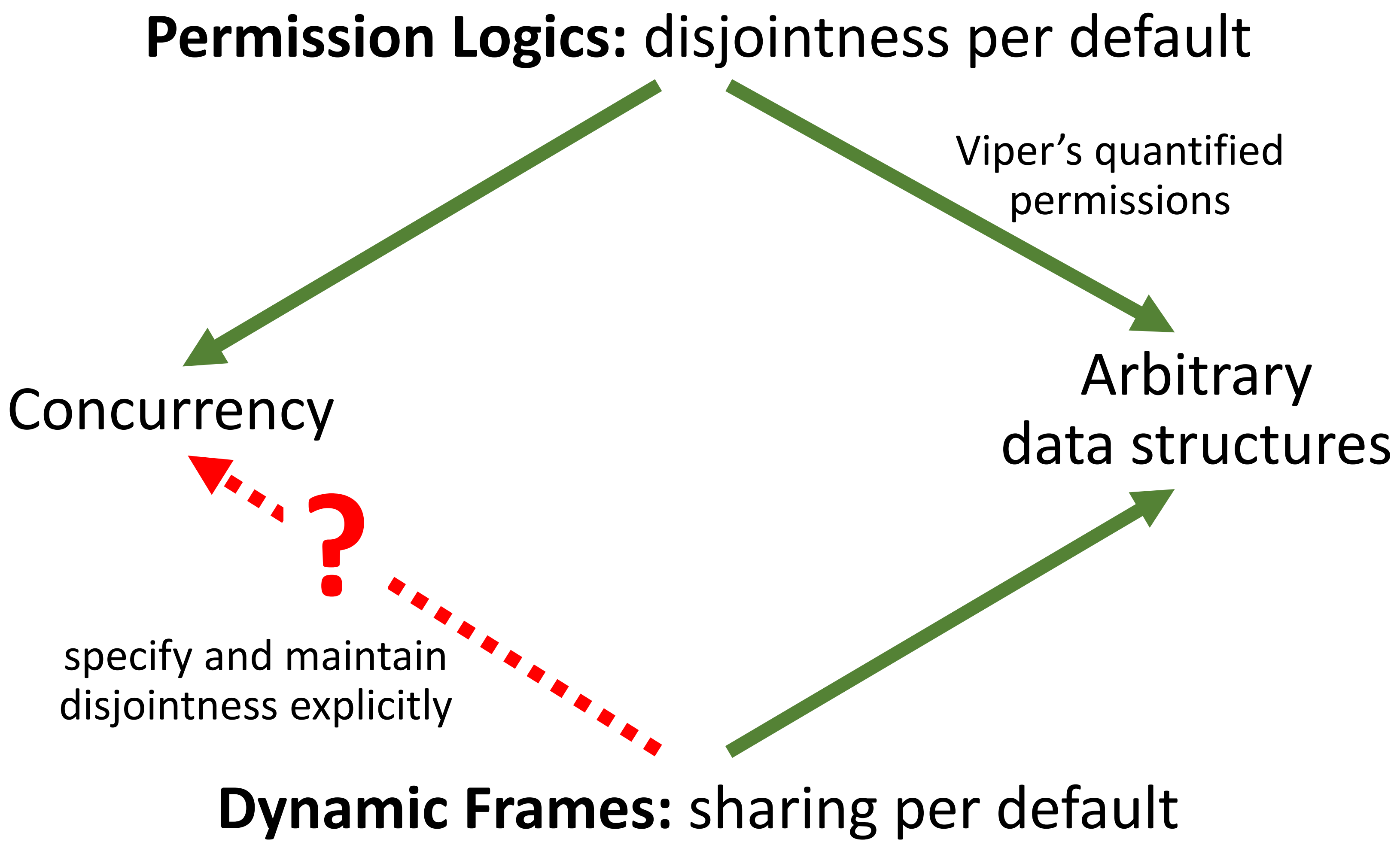
poorly supported
in tools

Concurrency

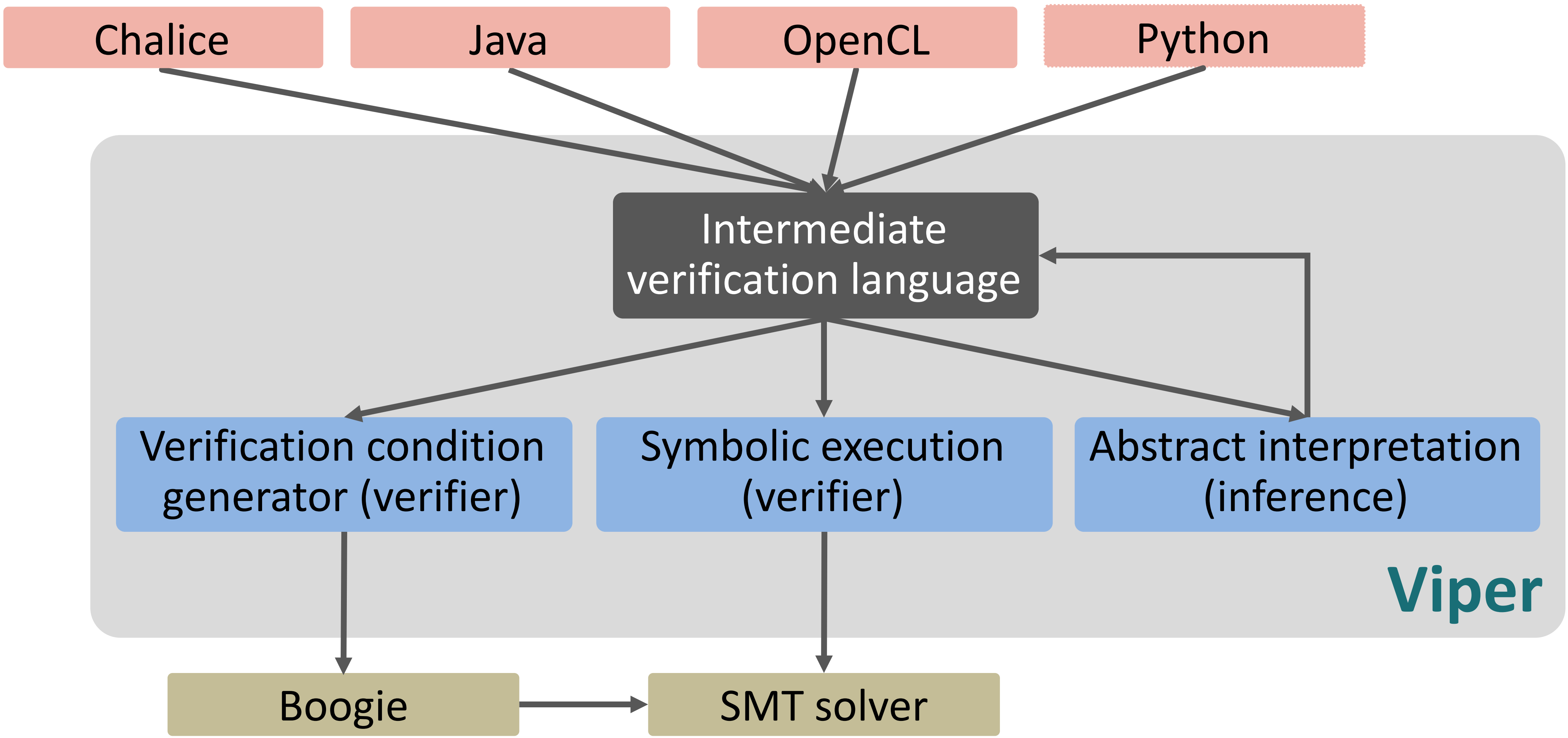
Arbitrary
data structures

Dynamic Frames: sharing per default

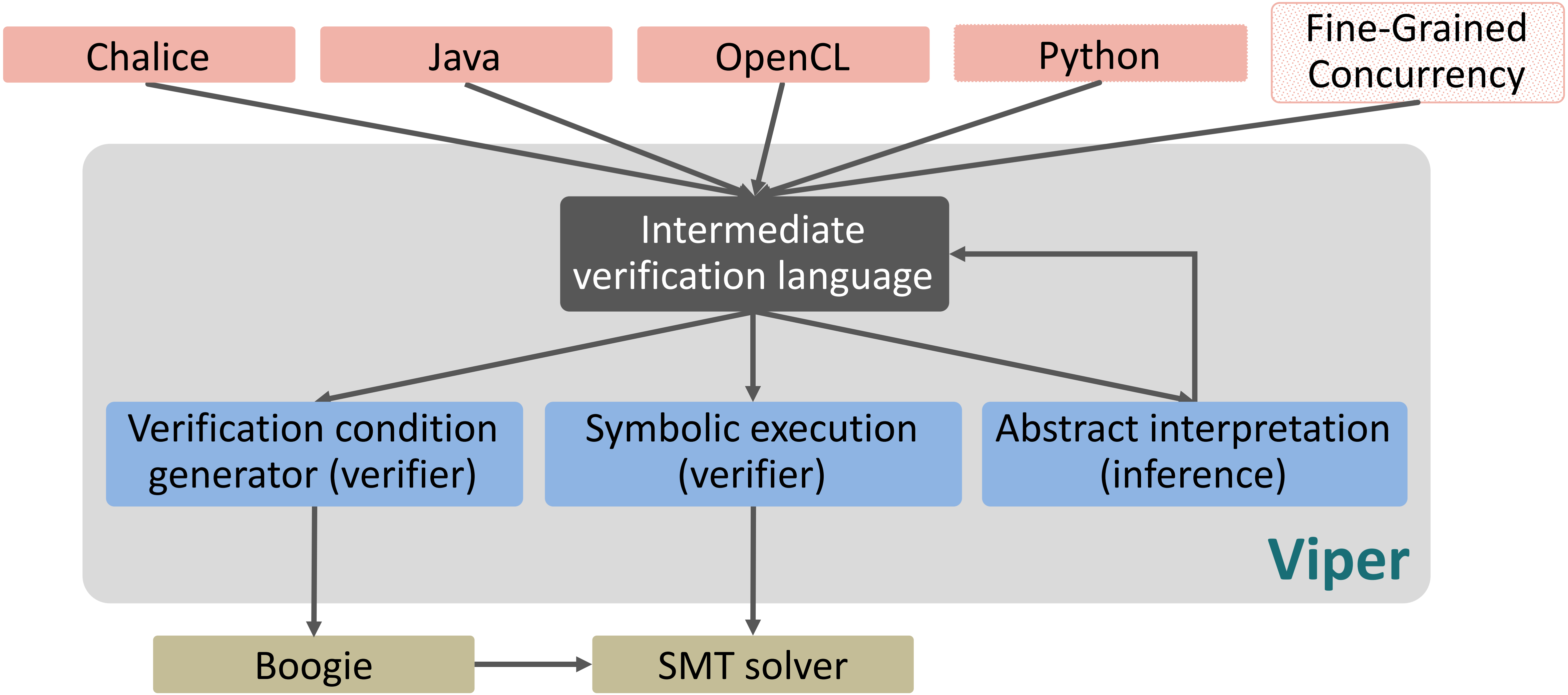
Dynamic Frames vs. Permissions



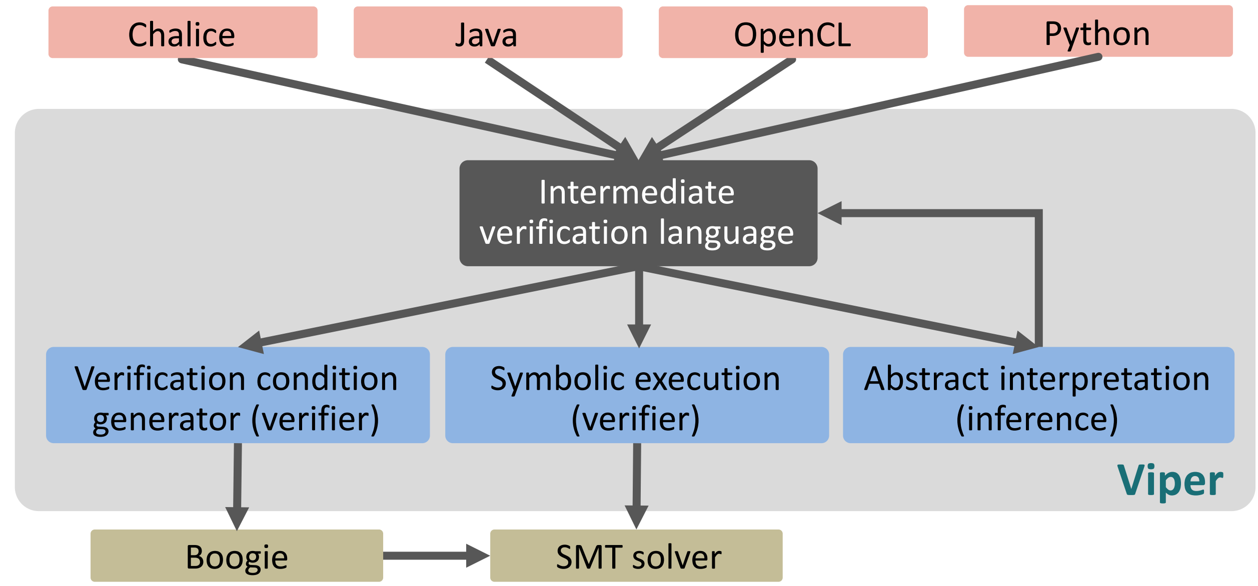
Viper: Currently



Viper: Next



Chalice
Java
OpenCL
Python
Fine-Grained Concurrency
Intermediate verification language
Verification condition generator (verifier)
Symbolic execution (verifier)
Abstract interpretation (inference)
Viper
Boogie
SMT solver



VIPER

http://viper.ethz.ch

$C_1 \parallel C_2$ +