Modular Program Verification

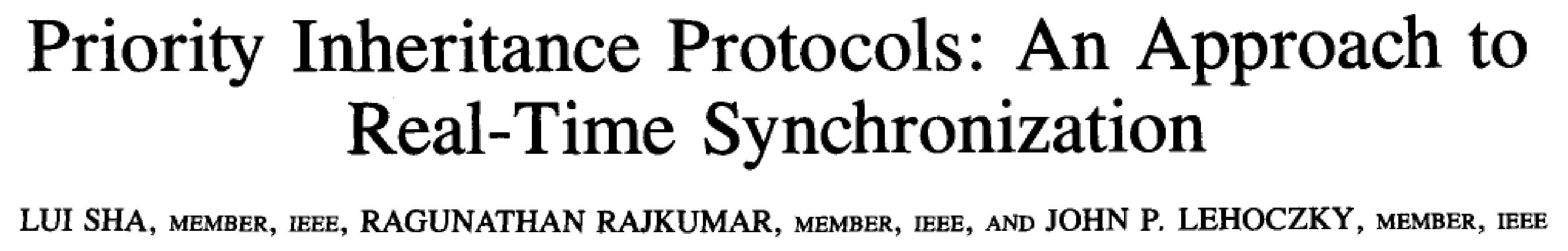
Peter Müller

OpenJDK's java.utils.Collection.sort() is broken:
The good, the bad and the worst case*

Stijn de Gouw[1,2], Jurriaan Rot[3,1], Frank S. de Boer[1,3], Richard Bubel[4], and
Reiner Hähnle[4]

- TimSort is the default sorting algorithm for Collections in Sun's JDK, OpenJDK, and Android SDK
- Certain large arrays (>= 67M) lead to index-out-of-bounds errors
- Bug was detected during a verification attempt
- Previous attempts to fix related errors were ineffective

Testing and code reviews are not sufficient to detect certain bugs

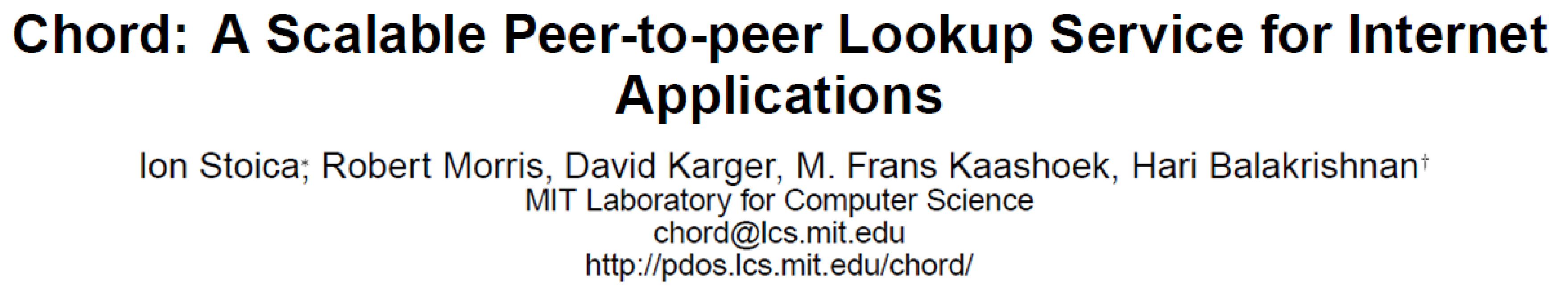


Priority Inheritance Protocols: An Approach to Real-Time Synchronization

LUI SHA, MEMBER, IEEE, RAGUNATHAN RAJKUMAR, MEMBER, IEEE, AND JOHN P. LEHOCZKY, MEMBER, IEEE

- Real-time operating systems use priority inheritance protocol to ensure that low-priority processes do not block high-priority processes
- Several operating systems implement the protocol incorrectly, leading to deadlocks and priority inversion

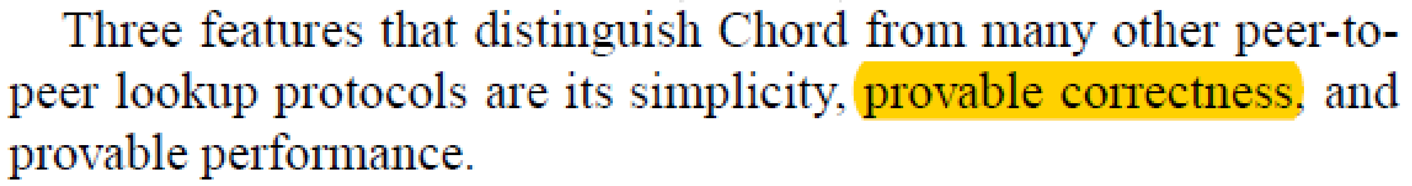
Code-level verification should complement reasoning on the algorithm/design level



Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica[*], Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan[†]
MIT Laboratory for Computer Science
chord@lcs.mit.edu
http://pdos.lcs.mit.edu/chord/

- Chord is a distributed hash table developed at MIT



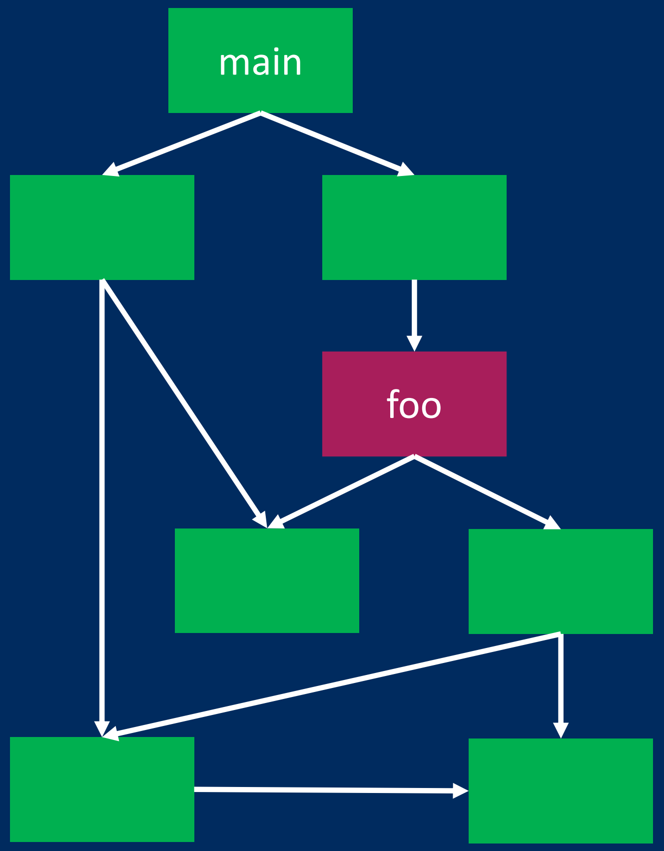
Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance.

- None of the seven properties claimed invariant of the original version is actually an invariant

Reasoning must be supported by tools

Modular Verification

- Verify each method separately
  - Scalability
- Do not use the implementation of callees
  - Software evolution
  - Dynamic method binding
- Do not use the implementation of callers and other methods
  - Correctness guarantees for libraries
  - Software evolution

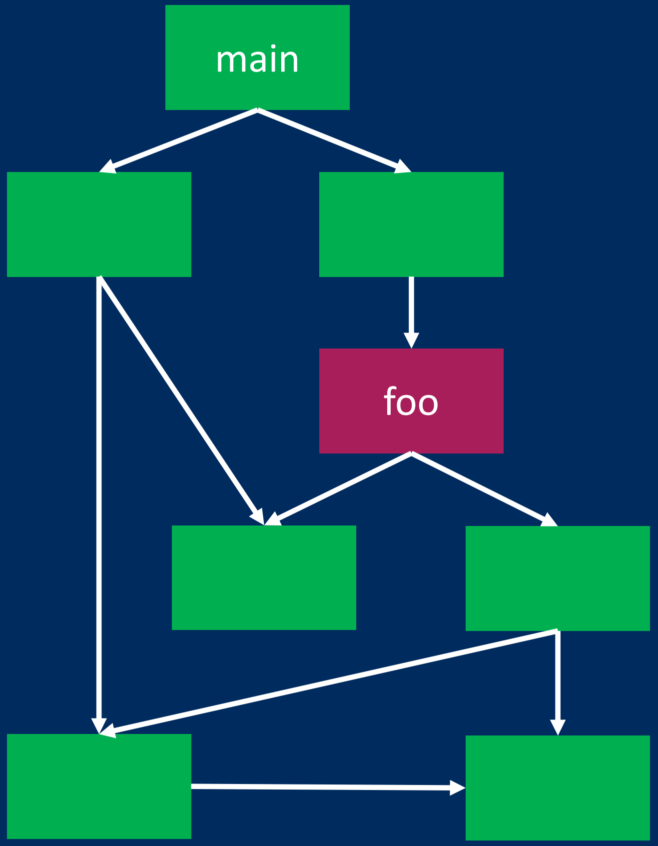


Outline

- Permission-based Verification
- The Viper Intermediate Language
- Building Verifiers
- Encoding of Advanced Verification Techniques

Contracts

- Contracts specify the intended behavior of parts of the program
- For the verification of a method, use the contracts of the rest of the program, not the implementation
- Verify calls in terms of method pre- and postconditions



Example: Contracts

```
class Account {
  var bal: int

  method deposit(amount: int)
    requires 0 < amount
    ensures bal == old(bal) + amount
  { … }

  method withdraw(amount: int)
    requires 0 < amount && amount <= bal
    ensures bal == old(bal) - amount
  { … }
}
```

```
method demo(a: Account)
  requires 0 <= a.bal
{
  a.deposit(200);
  a.withdraw(100);
}
```

Example: Side Effects

```
class Account {
  var bal: int

  method deposit(amount: int)
    requires 0 < amount
    ensures bal == old(bal) + amount
  { … }
}
```

```
method demo(a: Account, l: List)
  ensures l.len == old(l.len)
{
  a.deposit(200)
}
```

Example: Side Effects

```
class Account {
  var bal: int
  var transactions: List

  method deposit(amount: int)
    requires 0 < amount
    ensures bal == old(bal) + amount
  { transactions.add(amount) … }

  method getTransactions() returns (t: List)
  { t := transactions }
}
```

```
method demo(a: Account, l: List)
  ensures l.len == old(l.len)
{
  a.deposit(200)
}
```

```
demo(a, a.getTransactions())
```

The Frame Problem

$$\cancel{\dfrac{\{\,P\,\}\ S\ \{\,Q\,\} \qquad FV(S) \cap FV(R) = \{\,\}}{\{\,P \wedge R\,\}\ S\ \{\,Q \wedge R\,\}}}$$

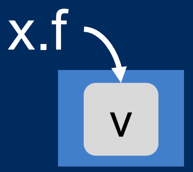
Footprints

$$\frac{\begin{array}{c}\{\,P\,\}\ S\ \{\,Q\,\} \\ \text{footprint}(\,S\,) \cap \text{footprint}(\,R\,) = \{\,\}\end{array}}{\{\,P \wedge R\,\}\ S\ \{\,Q \wedge R\,\}}$$

Separation Logic

- Heap properties are specified via points-to assertions

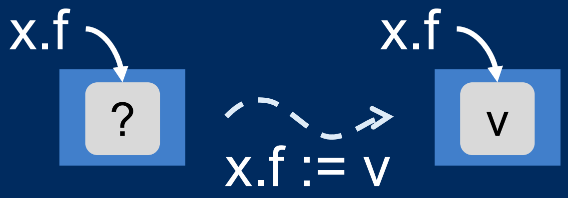
$$x.f \rightarrow v$$



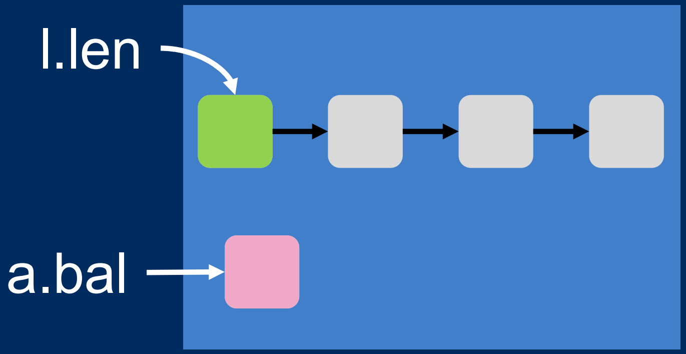
  - Holds in a partial heap that maps the memory location x.f to value v

- Each heap access to x.f requires the current partial heap to contain x.f

$$\{ x.f \rightarrow \_ \} \;\; x.f := v \;\; \{ x.f \rightarrow v \}$$



Footprints in Separation Logic



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

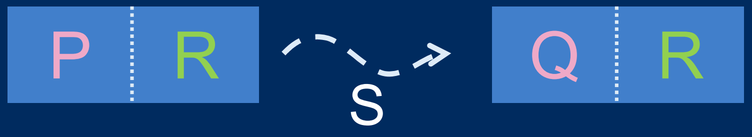
Separation and Framing

- Composition of partial heaps is described using separating conjunction

$$P * R$$

  - Holds in a partial heap if it can be split into two disjoint partial heaps, in which P and Q hold
  - $x.f \rightarrow \_ * x.f \rightarrow \_$ is equivalent to false
  - $x.f \rightarrow \_ * y.f \rightarrow \_$ implies $x \neq y$
- Frame rule

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



Example: Framing

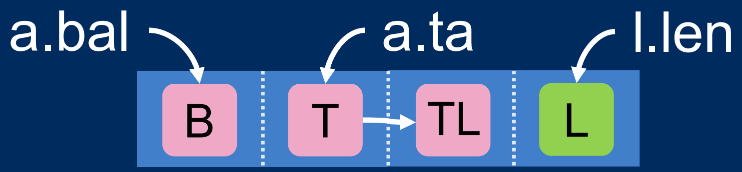
```
class Account {
  var bal: int

  method deposit(amount: int)
    requires this.bal → B ∗ 0 < amount
    ensures this.bal → B + amount
  { … }
}
```

```
method demo(a: Account, l: List)
  requires a.bal → B ∗ l.len → L
  ensures l.len → L
{ a.deposit(200) }
```

Example: Framing

```
class Account {
  var bal: int
  var ta: List

  method deposit(amount: int)
    requires this.bal → B ∗ 0 < amount ∗
             this.ta → T ∗ this.ta.len → TL
    ensures …
  { ta.add(amount) … }
}
```

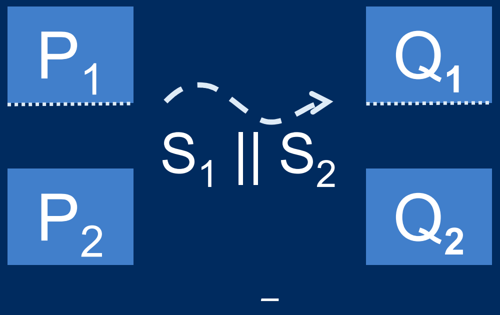
```
method demo(a: Account, l: List)
  requires a.bal → B ∗ l.len → L ∗
           a.ta → T ∗ a.ta.len → TL
  ensures l.len → L
{ a.deposit(200) }
```



Recall $x.f \rightarrow \_ \ast y.f \rightarrow \_$ implies $x \neq y$

demo(a, a.getTransactions())

Parallel Composition



P_1 Q_1

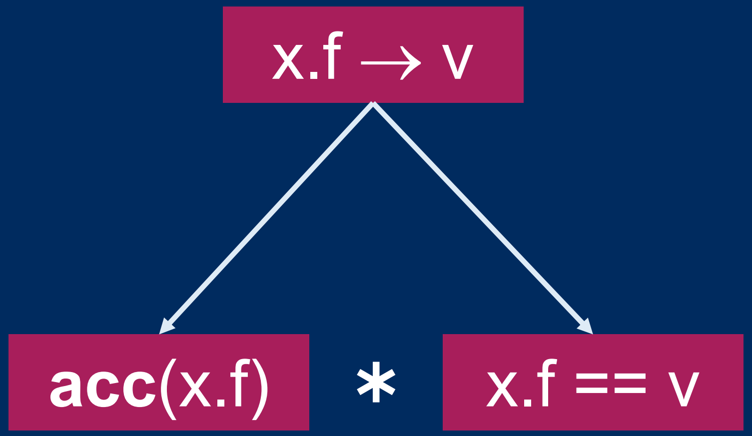
$S_1 \parallel S_2$

P_2 Q_2

Disjointness of footprints ensures data-race freedom

$$\frac{\{\, P_1 \,\}\ S_1\ \{\, Q_1 \,\} \qquad \{\, P_2 \,\}\ S_2\ \{\, Q_2 \,\}}{\{\, P_1 \ast P_2 \,\}\ S_1 \parallel S_2\ \{\, Q_1 \ast Q_2 \,\}}$$

Implicit Dynamic Frames



x.f → v

acc(x.f) ∗ x.f == v

```
method demo(a: Account, l: List)
  requires a.bal → B ∗ l.len → L
  ensures l.len → L
{ a.deposit(200) }
```

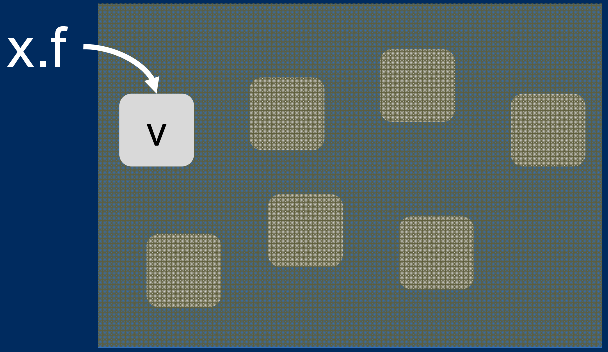
```
method demo(a: Account, l: List)
  requires acc(a.bal) ∗ acc(l.len)
  ensures acc(l.len) ∗ l.len == old(l.len)
{ a.deposit(200) }
```

- Assertions must be self-framing

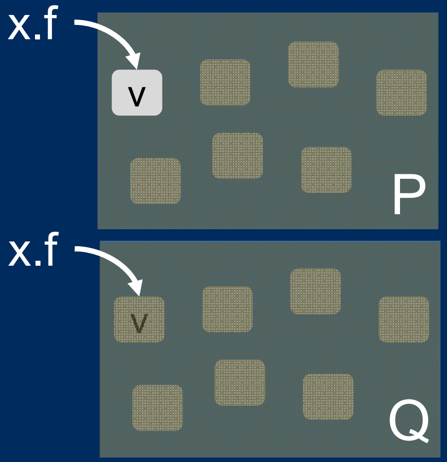
Total-Heap Semantics

- Partial-heap semantics is not suitable

acc(x.f) ∗ x.f == v



- Define semantics relative to a total heap and permission mask
  - x.f == v holds if heap(x,f) yields v
  - **acc**(x.f) holds if permission mask(x,f) yields true
  - P ∗ Q holds if the mask can be split into two compatible masks, in which P and Q hold

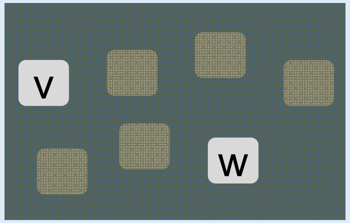


Ownership Transfer

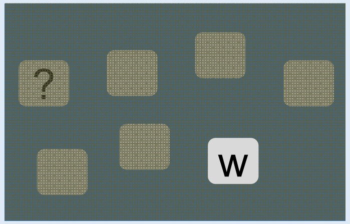
```
class Account {
  var bal: int

  method deposit(amount: int)
    requires acc(this.bal) ∗ …
    ensures acc(this.bal) ∗ …
  { … }
}
```

v’

```
method demo(a: Account, l: List)
  requires acc(a.bal) ∗ acc(l.len)
  ensures acc(l.len) ∗ l.len == old(l.len)
{
```



```
  a.deposit(200)
```



```
}
```

Fractional Permissions

- Permissions can be split and recombined

acc(x.f, 1/2)

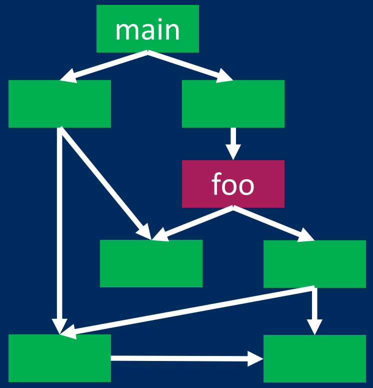
  - Read access requires a non-zero permission
  - Write access requires full permission

- Separating conjunction adds permissions

acc(x.f, 1/2) $\ast$ **acc**(x.f, 1/2) $\equiv$ **acc**(x.f)

Summary

- Modularity is important for scalability, components, and evolution



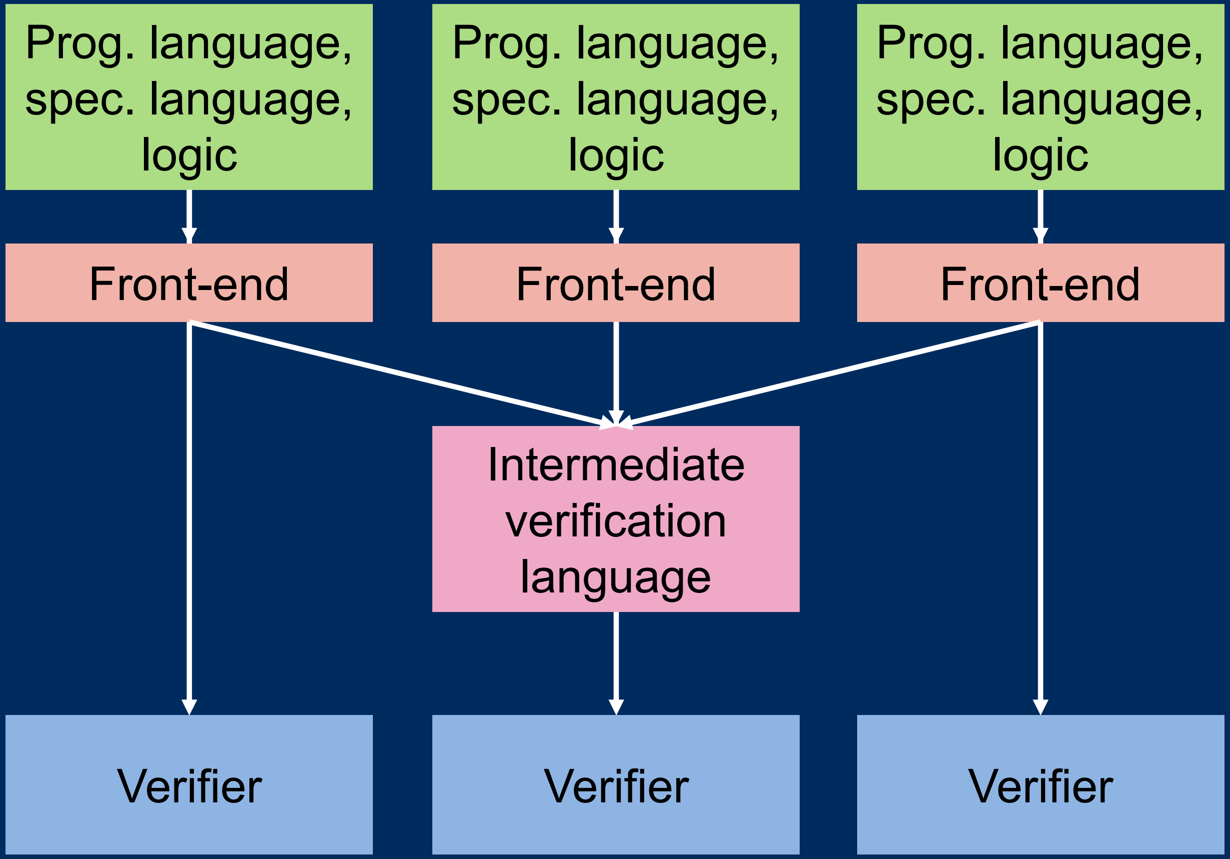
- Contracts enable modular verification
- Permissions
  - provide a solution to the frame problem
  - enable the verification of concurrent programs

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

$$\frac{\{\,P_1\,\}\ S_1\ \{\,Q_1\,\} \qquad \{\,P_2\,\}\ S_2\ \{\,Q_2\,\}}{\{\,P_1 * P_2\,\}\ S_1 \,||\, S_2\ \{\,Q_1 * Q_2\,\}}$$

Chalice

- Home page: www.pm.inf.ethz.ch/research/chalice.html
- Try online: www.rise4fun.com/Chalice
- Download: chalice.codeplex.com



Prog. language, spec. language, logic

Prog. language, spec. language, logic

Prog. language, spec. language, logic

Front-end

Front-end

Front-end

Intermediate verification language

Verifier

Verifier

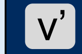
Verifier

Outline

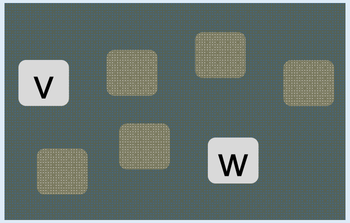
- Permission-based Verification
- The Viper Intermediate Language
- Building Verifiers
- Encoding of Advanced Verification Techniques

Ownership Transfer

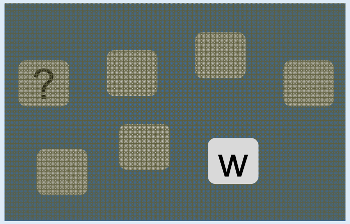
```
class Account {
  var bal: int

  method deposit(amount: int)
    requires acc(this.bal) ∗ …
    ensures acc(this.bal) ∗ …
  { … }
}
```



```
method demo(a: Account, l: List)
  requires acc(a.bal) ∗ acc(l.len)
  ensures acc(l.len) ∗ l.len == old(l.len)
{
```

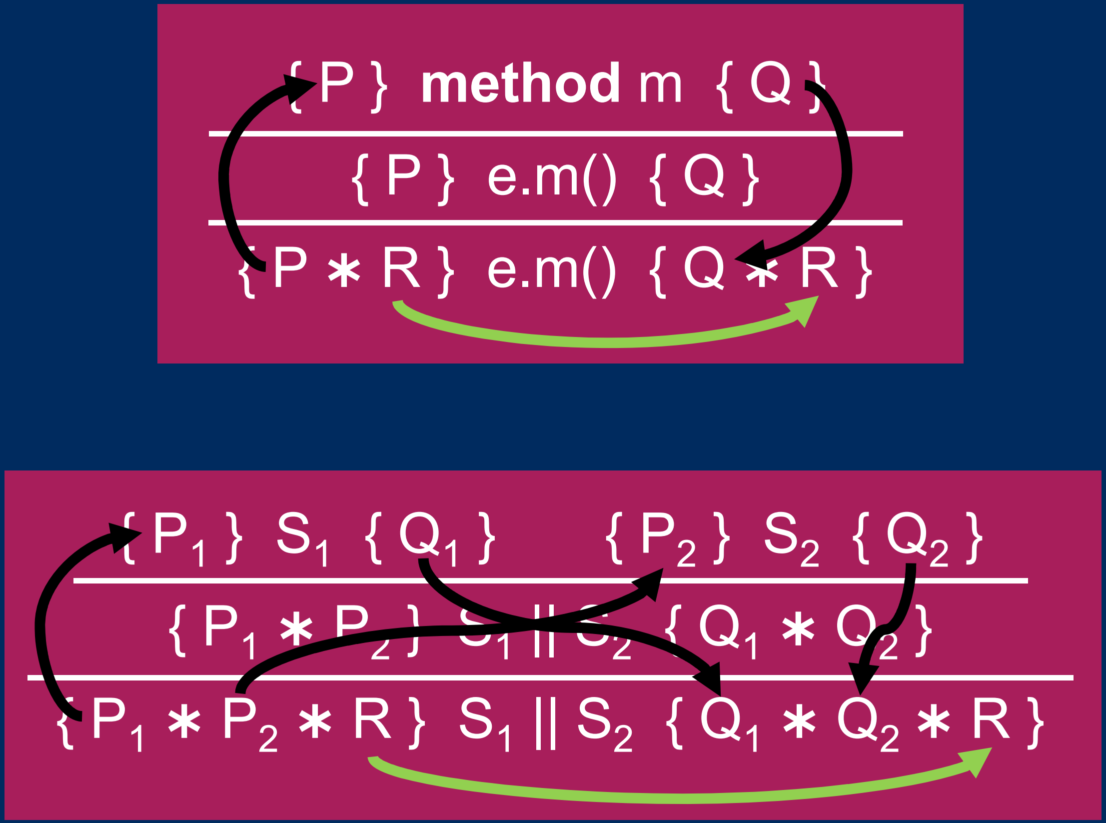


```
  a.deposit(200)
```



```
}
```

Ownership Transfer



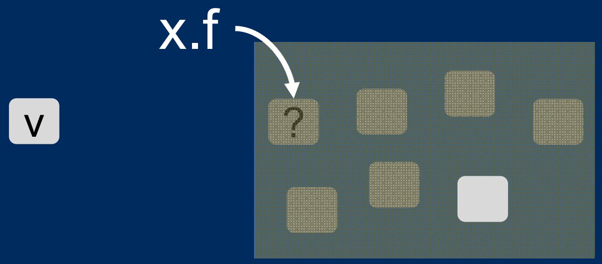
$$\cfrac{\cfrac{\{\,P\,\}\ \ \textbf{method}\ m\ \ \{\,Q\,\}}{\{\,P\,\}\ \ e.m()\ \ \{\,Q\,\}}}{\{\,P \ast R\,\}\ \ e.m()\ \ \{\,Q \ast R\,\}}$$

$$\cfrac{\cfrac{\{\,P_1\,\}\ S_1\ \{\,Q_1\,\} \qquad \{\,P_2\,\}\ S_2\ \{\,Q_2\,\}}{\{\,P_1 \ast P_2\,\}\ S_1 \,\|\, S_2\ \{\,Q_1 \ast Q_2\,\}}}{\{\,P_1 \ast P_2 \ast R\,\}\ S_1 \,\|\, S_2\ \{\,Q_1 \ast Q_2 \ast R\,\}}$$

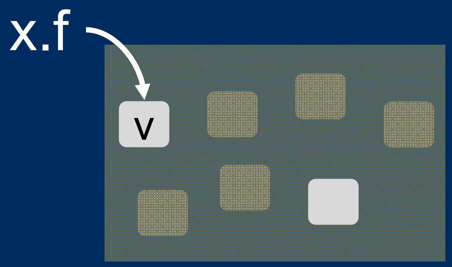
Inhale and Exhale

- **inhale** A means:
  - obtain all permissions required by A
  - assume all logical constraints
- **exhale** A means:
  - assert all logical constraints
  - check and remove all permissions required by A
  - havoc any locations to which all permission is lost
- Analogues of **assume** and **assert**

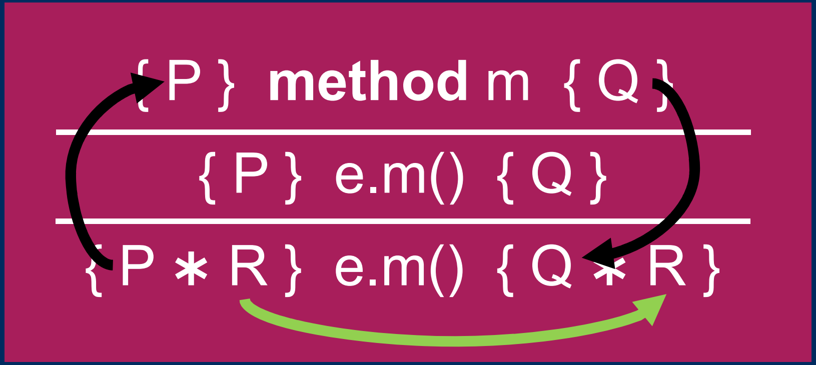
inhale acc(x.f) ∗ x.f == v



exhale acc(x.f) ∗ x.f == v

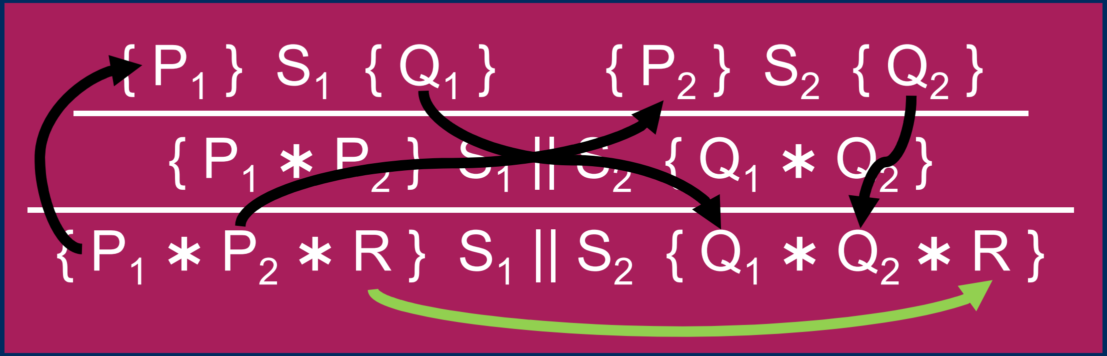


Encoding Ownership Transfer



$$\begin{array}{c} \{P\}\ \textbf{method}\ m\ \{Q\} \\ \hline \{P\}\ e.m()\ \{Q\} \\ \hline \{P * R\}\ e.m()\ \{Q * R\} \end{array}$$

exhale P
inhale Q



$$\begin{array}{c} \{P_1\}\ S_1\ \{Q_1\} \qquad \{P_2\}\ S_2\ \{Q_2\} \\ \hline \{P_1 * P_2\}\ S_1 \,||\, S_2\ \{Q_1 * Q_2\} \\ \hline \{P_1 * P_2 * R\}\ S_1 \,||\, S_2\ \{Q_1 * Q_2 * R\} \end{array}$$

exhale P_1
exhale P_2
inhale Q_1
inhale Q_2

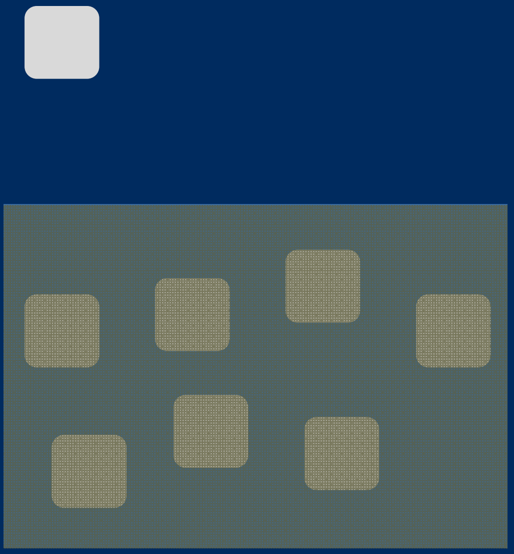
```
method demo(a: Account, l: List)
  requires acc(a.bal) ∗ acc(l.len)
  ensures acc(l.len) ∗ l.len == old(l.len)
{
  a.deposit(200)
}
```

```
method deposit(amount: int)
  requires acc(this.bal)
  ensures acc(this.bal)
```

```
inhale acc(a.bal) ∗ acc(l.len)
exhale acc(a.bal)
inhale acc(a.bal)
exhale acc(l.len) ∗ l.len == old(l.len)
```

Encoding Monitors

```
class Account {
  var bal: int

  invariant acc(this.bal)

  method deposit(amount: int)
  {
    acquire this
    this.bal := this.bal + amount
    release this
  }
}
```



```
inhale acc(this.bal)
this.bal := this.bal + amount
exhale acc(this.bal)
```

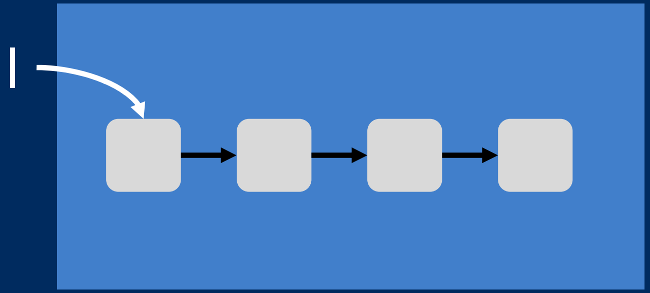
Abstraction

```
method demo(a: Account, l: List)
  requires acc(a.bal) ∗ acc(l.len)
  ensures acc(l.len) ∗ l.len == old(l.len)
{
  a.deposit(200)
}
```

Mentioning field names in contracts:

- Violates information hiding
- Cannot express access to a statically-unknown set of locations

Recursive Predicates



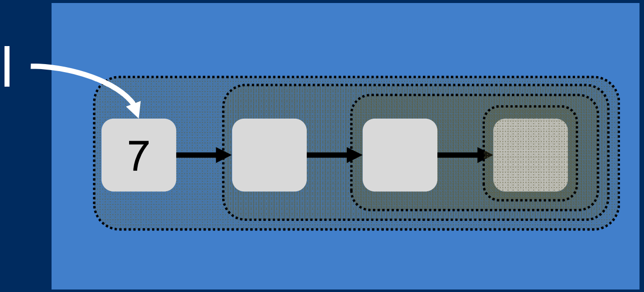
```
predicate list(this: Ref) {
  acc(this.next) ∗ acc(this.data) ∗
  (this.next != null ==> list(this.next))
}
```

- Predicate instances are manipulated similarly to access permissions
  - Ownership is transferred via inhale and exhale
  - But P(x) $*$ P(x) is not equivalent to false

inhale list(a)
exhale list(a)

predicate P(this: **Ref**)
{ **acc**(this.f, 1/3) }

- Folding and unfolding is done manually



unfold list(l)
l.data := 7
fold list(l)

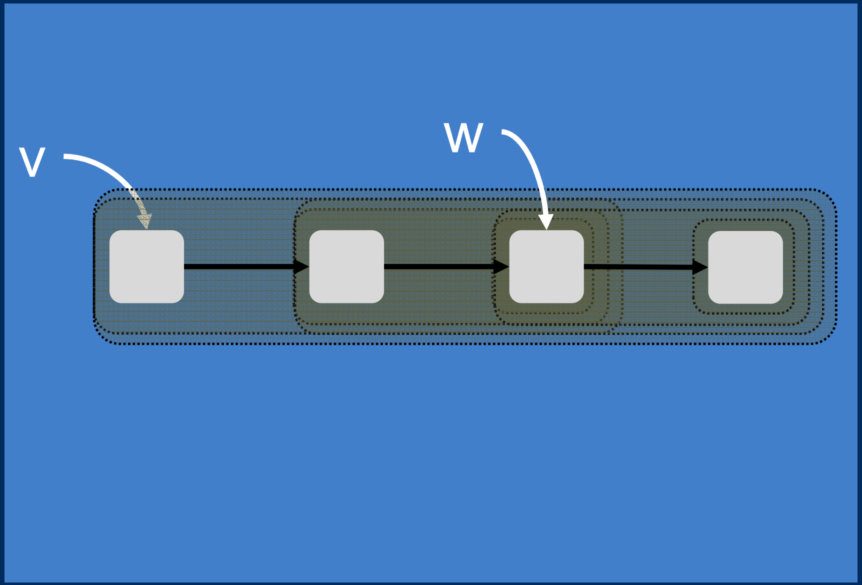
Examples: Recursive Predicates

```
predicate lseg(this: Ref, last: Ref) {
  this != last ==>
       acc(this.next) ∗ acc(this.data) ∗
       lseg(this.next, last)
}
```

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

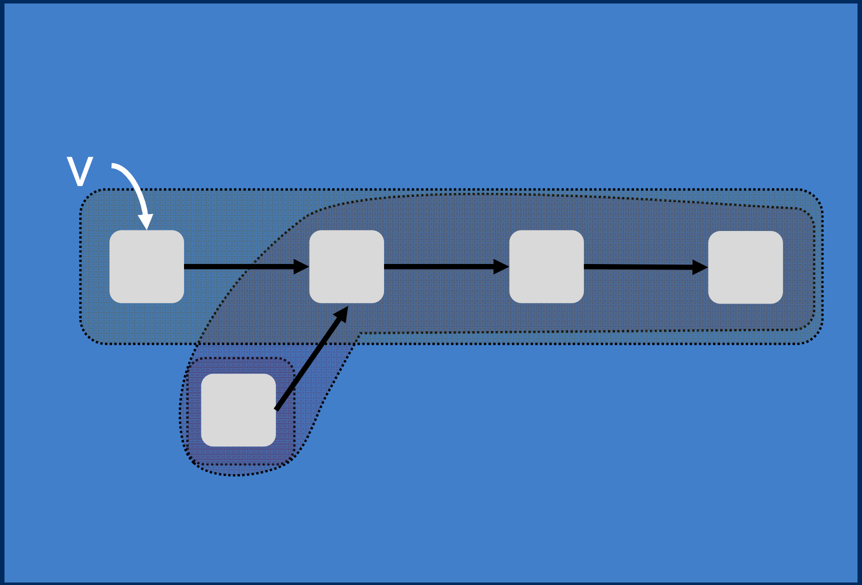
```
predicate list(this: Ref) {
  acc(this.next) ∗ acc(this.data) ∗
  (this.next != null ==> list(this.next) ∗
       this.data <= unfolding list(this.next) in this.next.data)
}
```

Limitations



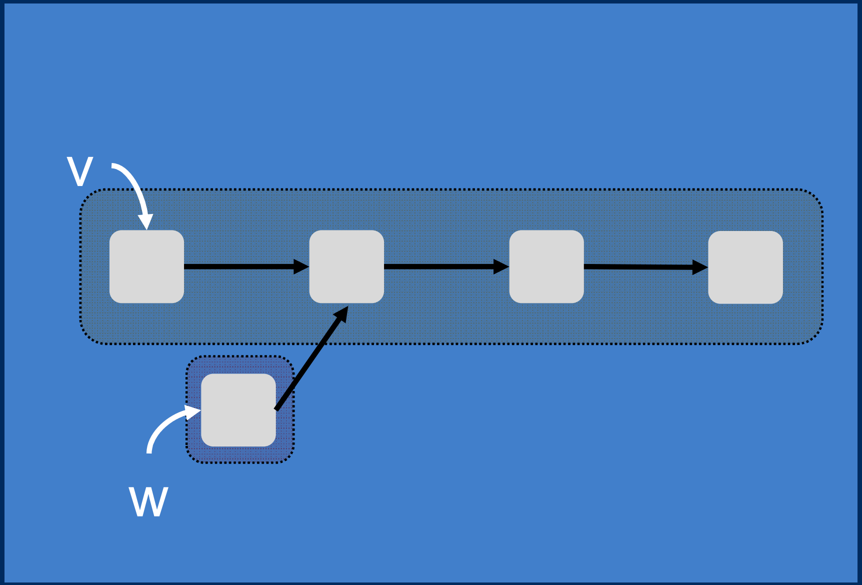
- Extending footprints

Limitations



- Extending footprints
- Sharing

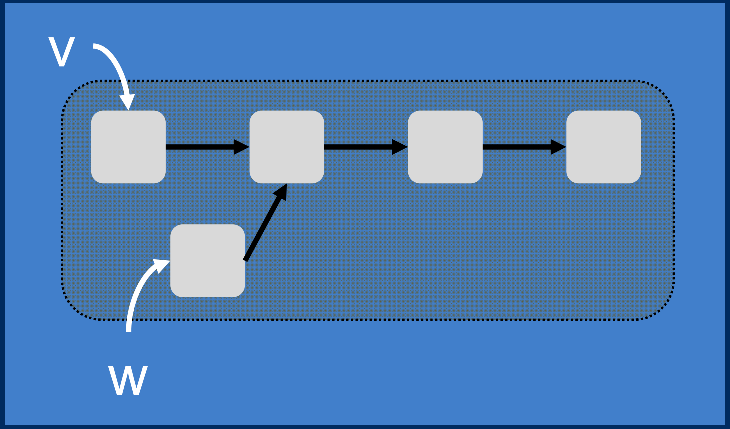
Limitations



- Extending footprints
- Sharing
- Traversal order

Quantified Permissions

```
predicate list( nodes: Set[ Ref ] ) {
  forall n: Ref :: n in nodes ==>
                              acc(n.next) ∗
                              n.next in nodes
}
```



```
list(nodes) ∗
v in nodes ∗ w in nodes
```

Abstraction

```
method demo(a: Account, l: List)
  requires acc(a.bal) ∗ acc(l.len)
  ensures acc(l.len) ∗ l.len == old(l.len)
{
  a.deposit(200)
}
```

Mentioning field names in contracts:

- Violates information hiding
- Cannot express access to a statically-unknown set of locations

Heap-Dependent Functions

- Predicates abstract over permissions

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

- Functions abstract over expressions

```
function length(this: Ref): Int
  requires list(this)
{
  unfolding list(this) in
  this.next == null ? 1 : 1 + length(this.next)
}
```

Abstraction

```
method demo(a: Account, l: List)
  requires account(a) ∗ list(l)
  ensures list(l) ∗ length(l) == old(length(l))
{
  a.deposit(200)
}
```

- Predicates and functions need not have definitions

Function Framing

- Heap-dependent functions are mathematical functions of their arguments and their footprint

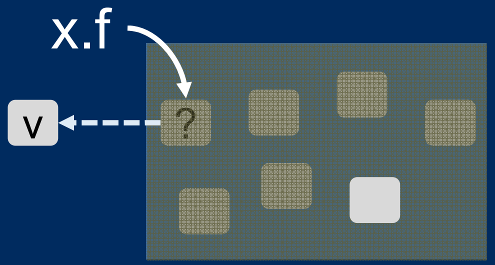
```
function length(this: Ref): Int
  requires list(this)
{
  unfolding list(this) in
  this.next == null ? 1 : 1 + length(this.next)
}
```

Summary

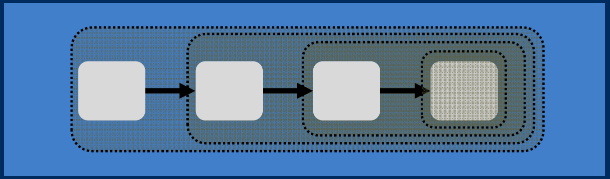
- Intermediate verification languages facilitate the development of program verifiers

- Inhale and exhale primitives express ownership transfer

exhale acc(x.f) ∗ x.f == v



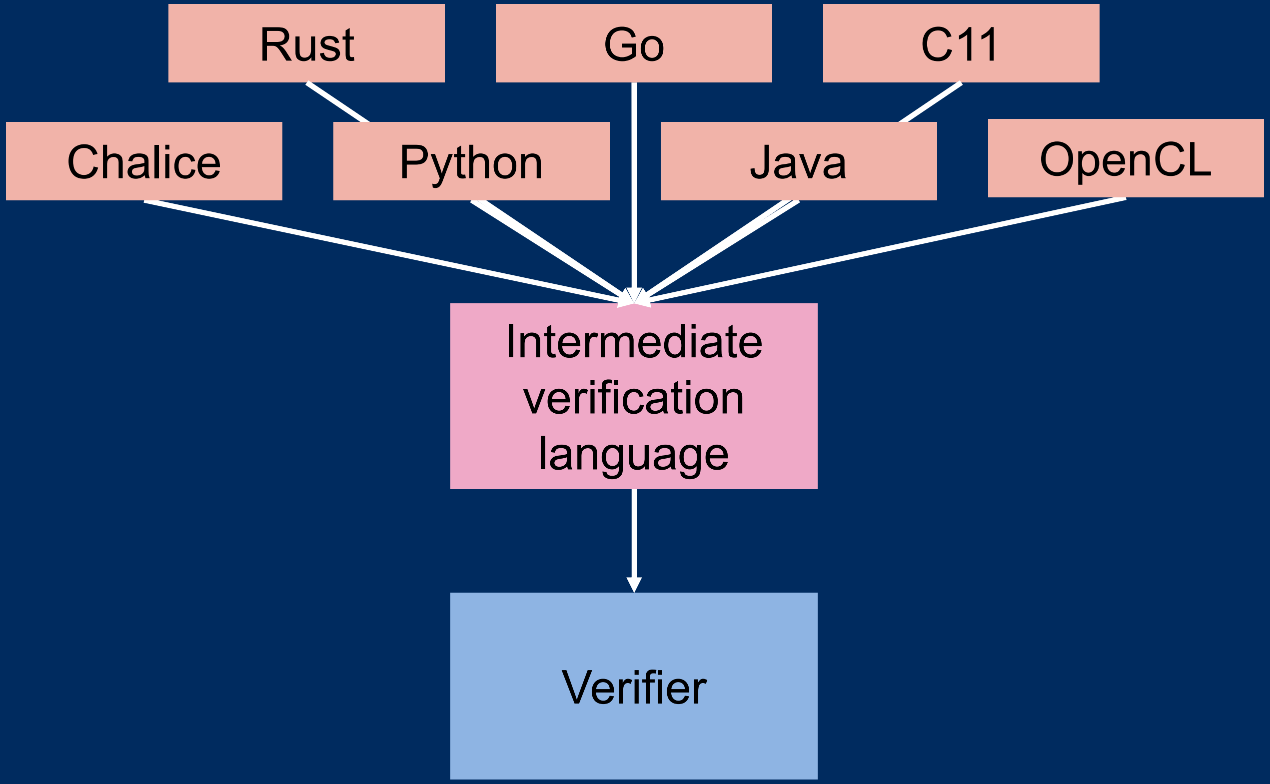
- Predicates and functions abstract over permissions and values



function length(this: **Ref**): **Int**
  requires list(this)

Viper

- Home page: viper.ethz.ch
- Try online: viper.ethz.ch/examples
- Download: bitbucket.org/viperproject
- Download a version for the tutorial on Friday



Rust

Go

C11

Chalice

Python

Java

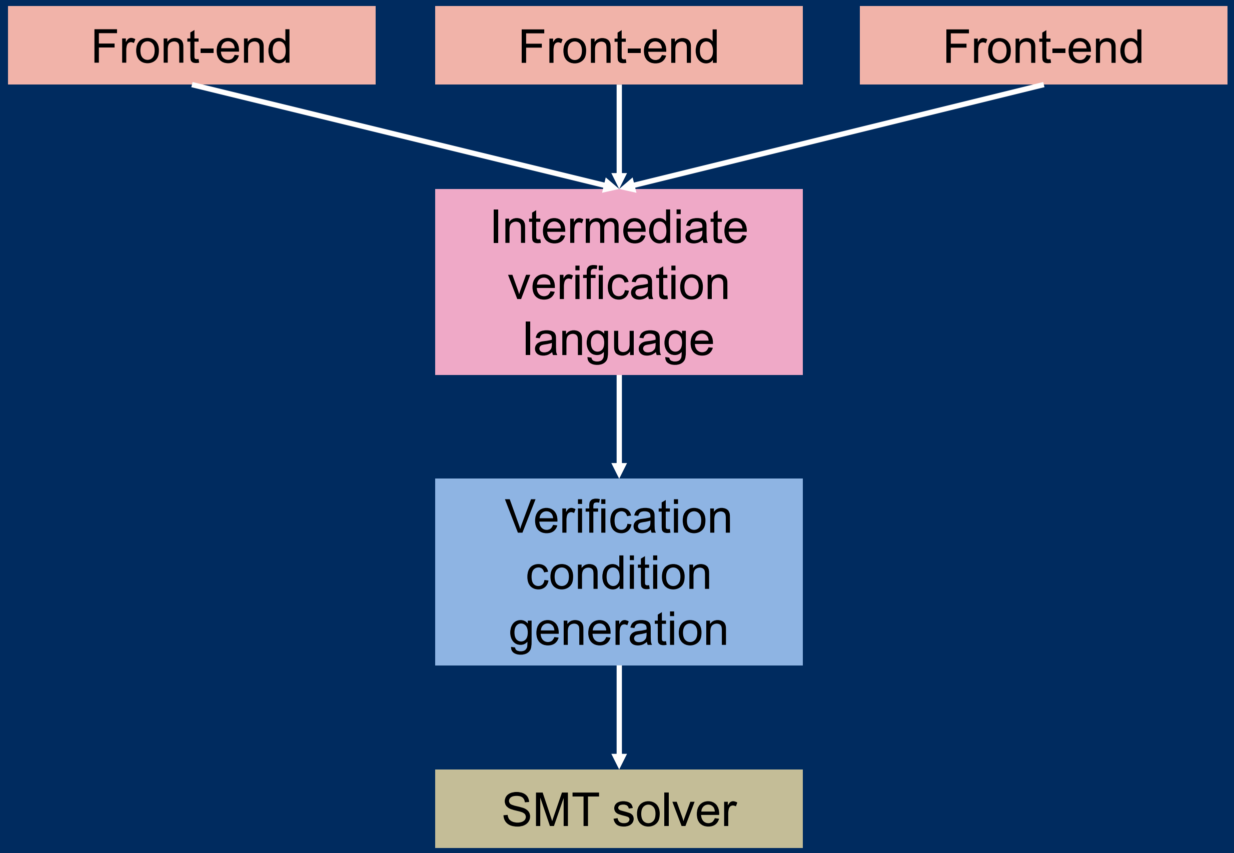
OpenCL

Intermediate verification language

Verifier

Outline

- Permission-based Verification
- The Viper Intermediate Language
- Building Verifiers
- Encoding of Advanced Verification Techniques



Front-end

Front-end

Front-end

Intermediate verification language

Verification condition generation

SMT solver

Verification Condition Generation

- For a given program, verification condition generation computes a logical formula whose validity implies the correctness of the program

- Verification condition reflects semantics of the program and its specification

- We will compute verification conditions in two steps:
  - Encode the Viper program into guarded commands
  - Compute weakest preconditions of guarded-commands programs

Guarded Commands

S ::= x := E
| S_1; S_2
| $S_1 \;\square\; S_2$
| **assert** P
| **assume** P
| **havoc** x

- Assertions P are first-order logic formulas
  - Including quantifiers, uninterpreted functions, arithmetic, etc.

Encoding into Guarded Commands

```
method abs(a: Int) returns (res: Int)
  ensures  0 <= res
{
  if(0 <= a)   { res := a }
  else         { res := -a }
}
```

```
 (
  assume 0 <= a
  res := a
▯
  assume a < 0
  res := -a
)
assert 0 <= res
```

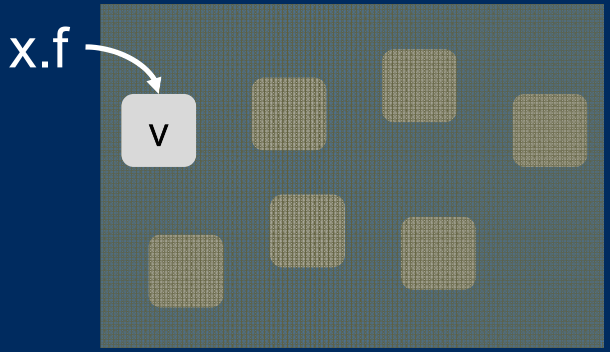
Weakest Preconditions

$$
\begin{aligned}
wp(x := E, Q) &\equiv Q[E/x] \\
wp(S_1; S_2, Q) &\equiv wp(S_1, wp(S_2, Q)) \\
wp(S_1 \, \square \, S_2, Q) &\equiv wp(S_1, Q) \wedge wp(S_2, Q) \\
wp(\textbf{assert } P, Q) &\equiv P \wedge Q \\
wp(\textbf{assume } P, Q) &\equiv P \Rightarrow Q \\
wp(\textbf{havoc } x, Q) &\equiv \forall x \bullet Q
\end{aligned}
$$

To verify statement S, prove that wp(S, true) holds

Heap Model

- Define semantics relative to a total heap and permission mask
  - x.f == v holds if heap(x,f) yields v
  - **acc**(x.f) holds if permission mask(x,f) yields true



- Model heap as total map

Ref × Field<T> → T

- Model permission mask as total map

Ref × Field<T> → Bool

Encoding inhale

$$\begin{aligned}
\langle \textbf{inhale}\ e\rangle &\equiv \textbf{assume}\ \langle e\rangle\\
\langle \textbf{inhale}\ \textbf{acc}(e.f)\rangle &\equiv \textbf{assume}\ \langle e\rangle \neq \text{null}\\
&\ \textbf{assume}\ \neg\text{mask}(\langle e\rangle, f)\\
&\ \text{mask}(\langle e\rangle, f) := \textbf{true}\\
\langle \textbf{inhale}\ a_1 \ast a_2\rangle &\equiv \langle \textbf{inhale}\ a_1\rangle; \langle \textbf{inhale}\ a_2\rangle
\end{aligned}$$

- ⟨ _ ⟩ is the encoding function
- a is an assertion
- e is an expression (not containing permissions)

```
inhale acc(x.f) ∗ x.f == v
```

```
assume x ≠ null
assume ¬mask(x,f)
mask(x,f) := true

assume heap(x,f) == v
```

```
inhale acc(x.f) ∗ acc(y.f)
```

```
assume x ≠ null
assume ¬mask(x,f)
mask(x,f) := true

assume y ≠ null
assume ¬mask(y,f)
mask(y,f) := true
```

Encoding exhale

⟨⟨**exhale** e⟩⟩ ≡ **assert** ⟨e⟩

⟨⟨**exhale acc**(e.f)⟩⟩ ≡ **assert** mask(⟨e⟩, f)
mask(⟨e⟩, f) := **false**

⟨⟨**exhale** a_1 ∗ a_2⟩⟩ ≡ ⟨⟨**exhale** a_1⟩⟩; ⟨⟨**exhale** a_2⟩⟩

⟨**exhale** a⟩ ≡ ⟨⟨**exhale** a⟩⟩
h := heap
havoc heap
assume ∀x,f • mask(x,f) ⇒ heap(x,f) == h(x,f)

```
exhale acc(x.f) ∗ x.f == v
```

```
assert mask(x,f)
mask(x,f) := false

assert heap(x,f) == v

// havoc heap(x,f)
```

```
exhale acc(x.f) ∗ acc(y.f)
```

```
assert mask(x,f)
mask(x,f) := false

assert mask(y,f)
mask(y,f) := false

// havoc heap(x,f), heap(y,f)
```

Predicates

- For each predicate, introduce a function that maps a predicate instance to a field name

predicate lseg(this: **Ref**, last: **Ref**)

lsegField: Ref × Ref → Field<**Int**>

- Use this field to index mask

inhale lseg(a, b)

mask(null, lsegField(a, b)) := **true**

Heap-Dependent Functions

- Encode Viper function as mathematical function

```
function balance(this: Ref): Int
  requires acc(this.bal)
```

balance: Ref × Int → **Int**

- Function applications evaluate footprint

```
balance(x)
```

balance(x, heap(x.bal))

Heap-Dependent Functions

- Encode Viper function as mathematical function

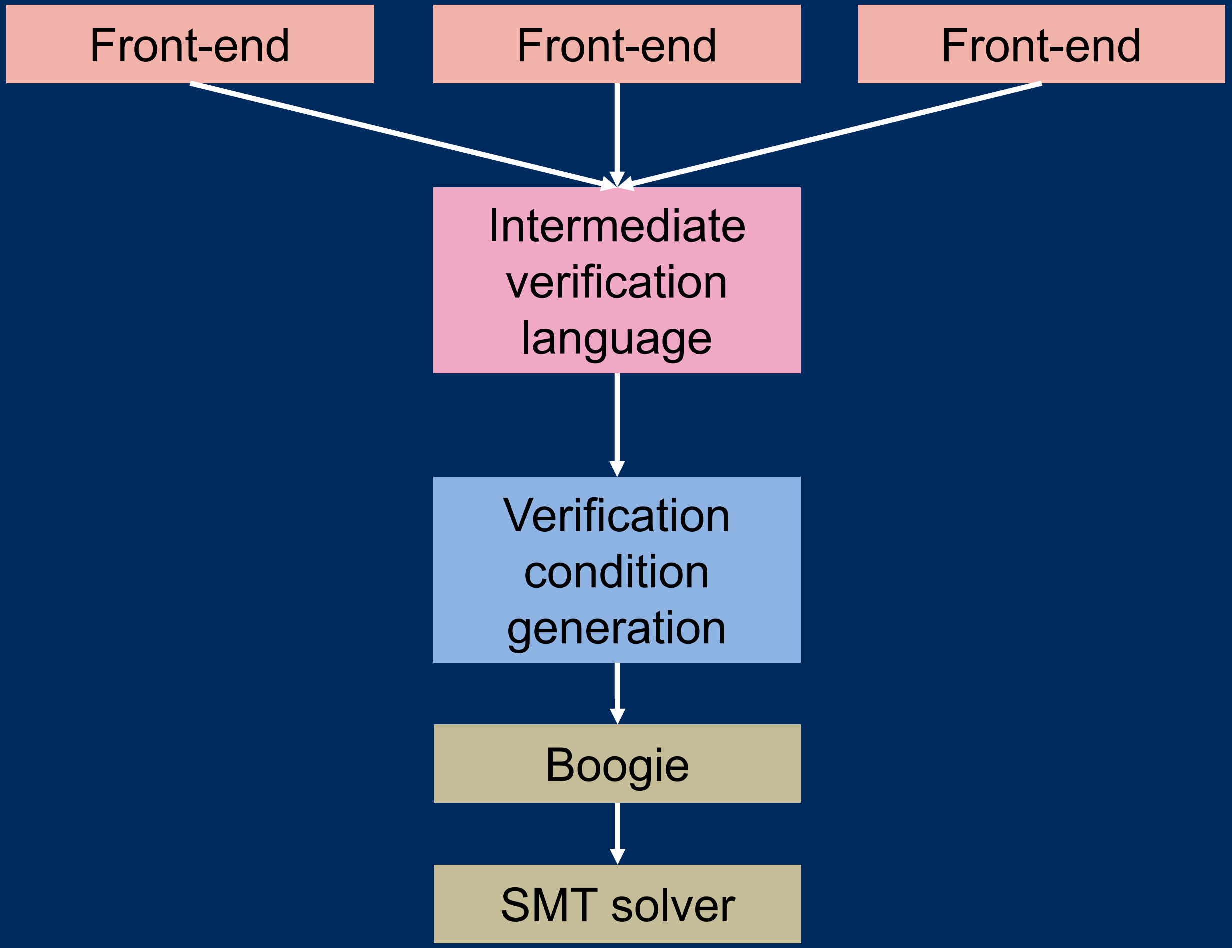
```
function length(this: Ref): Int
  requires list(this)
```

```
balance: Ref × Int→ Int
```

- Give predicate instances a version number
  - Changes each time the predicate’s footprint may change

```
length(x)
```

```
length(x, heap(null, listField(x)))
```



Front-end

Front-end

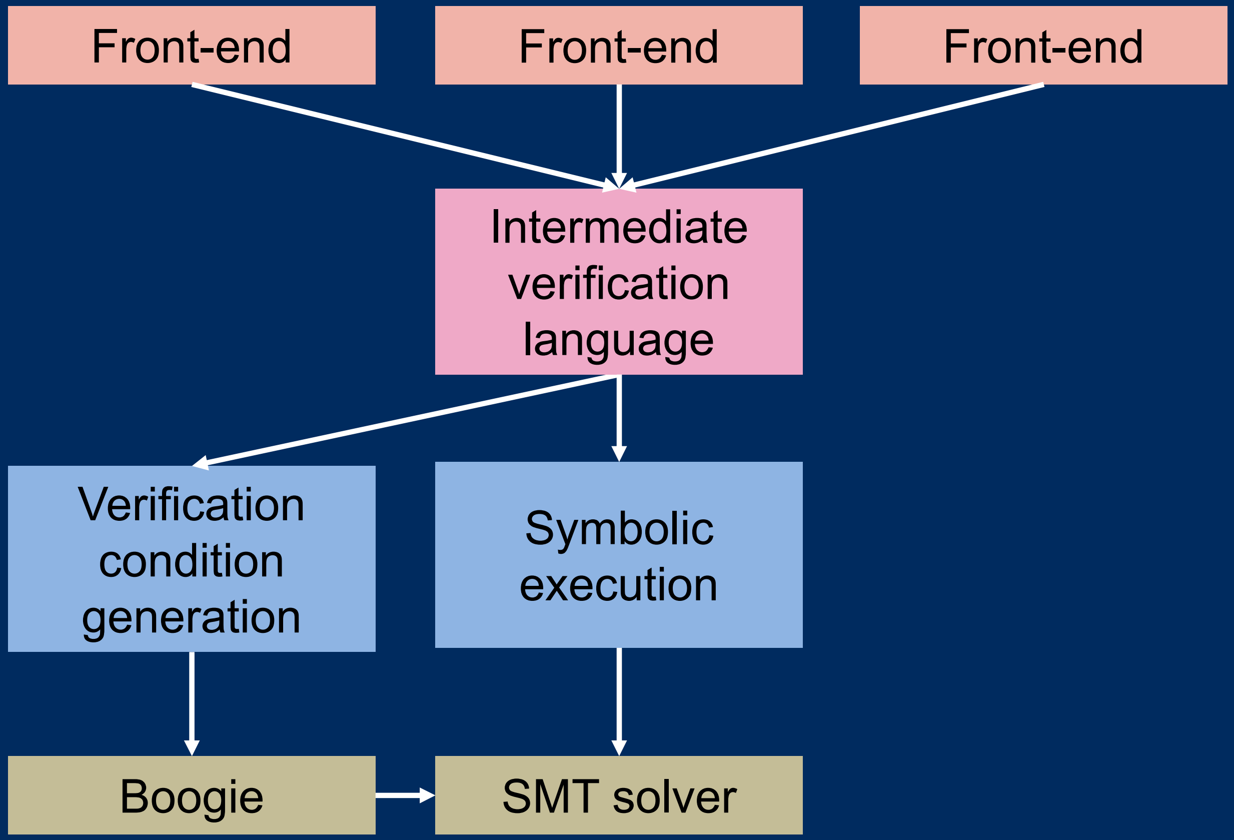
Front-end

Intermediate verification language

Verification condition generation

Boogie

SMT solver



Front-end

Front-end

Front-end

Intermediate verification language

Verification condition generation

Symbolic execution

Boogie

SMT solver

Symbolic Execution

- Symbolic execution simulates the execution of a program statically, using symbolic rather than concrete values
  - Introduce symbolic variables to represent inputs

- A configuration consists of
  - The statement to be executed
  - An environment, mapping program variables to expressions over the symbolic variables
  - A path condition, a logic formula representing information about the symbolic variables

```
method abs(a: Int) returns (res: Int)
  ensures  0 <= res
{
  if(0 <= a)  { res := a }
  else        { res := -a }
}
```

Statement	Environment		Path Condition
	a	res	
if(0 <= a) { res := a } **else** { res := -a } **assert** 0 <= res	A	R	true
res := a; **assert** 0 <= res	A	R	$0 \le A$
assert 0 <= res	A	A	$0 \le A$

check $0 \le A \Rightarrow 0 \le A$ ✓

Statement	Environment		Path Condition
	a	res	
res := -a; **assert** 0 <= res	A	R	$A < 0$
assert 0 <= res	A	-A	$A < 0$

check $A < 0 \Rightarrow 0 \le -A$ ✓

Algorithm

```
$\sigma$ := { (s;stop, $env_0$, true) }
while $\sigma \neq \{ \}$ do
  $\gamma$ := take($\sigma$)
  r := step($\gamma$)
  if r = $\perp$ then return failure
  $\sigma := \sigma \cup r$
end
return success
```

- s is the program to be verified
- stop is an artificial stop-marker
- env_0 maps each program variable to a fresh symbolic variable

Algorithm

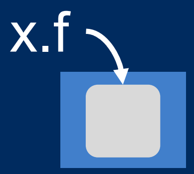
$$\begin{aligned}
&\text{step}(x := e;s,\ env,\ \pi) &&= \{\ (s,\ env[x := \langle e \rangle],\ \pi)\ \}\\
&\text{step}(\textbf{if}(e)\ \{\ s_1\ \}\ \textbf{else}\ \{\ s_2\ \};s,\ env,\ \pi) &&= \{\ (s_1;s,\ env,\ \pi \wedge \langle e \rangle),\ (s_2;s,\ env,\ \pi \wedge \neg\langle e \rangle)\ \}\\
&\text{step}(\textbf{assert}\ e;s,\ env,\ \pi) &&= \text{if check}(\pi \Rightarrow \langle e \rangle)\ \text{then}\ \{\ (s,\ env,\ \pi)\ \}\ \text{else}\ \bot\\
&\text{step}(\text{stop},\ env,\ \pi) &&= \{\ \}
\end{aligned}$$

- $\langle \_ \rangle$ symbolically evaluates an expression to a symbolic expression
- check is a query to the SMT solver

Heap Model

- The symbolic heap models the partial-heap semantics

acc(x.f)



- Model heap as set of heap chunks (E_r and E_v are symbolic expressions)

$E_r.f \rightarrow E_v$

- Extend configurations by a symbolic heap

Executing inhale

$\text{step}(\textbf{inhale}\ e;s,\ env,\ \pi,\ h) \equiv$

$\qquad \{\ (s,\ env,\ \pi \wedge \langle e \rangle,\ h)\ \}$

$\text{step}(\textbf{inhale}\ \textbf{acc}(e.f);s,\ env,\ \pi,\ h) \equiv$

$\qquad \{\ (s,\ env,\ \pi \wedge \langle e \rangle \neq \text{null} \wedge \bigwedge_{E.f \to \_\ \in\ h} E \neq \langle e \rangle,\ h \cup \{\ \langle e \rangle.f \to V\ \})\ \}$

$\text{step}(\textbf{inhale}\ a_1 \ast a_2;s,\ env,\ \pi,\ h) \equiv$

$\qquad \text{step}(\textbf{inhale}\ a_1;\textbf{inhale}\ a_2;s,\ env,\ \pi,\ h)$

Statement	Environment		Path Condition	Heap
	x	v		
inhale acc(x.f) **inhale** x.f == v	X	V	true	{ }
inhale x.f == v	X	V	true	{ X.f→W }
	X	V	W = V	{ X.f→W }

Statement	Environment		Path Condition	Heap
	x	y		
inhale acc(x.f) **inhale acc**(y.f)	X	Y	true	{ }
inhale acc(y.f)	X	Y	true	{ X.f→W }
	X	Y	X ≠ Y	{ X.f→W, Y.f→Z }

Executing exhale

```
inhale acc(x.f) ∗ x.f == v
exhale acc(x.f) ∗ x.f == v
```

Environment		Path Condition	Heap
x	v		
X	V	true	{ }
X	V	true	{ X.f→W }
X	V	W = V	{ X.f→W }
X	V	W = V	{ } ✗

- Solution: evaluate expressions in the heap before the exhale

Executing exhale

step(**exhale** a;s, env, π, h) $\equiv$ step'(**exhale** a;s, env, π, h, h)

step'(**exhale** e;s, env, π, h, h_0) $\equiv$
    if check($\pi \Rightarrow \langle e \rangle_0$) then { (s, env, π, h) } else $\bot$

step'(**exhale acc**(e.f);s, env, π, h, h_0) $\equiv$
    if $\langle e \rangle_0$.f$\rightarrow$_ $\in$ h then { (s, env, π, h \ { $\langle e \rangle_0$.f$\rightarrow$_ }) } else $\bot$

step'(**exhale** $a_1 * a_2$;s, env, π, h, h_0) $\equiv$
    let γ = step'(**exhale** a_1;**exhale** a_2;s, env, π, h, h_0) in
    if $\gamma = \bot$ then $\bot$ else step'(γ, h_0)

Predicates

- Predicate instances are represented by heap chunks of the form

$$p(E_1, \ldots, E_n) \to E_v$$

where E_v is a symbolic expression denoting the instance's footprint

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

$$FP_{list} ::= (Ref, Int, FP_{list}) \mid \varepsilon$$

Heap-Dependent Functions

- Symbolic expressions may contain function applications

```
function balance(this: Ref): Int
  requires acc(this.bal)
```

balance: Ref × **Int**→ **Int**

- Predicate footprints allow framing

```
function length(this: Ref): Int
  requires list(this)
```

length: Ref × $\mathrm{FP}_{\mathrm{list}}$→ **Int**

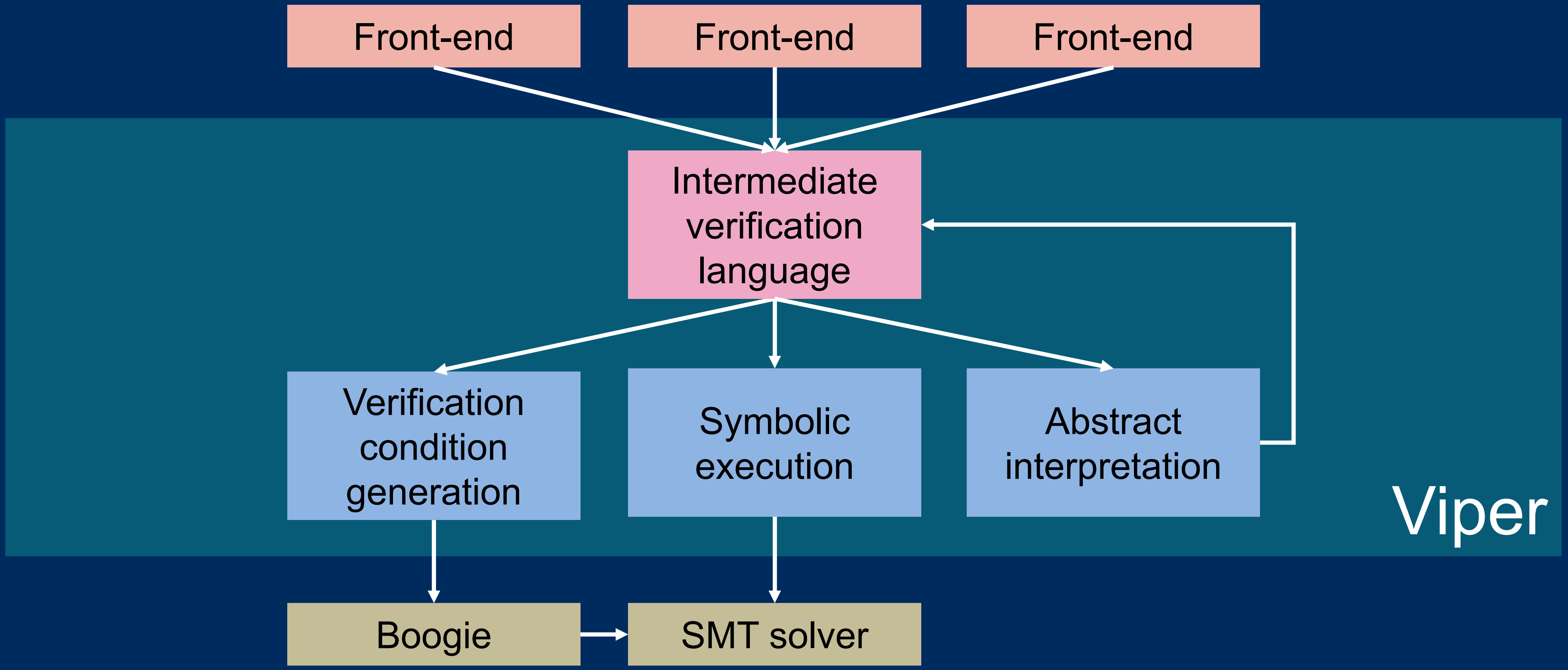
Pros and Cons of Symbolic Execution

Pros

- Small prover queries lead to better, more predictable performance
- Higher control over proof search enables dedicated algorithms

Cons

- Separation of path and heap information can lead to incompleteness
- Internal treatment of heap reasoning requires dedicated algorithms



Front-end

Front-end

Front-end

Intermediate verification language

Verification condition generation

Symbolic execution

Abstract interpretation

Viper

Boogie

SMT solver

Summary

- SMT solvers can be used to automate permission-based verification

- Verification condition generation encodes total-heap semantics
- VCG can be implemented as an encoding into existing languages and tools

- Symbolic execution implements partial-heap semantics
- Performs heap-reasoning internally, and uses SMT solver to reason about value information

Outline

- Permission-based Verification
- The Viper Intermediate Language
- Building Verifiers
- Encoding of Advanced Verification Techniques

Reminder: Encoding Monitors

```
class Account {
  var bal: int

  invariant acc(this.bal)

  method deposit(amount: int)
  {
    acquire this
    this.bal := this.bal + amount
    release this
  }
}
```

```
inhale acc(this.bal)
this.bal := this.bal + amount
exhale acc(this.bal)
```

Non-Blocking Data Structures

- Permissions ensures data race freedom
  - Monitors and other synchronization are used to transfer ownership between threads

- Non-blocking data structures can increase performance by allowing extra concurrency
  - Synchronization is done through atomic operations such as compare-and-swap (CAS)
  - Data races are permitted

```
typedef int SpinLock;

void Lock(SpinLock* sl) {
  while(CAS(sl, 0, 1))
    ;
}

void UnLock(SpinLock* sl) {
  *sl = 0;
}
```

Weak Memory

- Modern hardware often does not provide sequentially consistent shared memory
- Weak memory permits behaviors that are not possible under sequential consistency
- However, data-race free programs have only sequentially consistent behaviors

```
                a = 0;
                b = 0;

a = 1;          ||        b = 1;
print(b);       ||        print(a);
```

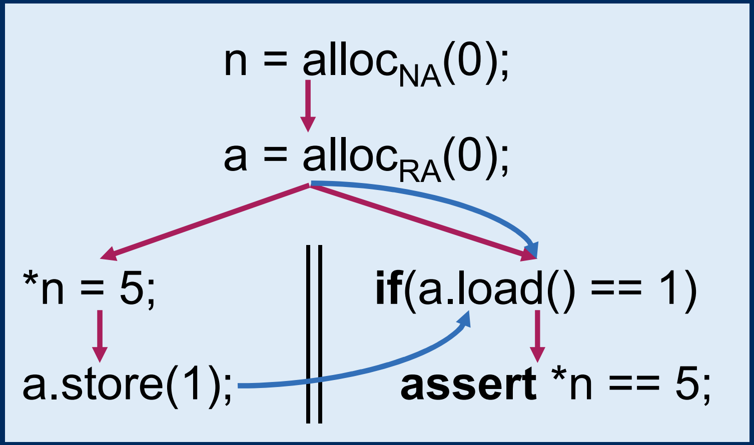
Possible results:
under SC: 10, 01, 11
under WM: also 00

C11

- The C11 memory model provides several kinds of variables

- Non-atomic variables
    - Data races are errors

- Atomic variables with release-write and acquire-read
    - Writes and reads are synchronized



n = alloc_{NA}(0);

a = alloc_{RA}(0);

*n = 5;

a.store(1);

||

if(a.load() == 1)

assert *n == 5;

- Relaxed separation logic (RSL) supports some features of the C11 memory model

Non-Atomic Variables

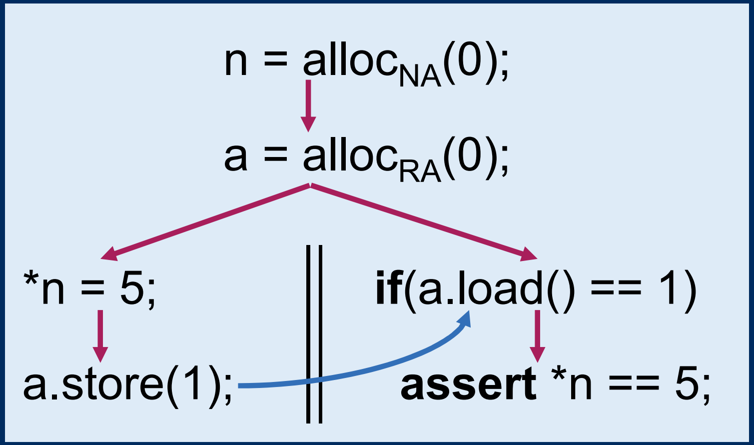
- Permissions prevent data races on non-atomic variables

$$\{\ \text{true}\ \}\ \ x = \text{alloc}_{NA}(v)\ \ \{\ x \rightarrow v\ \}$$

$$\{\ x \rightarrow \_\ \}\ \ {}^*x = v\ \{\ x \rightarrow v\ \}$$

$$\{\ x \rightarrow V\ \}\ \ t = {}^*x\ \{\ x \rightarrow V \ast t = V\ \}$$

Release-Acquire


n = alloc$_{NA}$(0);

a = alloc$_{RA}$(0);

*n = 5;

a.store(1);

||

if(a.load() == 1)

assert *n == 5;

- Races on atomic variables are permitted
- Release-acquire can be seen as message passing
- Messages may transfer ownership to non-atomic variables

Reminder: Monitor Invariants

- Monitor invariant Q specifies an assertion that holds when monitor is not currently held

- Acquire transfers ownership of Q from monitor to thread

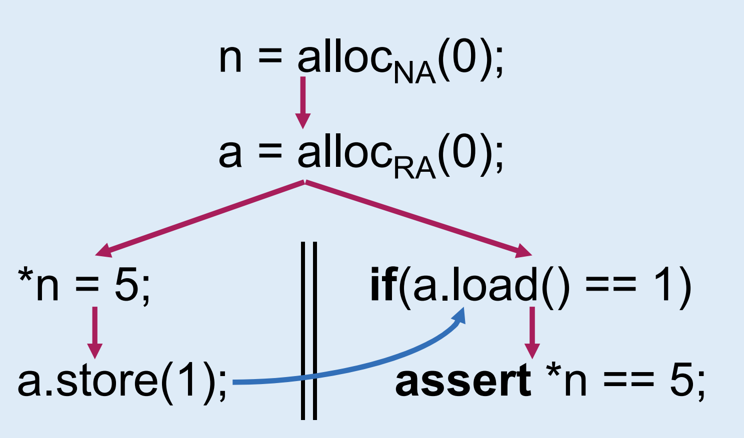
- Release transfers ownership of Q from thread to monitor

```
class Account {
  var bal: int

  invariant acc(this.bal)

  method deposit(amount: int)
  {
    acquire this
    this.bal := this.bal + amount
    release this
  }
}
```

Location Invariants

- Location invariant Q(v) specifies an assertion that holds when the location has value v
- Acquire-read of value v transfers ownership of Q(v) from atomic variable to thread
- Release-write of value v transfers ownership of Q(v) from thread to atomic variable

$$Q(v) \equiv \begin{cases} n \to 5 & \text{if } v = 1 \\ \text{true} & \text{otherwise} \end{cases}$$



n = alloc_{NA}(0);

a = alloc_{RA}(0);

*n = 5; || **if**(a.load() == 1)

a.store(1); || **assert** *n == 5;

Proof Rules

- Choose location invariant when allocating an atomic location

$$\{ Q(v) \} \;\; x = \mathrm{alloc}_{RA}(v) \;\; \{ \mathrm{Rel}_Q(x) \ast \mathrm{Acq}_Q(x) \}$$

- Release-write gives up ownership

$$\{ \mathrm{Rel}_Q(x) \ast Q(v) \} \;\; x.\mathrm{store}(v) \; \{ \mathrm{Rel}_Q(x) \}$$

- Acquire-read gains ownership

$$\{ \mathrm{Acq}_Q(x) \} \;\; t = x.\mathrm{load}() \; \{ Q(t) \ast \mathrm{Acq}_Q(x) \}$$

Proof Outline

$\{ \text{true} \}$

$n = \text{alloc}_{NA}(0);$

$\{ n \rightarrow 0 \}$

$a = \text{alloc}_{RA}(0);$

$\{ n \rightarrow 0 \ast \text{Rel}_Q(a) \ast \text{Acq}_Q(a) \}$

$\{ n \rightarrow 0 \ast \text{Rel}_Q(a) \}$

$*n = 5;$

$\{ n \rightarrow 5 \ast \text{Rel}_Q(a) \}$

$a.\text{store}(1);$

$\{ \text{Rel}_Q(a) \}$

$\{ \text{Acq}_Q(a) \}$

if(a.load() == 1)

$\{ Q(1) \ast \text{Acq}_Q(a) \}$

assert *n == 5;

$$Q(v) \equiv \begin{cases} n \rightarrow 5 & \text{if } v = 1 \\ \text{true} & \text{otherwise} \end{cases}$$

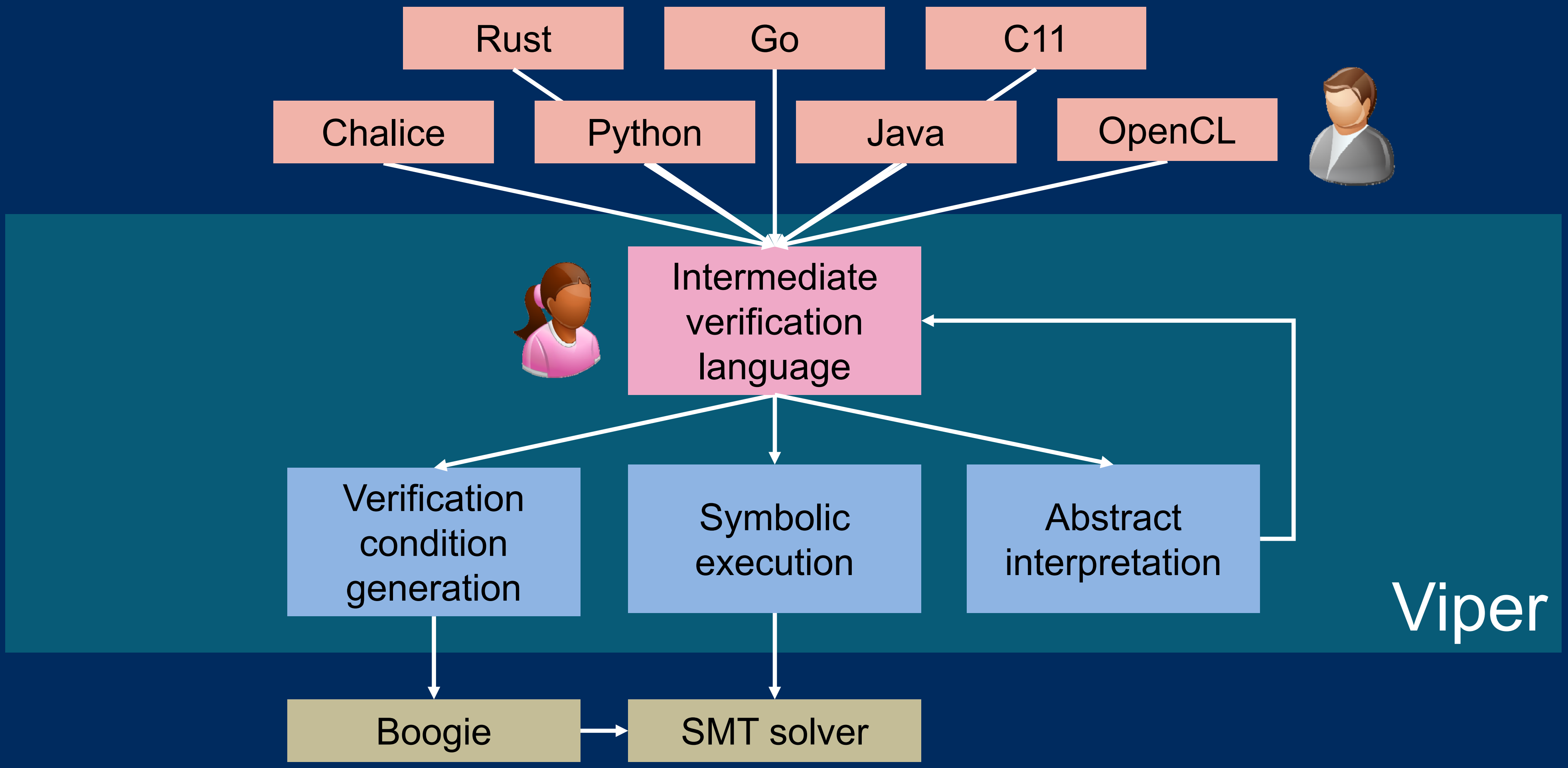
Proof Rules

$$\{\ \mathrm{Acq}_Q(x)\ \}\ \ t = x.\mathrm{load}()\ \{\ Q(t) \ast \mathrm{Acq}_{Q[t\ :=\ \mathrm{true}]}(x)\ \}$$

- Reading the same value more than once would duplicate permissions

$$Q(v) \equiv \begin{cases} n \rightarrow 5 & \text{if } v = 1 \\ \text{true} & \text{otherwise} \end{cases}$$

```
x = a.load();
y = a.load();
if(x == 1 && y == 1)
   assert false;
```



Rust

Go

C11

Chalice

Python

Java

OpenCL

Intermediate verification language

Verification condition generation

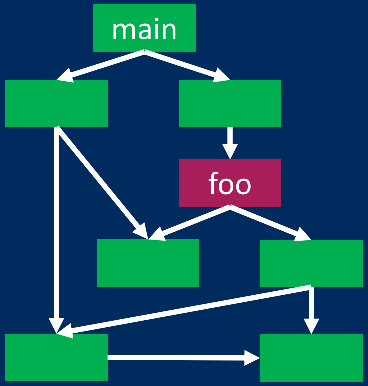
Symbolic execution

Abstract interpretation

Viper

Boogie

SMT solver



main

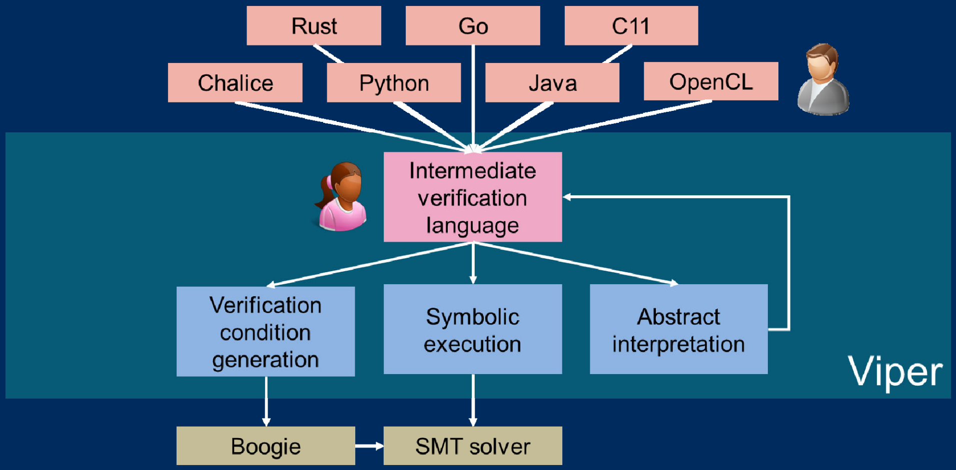
foo

Modularity is important for scalability, components, and evolution

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

$$\frac{\{\,P_1\,\}\ S_1\ \{\,Q_1\,\} \qquad \{\,P_2\,\}\ S_2\ \{\,Q_2\,\}}{\{\,P_1 * P_2\,\}\ S_1 \,||\, S_2\ \{\,Q_1 * Q_2\,\}}$$

Permissions enable framing and reasoning about concurrency



Rust

Go

C11

Chalice

Python

Java

OpenCL

Intermediate verification language

Verification condition generation

Symbolic execution

Abstract interpretation

Viper

Boogie

SMT solver

Intermediate languages enable reuse of infrastructure

VIPER

Viper lets you encode a wide variety of reasoning techniques

References: Lecture 1

- J. C. Reynolds: Separation Logic: A Logic for Shared Mutable Data Structures. LICS 2002
- P. W. O'Hearn: Resources, Concurrency and Local Reasoning. CONCUR 2004
- J. Smans, B. Jacobs, F. Piessens: Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. ECOOP 2009
- M. J. Parkinson, A. J. Summers: The Relationship between Separation Logic and Implicit Dynamic Frames. Logical Methods in Computer Science 8(3), 2012
- K. R. M. Leino, P. Müller: A Basis for Verifying Multi-threaded Programs. ESOP 2009

References: Lecture 2

- P. Müller, M. Schwerhoff, A. J. Summers: Viper: A Verification Infrastructure for Permission-Based Reasoning. VMCAI 2016
- K. R. M. Leino, P. Müller: A Basis for Verifying Multi-threaded Programs. ESOP 2009
- P. Müller, M. Schwerhoff, A. J. Summers: Automatic Verification of Iterated Separating Conjunctions using Symbolic Execution. CAV 2016
- M. J. Parkinson, G. M. Bierman: Separation logic and abstraction. POPL 2005

References: Lecture 3

- K. R. M. Leino, P. Müller: A Basis for Verifying Multi-threaded Programs. ESOP 2009
- S. Heule, I. T. Kassios, P. Müller, A. J. Summers: Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions. ECOOP 2013
- M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, K. R. M. Leino: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. FMCO 2005
- J. Berdine, C. Calcagno, P. W. O'Hearn: Smallfoot: Modular Automatic Assertion Checking with Separation Logic. FMCO 2005
- J. Smans, B. Jacobs, F. Piessens: Heap-Dependent Expressions in Separation Logic. FMOODS/FORTE 2010
- I. T. Kassios, P. Müller, M. Schwerhoff: Comparing Verification Condition Generation with Symbolic Execution: An Experience Report. VSTTE 2012

References: Lecture 4

- A. J. Summers, P. Müller: Automating Deductive Verification for Weak-Memory Programs, TACAS 2018
- P. Müller, M. Schwerhoff, A. J. Summers: Viper: A Verification Infrastructure for Permission-Based Reasoning. VMCAI 2016
- V. Vafeiadis, C. Narayan: Relaxed separation logic: a program logic for C11 concurrency. OOPSLA 2013