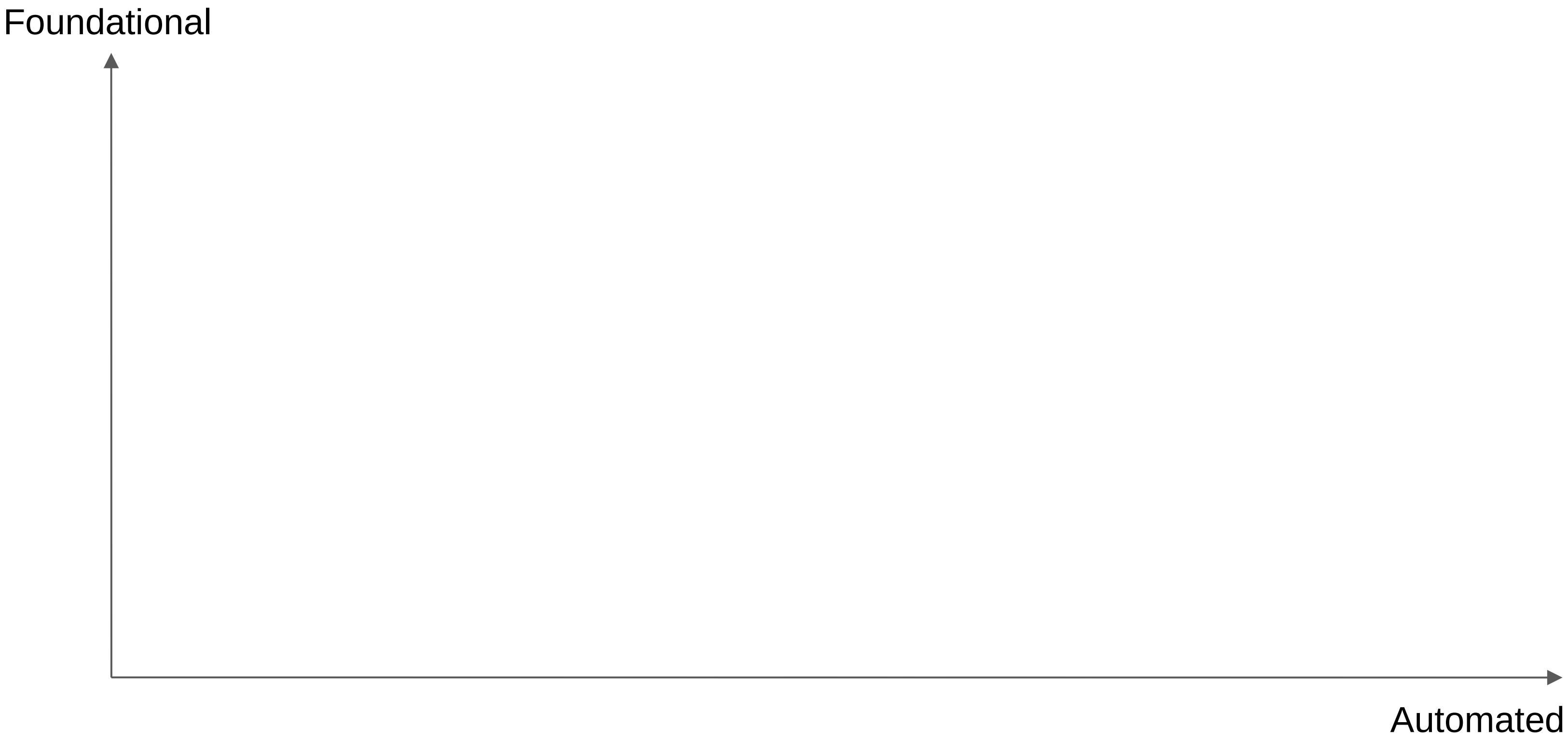
A Verification Framework Designed to Automate Separation Logic

Thibault Dardinier

ETH zürich

Program Verifiers Based on Separation Logic

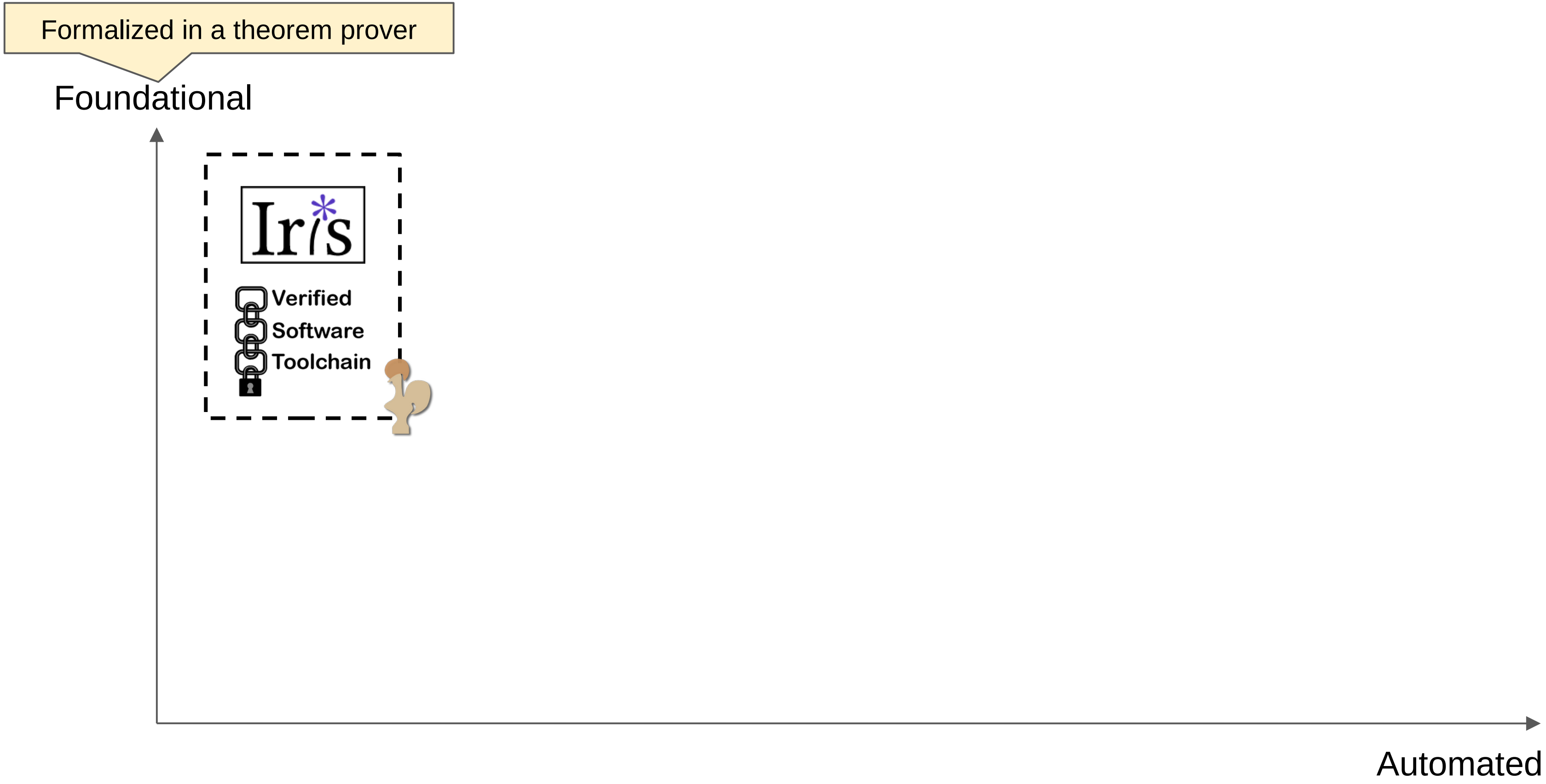


Program Verifiers Based on Separation Logic

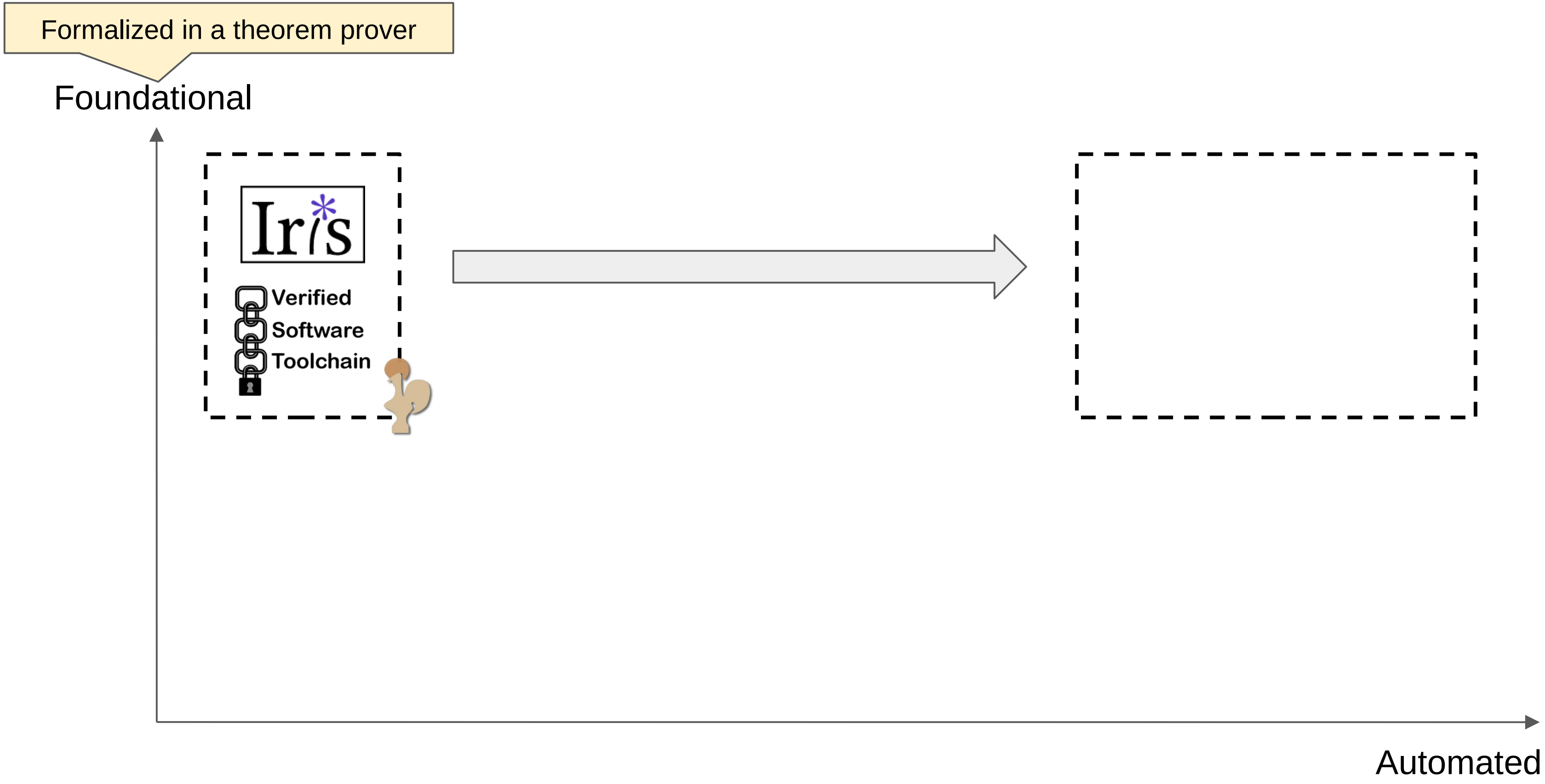
Program Verifiers Based on Separation Logic

Program Verifiers Based on Separation Logic

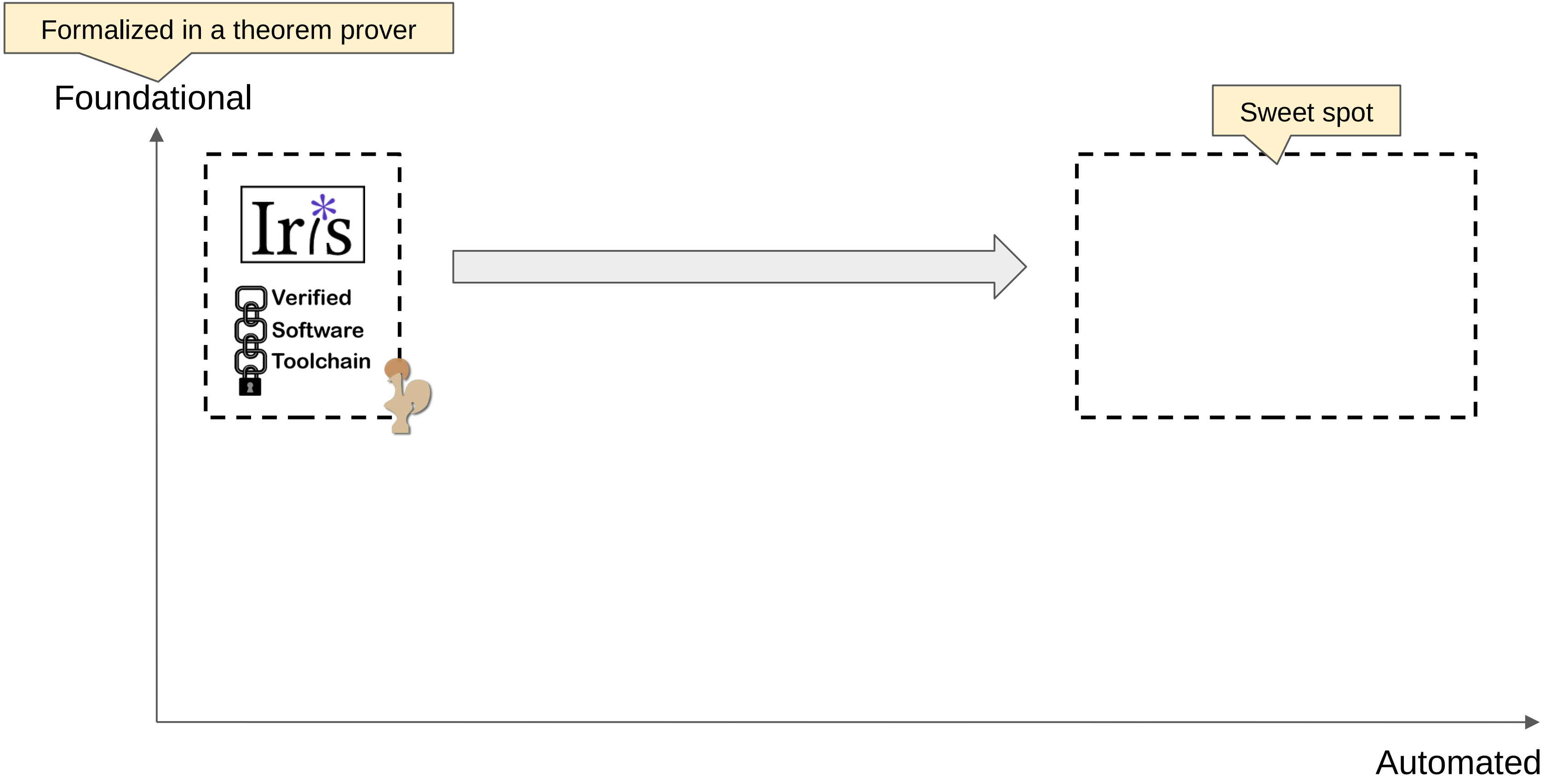
Program Verifiers Based on Separation Logic



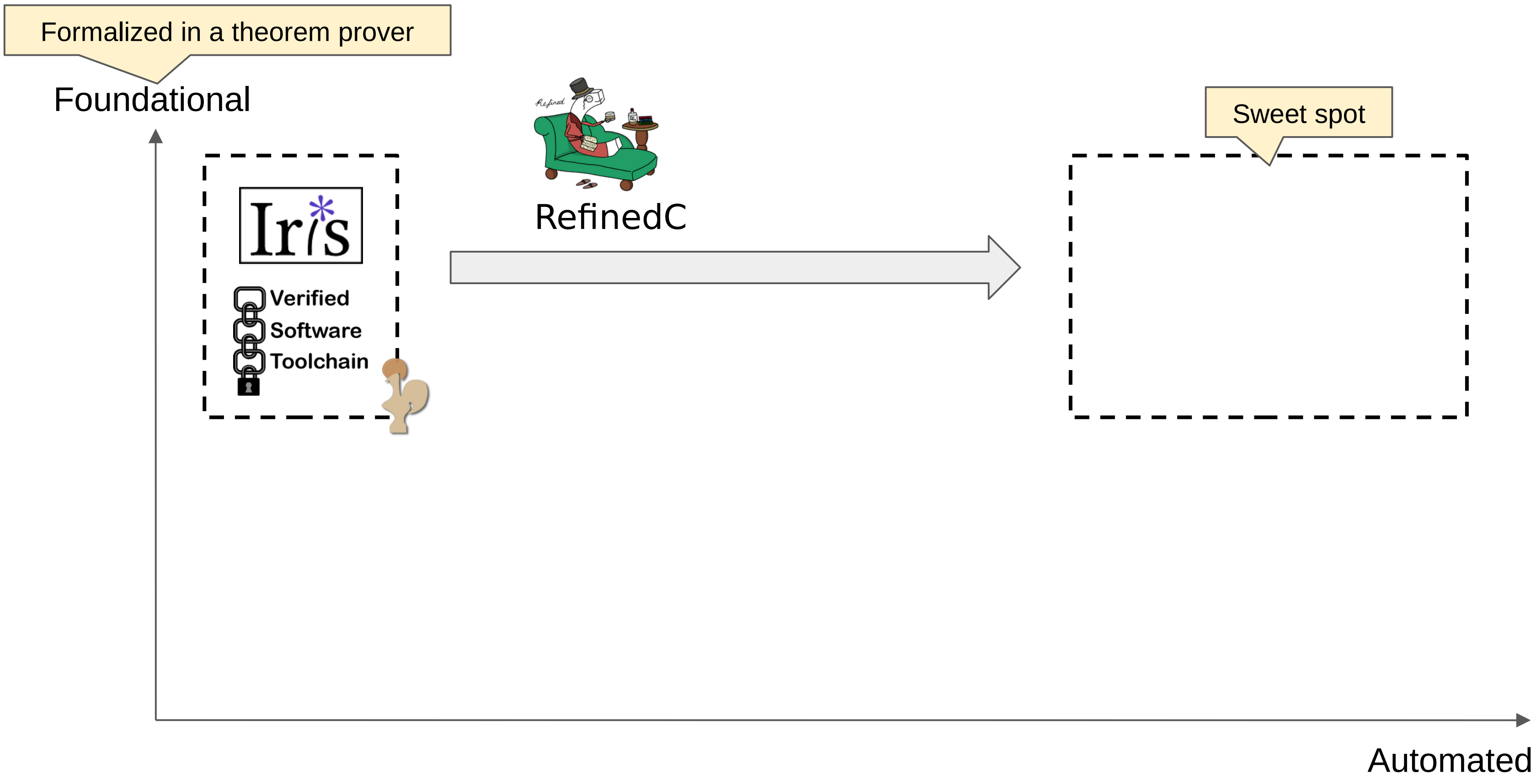
Program Verifiers Based on Separation Logic



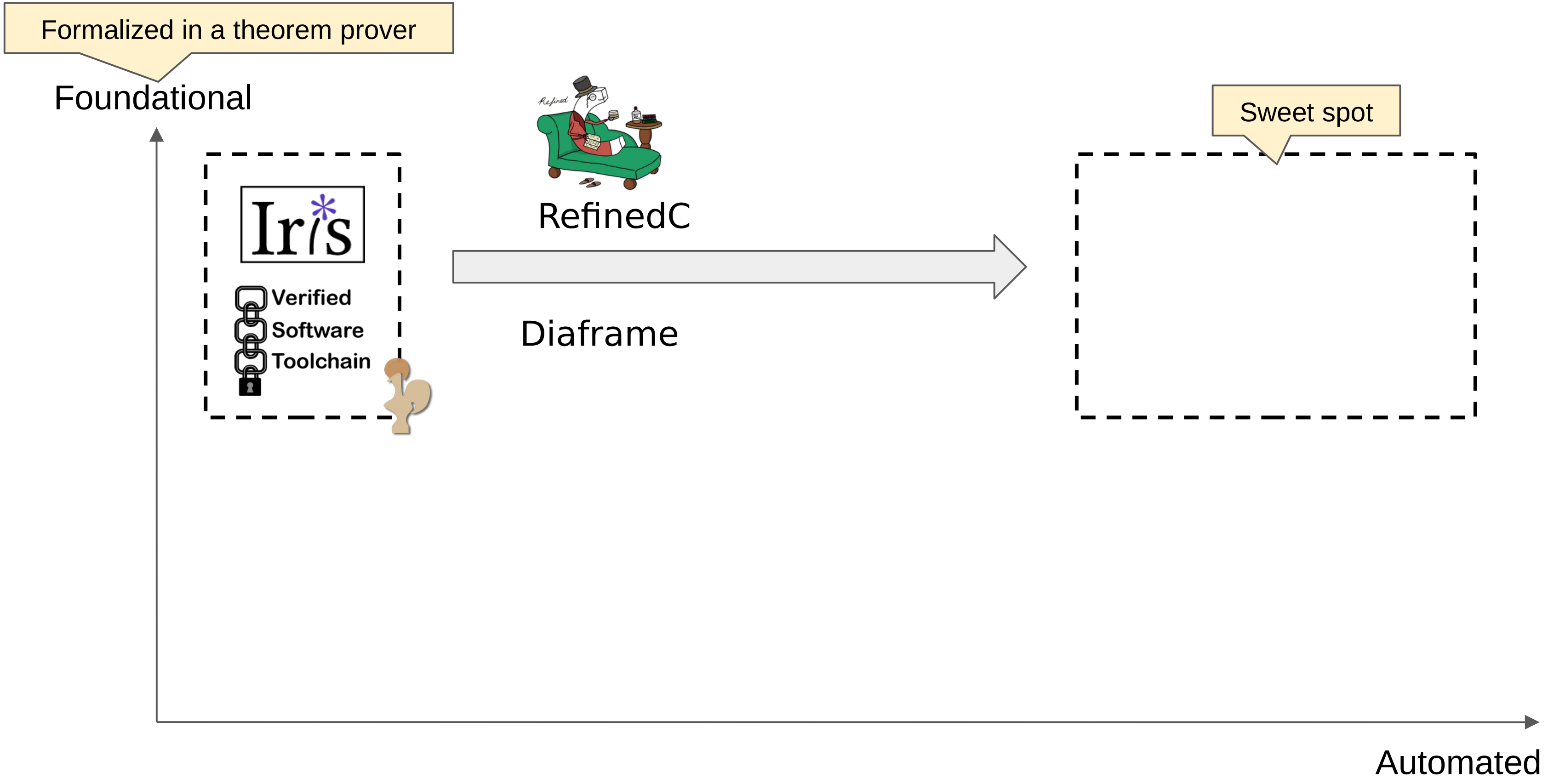
Program Verifiers Based on Separation Logic



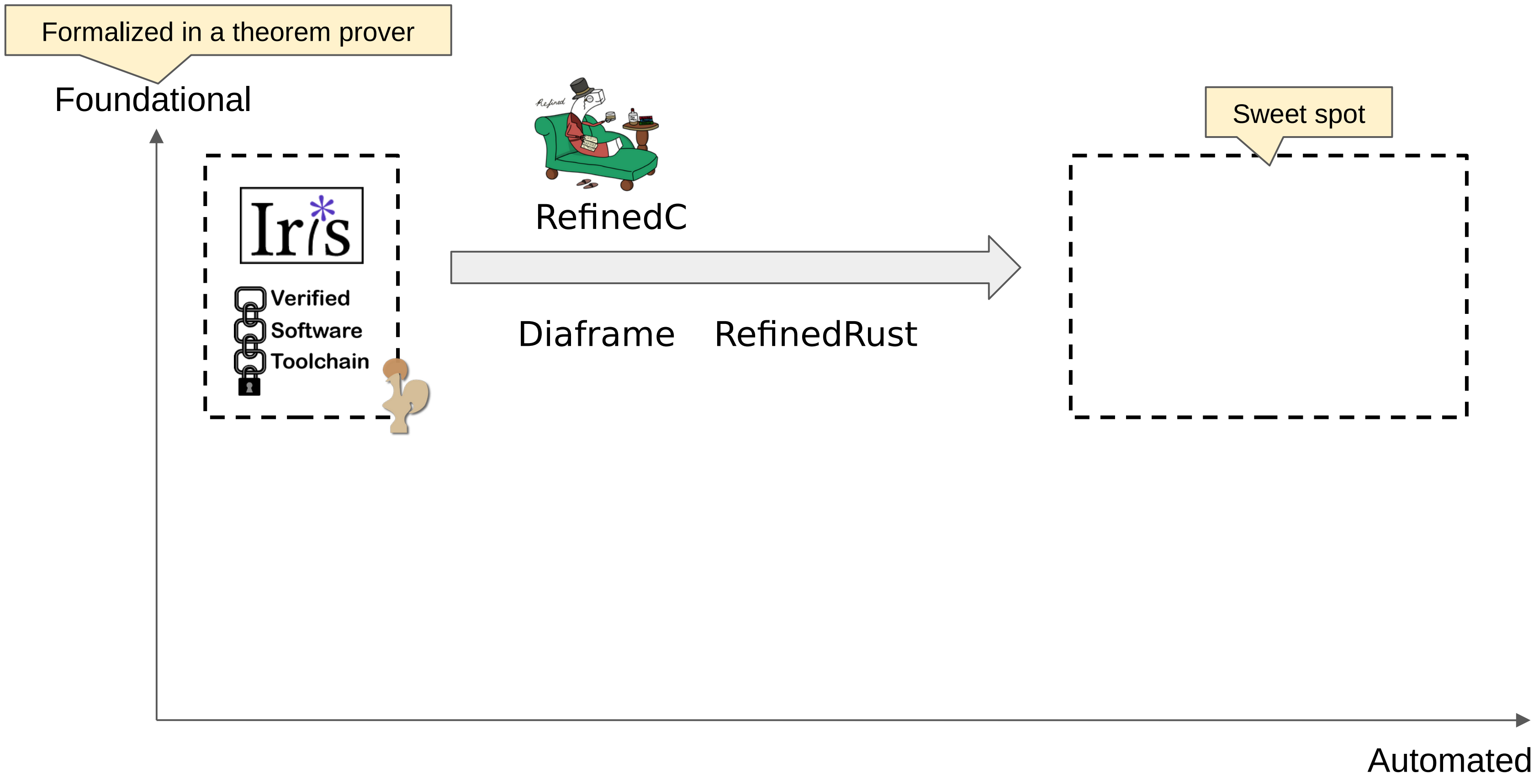
Program Verifiers Based on Separation Logic



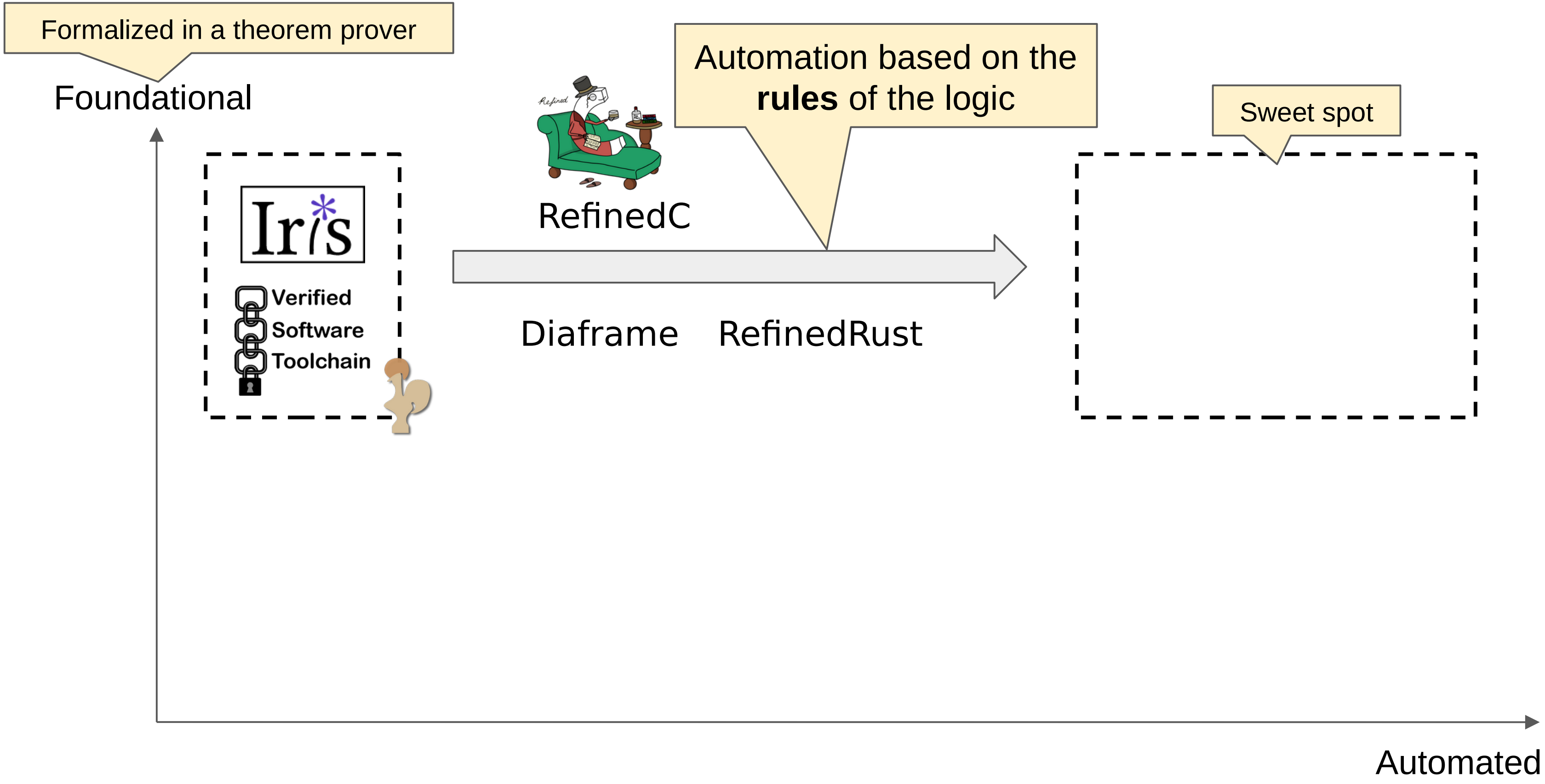
Program Verifiers Based on Separation Logic



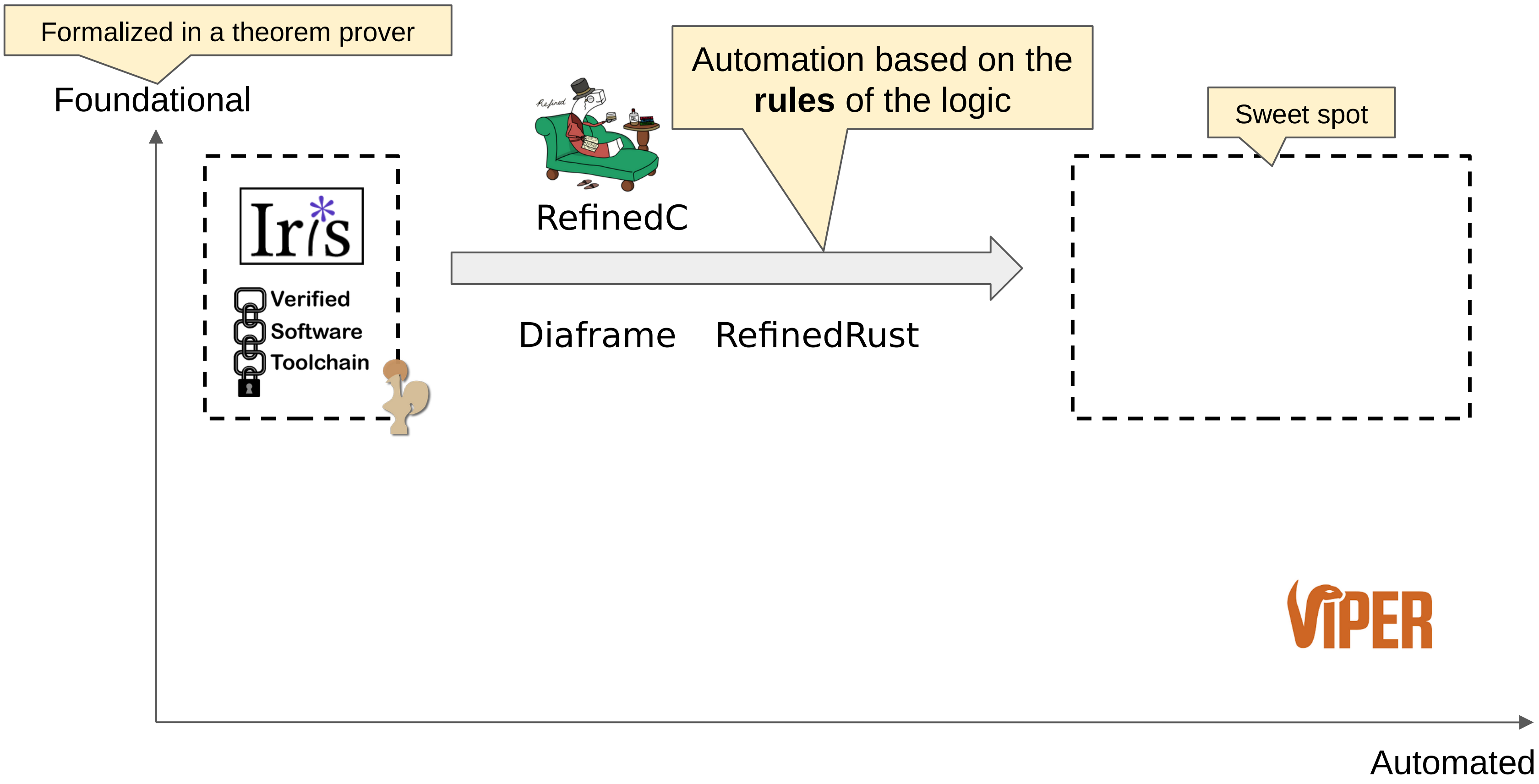
Program Verifiers Based on Separation Logic



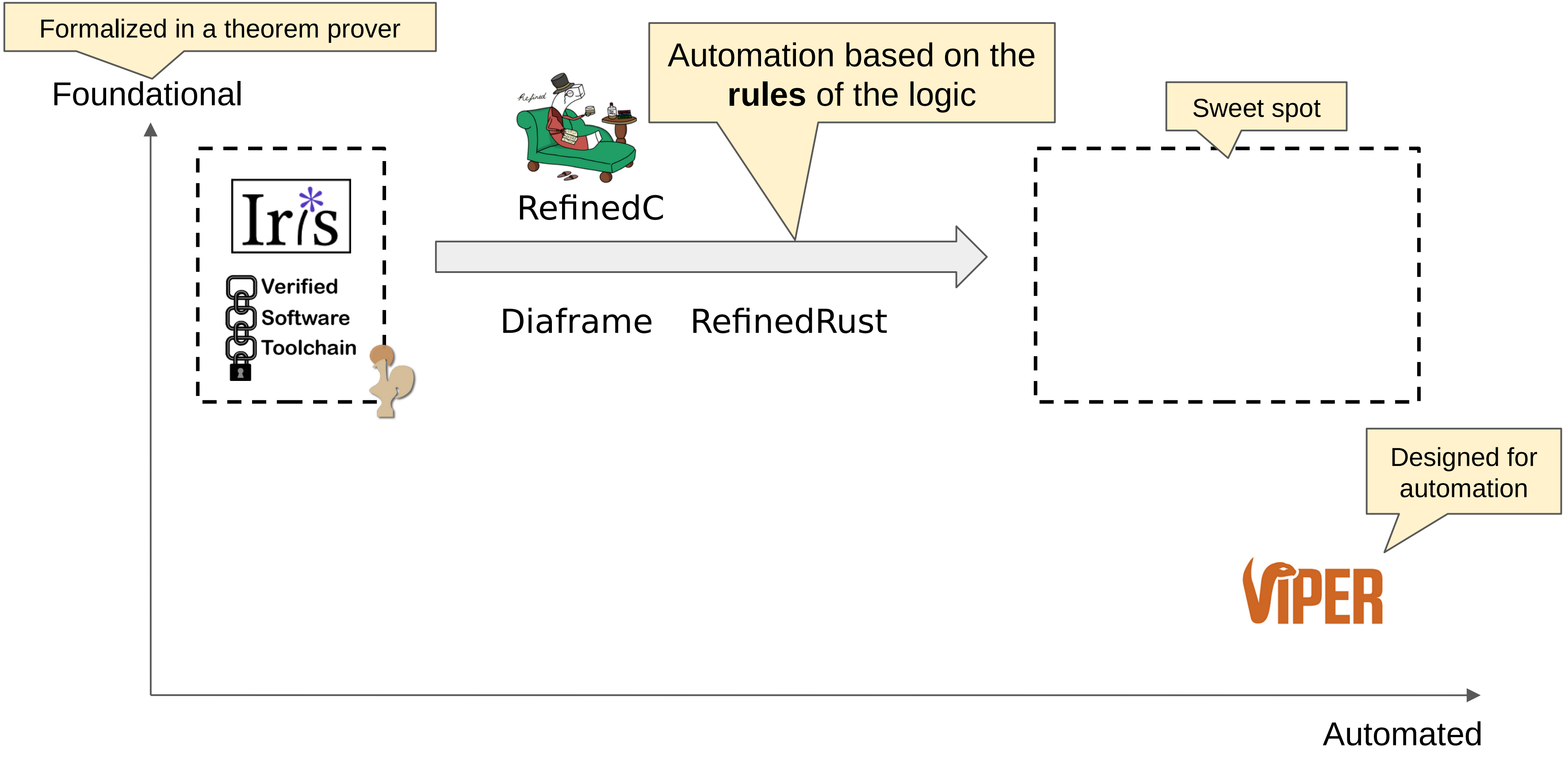
Program Verifiers Based on Separation Logic



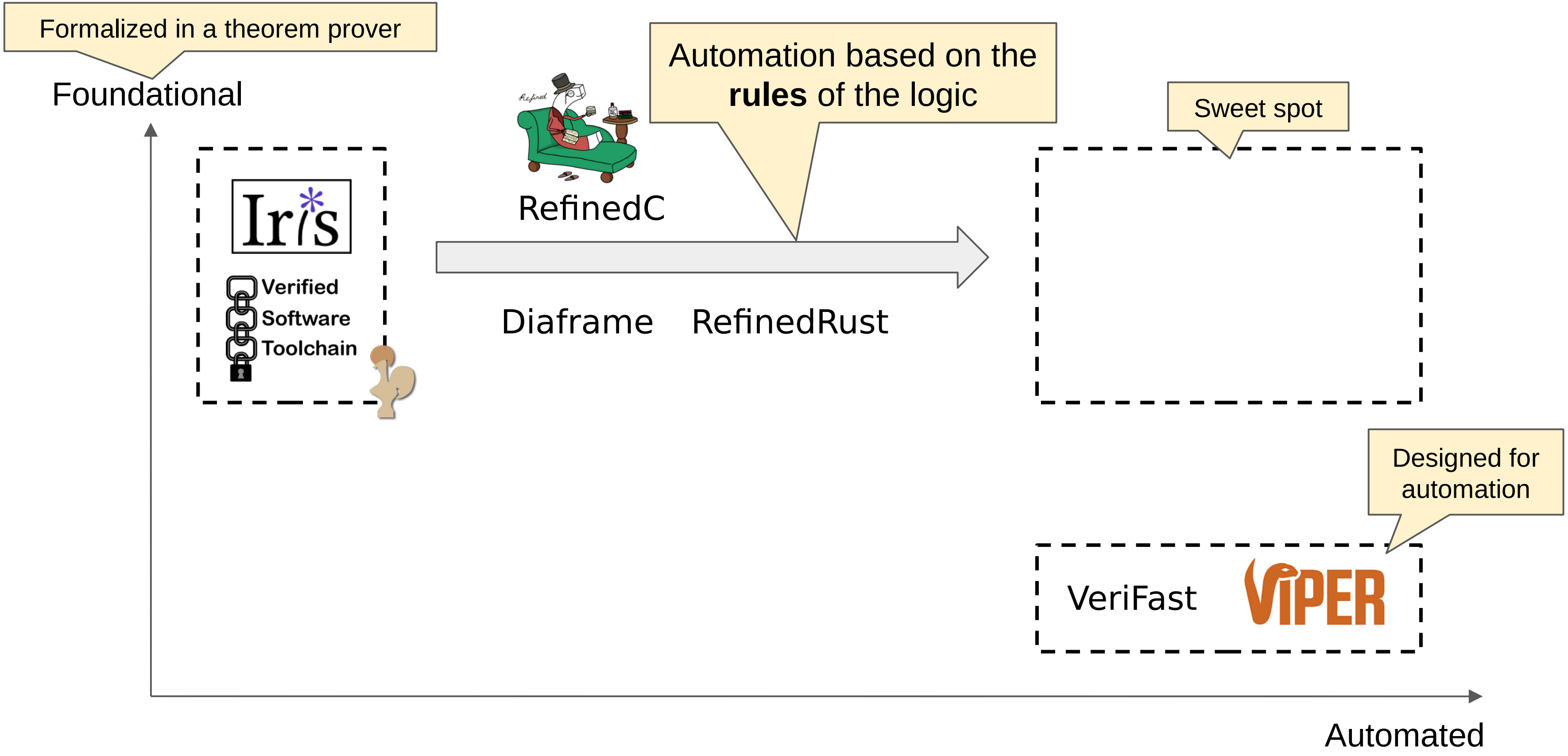
Program Verifiers Based on Separation Logic



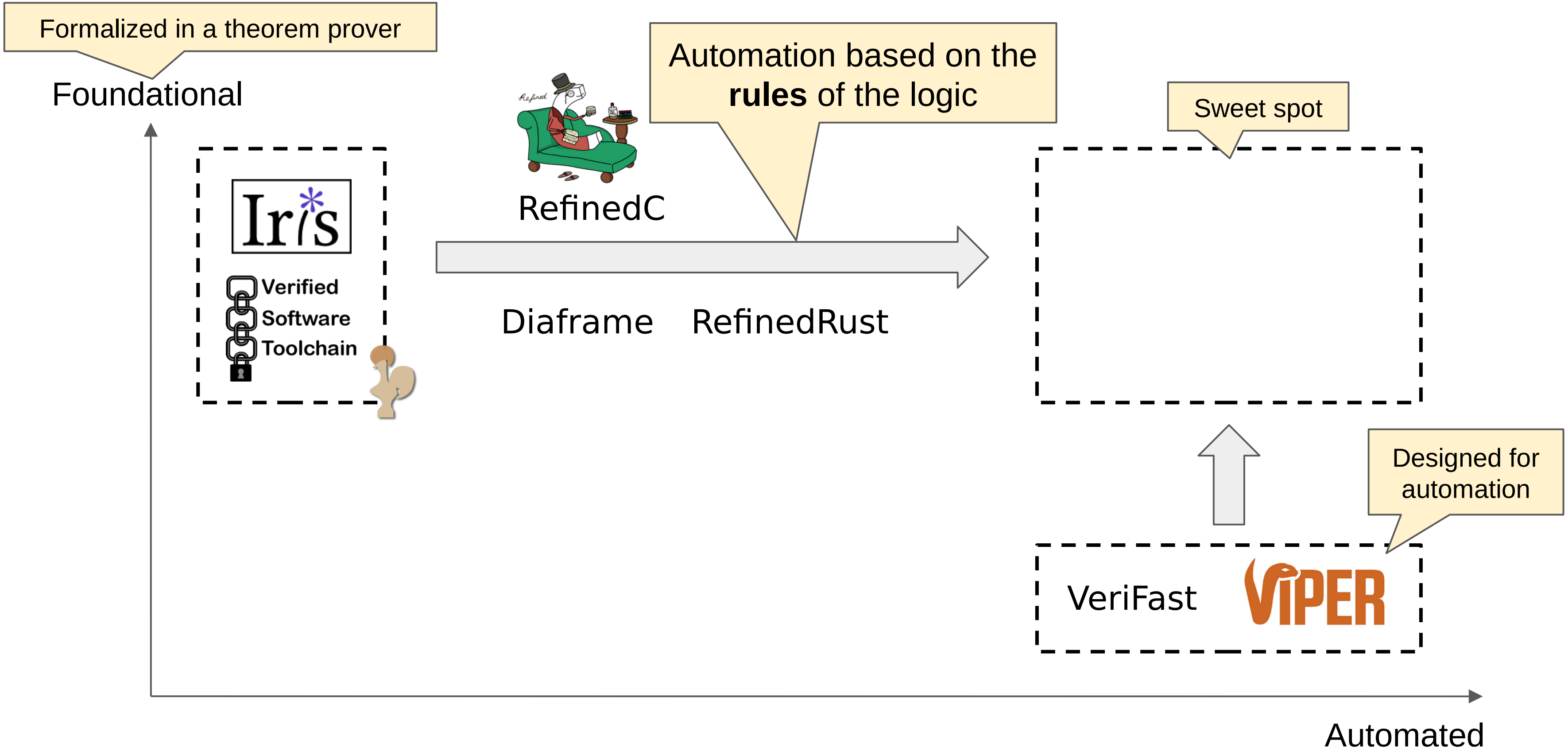
Program Verifiers Based on Separation Logic



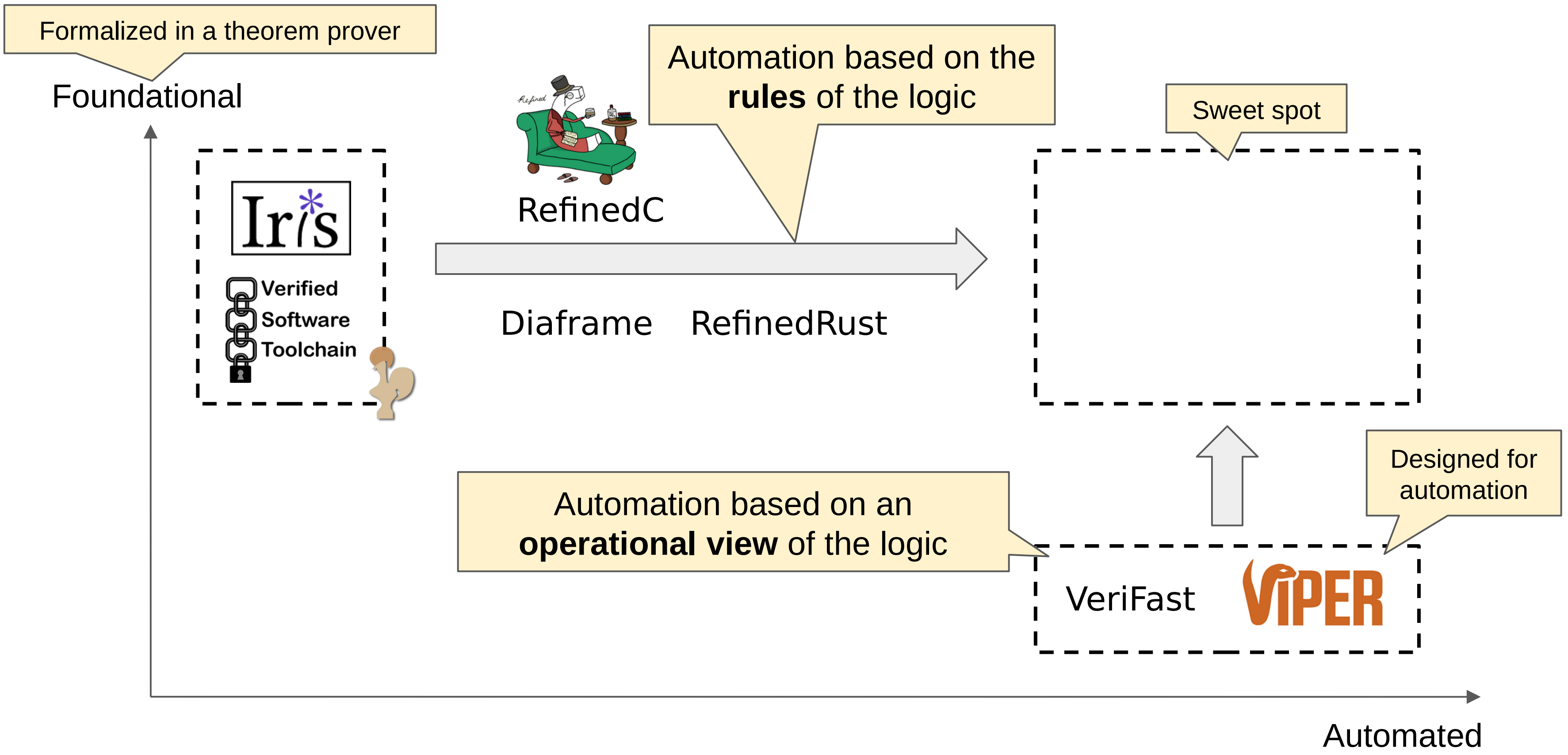
Program Verifiers Based on Separation Logic



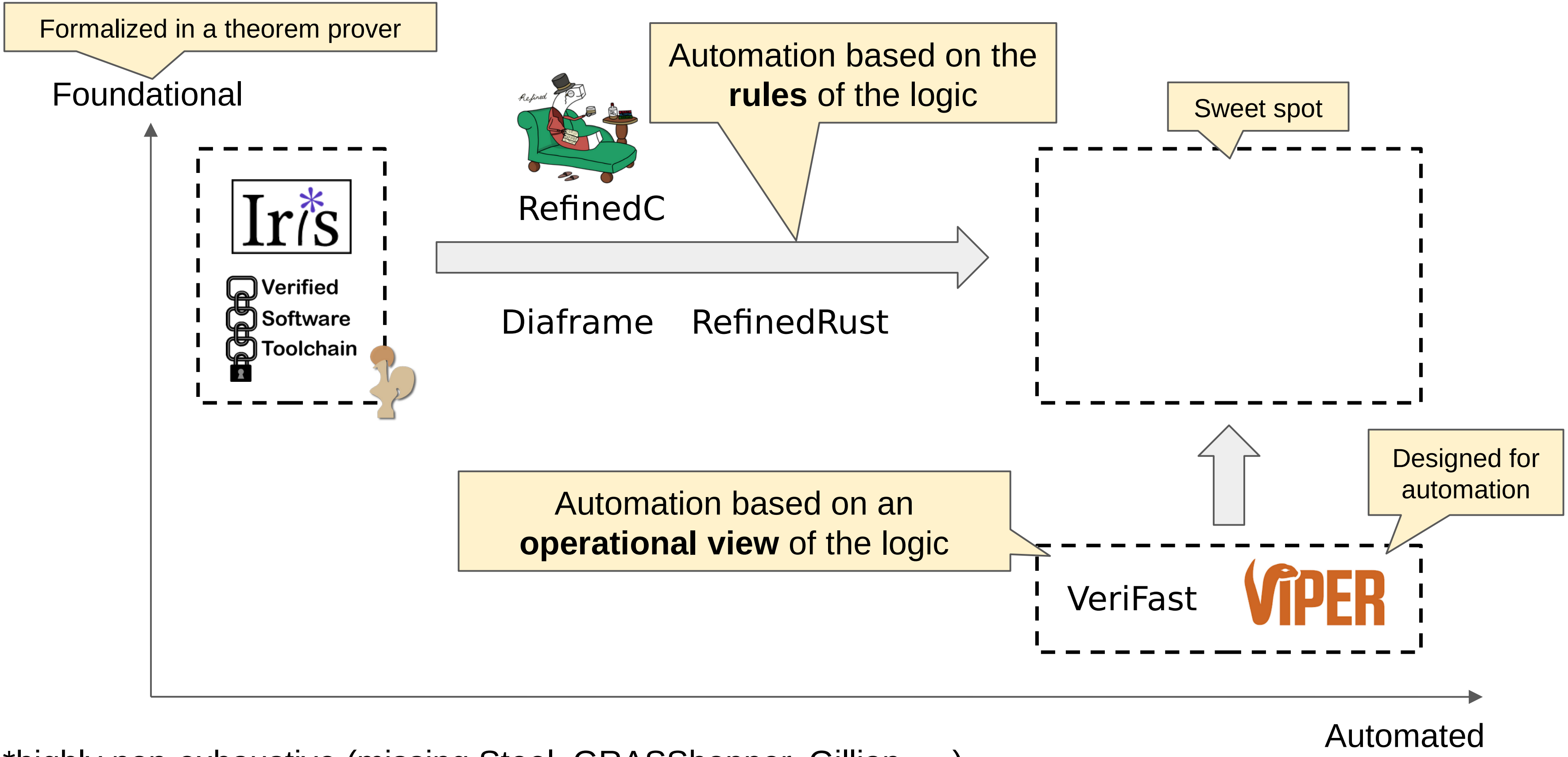
Program Verifiers Based on Separation Logic



Program Verifiers Based on Separation Logic



Program Verifiers Based on Separation Logic



*highly non-exhaustive (missing Steel, GRASShopper, Gillian, …)

Outline of the Talk

Outline of the Talk

1. Overview of Viper

Outline of the Talk

1. Overview of Viper

2. Inhale and Exhale: An Operational View of Separation Logic

Outline of the Talk

1. Overview of Viper
2. Inhale and Exhale: An Operational View of Separation Logic
3. Designed for Automation

Outline of the Talk

1. Overview of Viper
2. Inhale and Exhale: An Operational View of Separation Logic
3. Designed for Automation
4. Toward a Foundational Viper

Outline of the Talk

1. **Overview of Viper**

2. Inhale and Exhale: An Operational View of Separation Logic

3. Designed for Automation

4. Toward a Foundational Viper

Demo

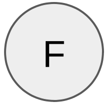
```
field x: Int
field y: Int

method main(point: Ref)
    requires acc(point.x) && acc(point.y)
    // point.x |-> _ * point.y |-> _
{
    point.x := 5
    point.y := 7
    add(point)
    assert point.x == 5
    assert point.y == 12
}

method add(p: Ref)
    requires acc(p.x, 1/2) && acc(p.y)
    ensures acc(p.x, 1/2) && acc(p.y)
    // ensures p.y == old(p.x + p.y)
{
    p.y := p.x + p.y
}
```

⊗ 1 ⚠ 0 silicon × Verifying demo.vpr failed after 0.5 seconds with 1 error Live Share "demo.vpr" 21L 357C written UTF-8 LF Viper

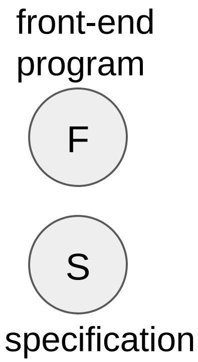
The Viper Verification Framework

The Viper Verification Framework

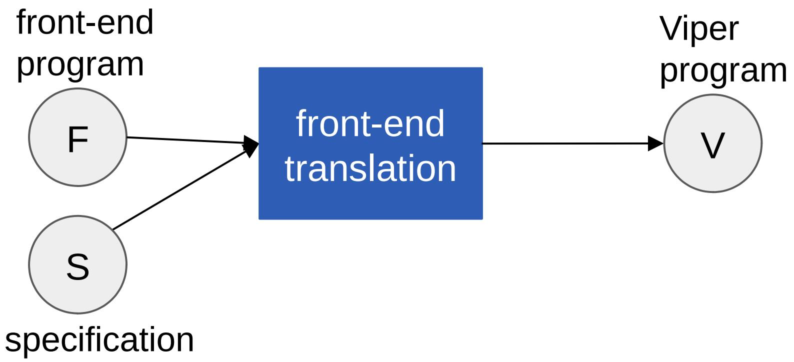
front-end
program



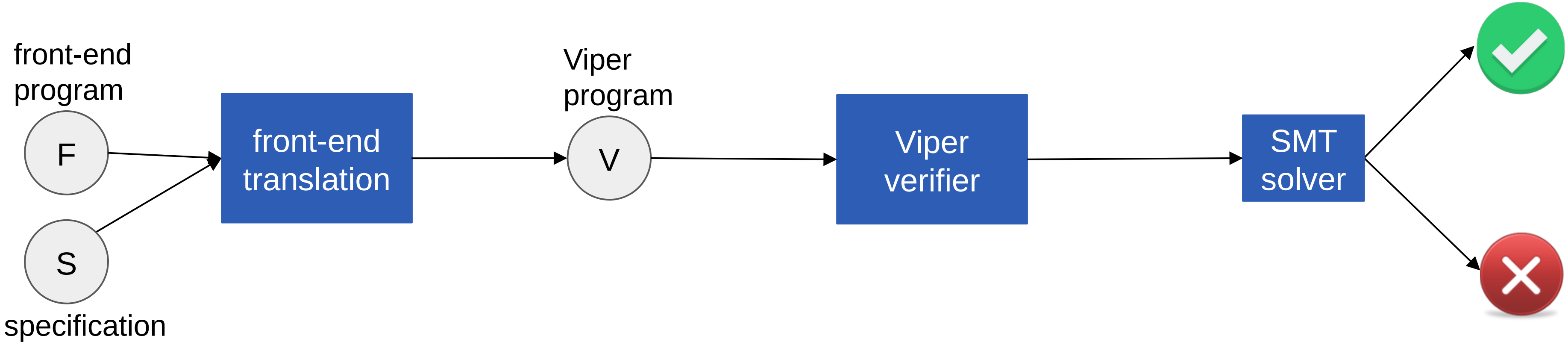
The Viper Verification Framework



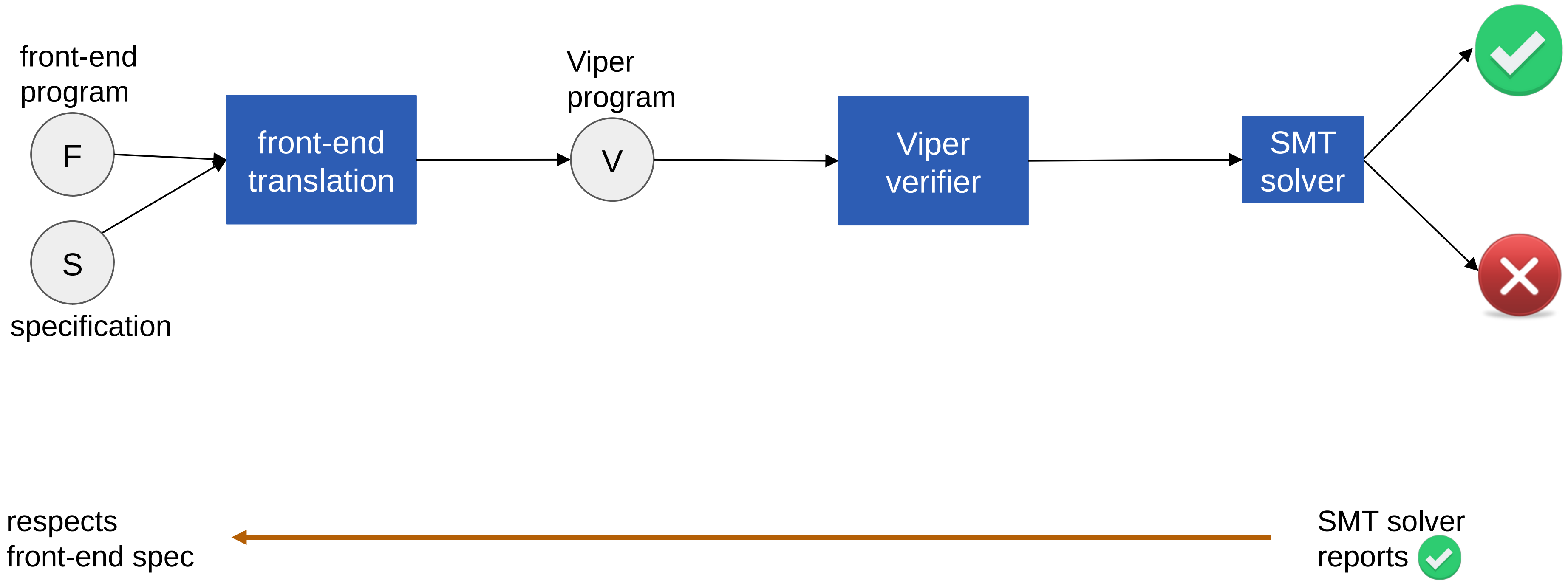
The Viper Verification Framework



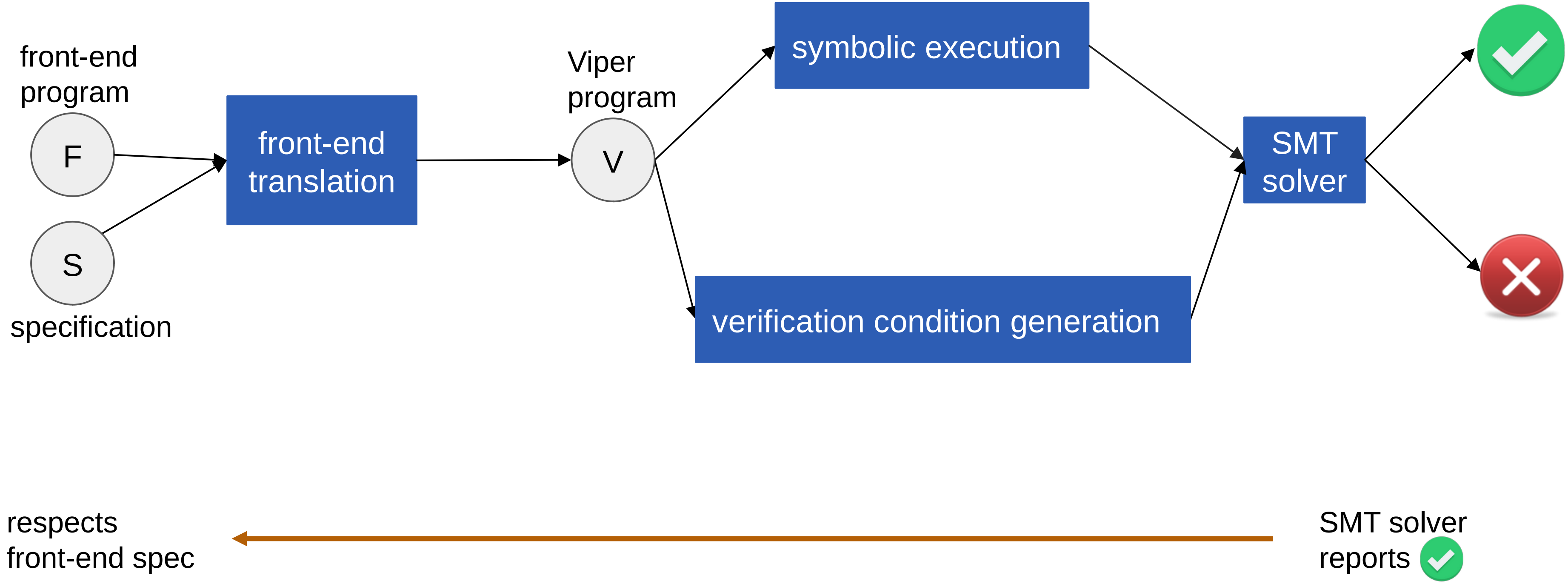
The Viper Verification Framework



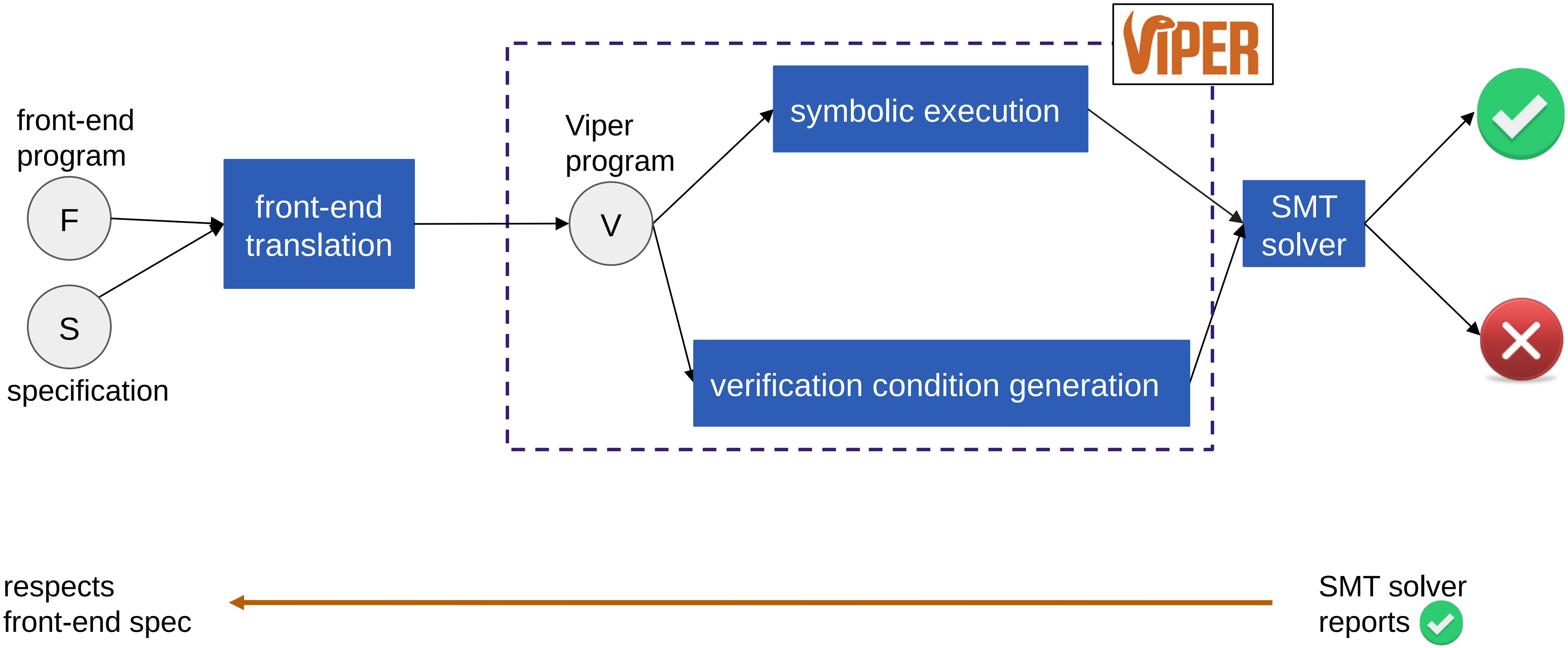
The Viper Verification Framework



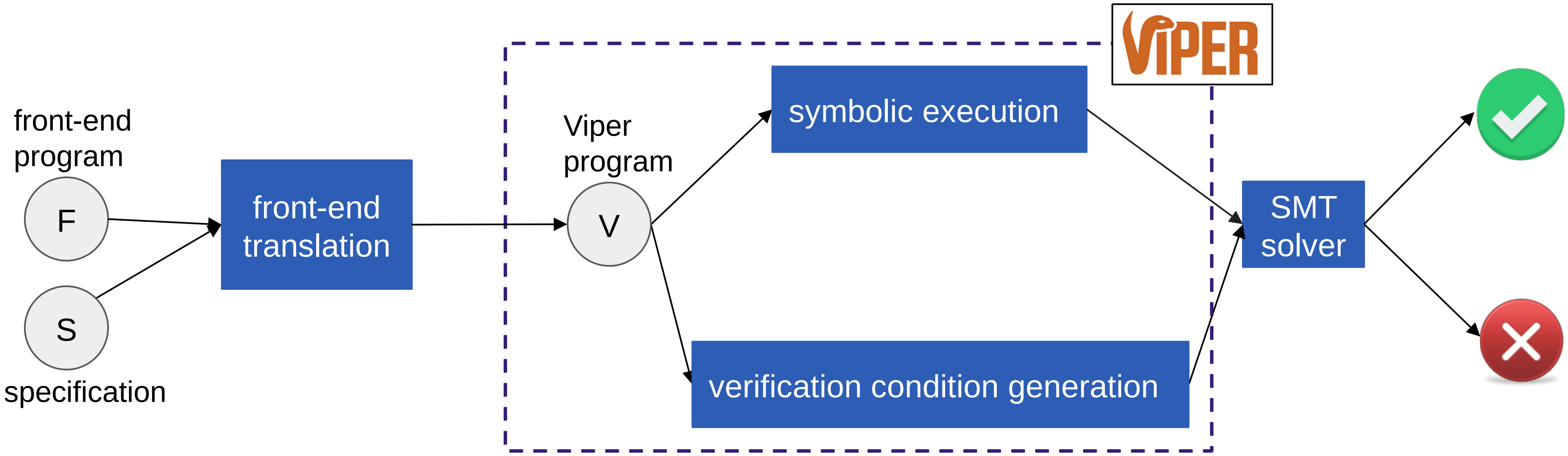
The Viper Verification Framework



The Viper Verification Framework



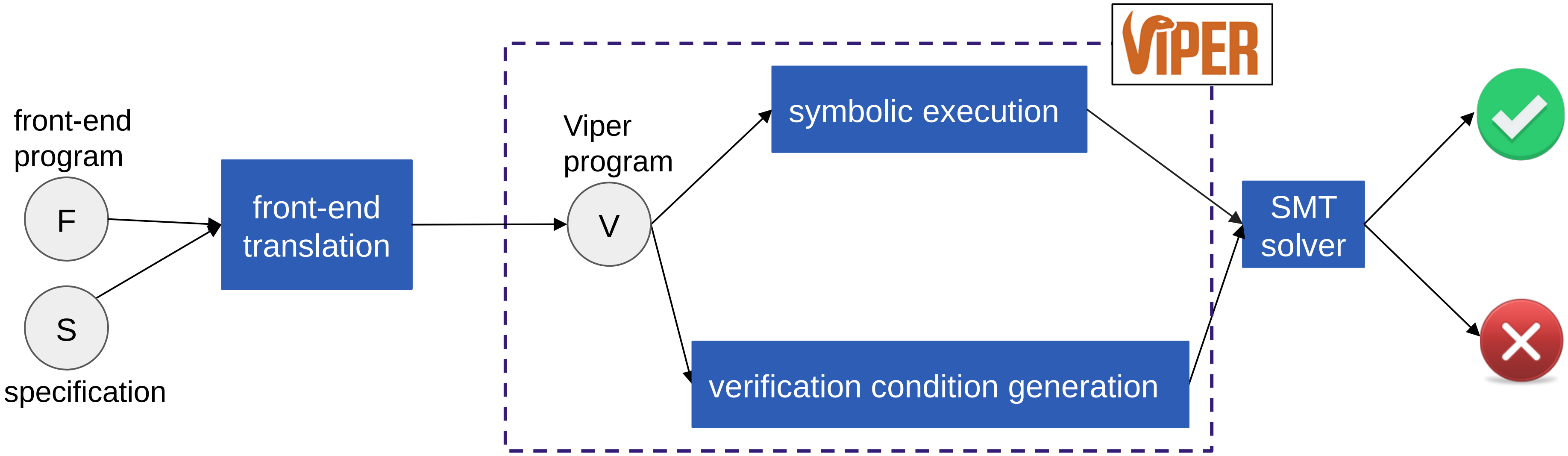
The Viper Verification Framework



Program verifiers built on top of Viper

Rust (*Prusti*)	Go (*Gobra*)	Python (*Nagini*)
Java, C, OpenCL, OpenMP (*VerCors*)	Smart contracts	RSL, FSL, FSL++
Secure information flow	Gradual verification	…

The Viper Verification Framework



Program verifiers built on top of Viper

Verification of the SCION Internet architecture (existing router implementation ~5k LOC)

Rust (*Prusti*)	Go (*Gobra*)	Python (*Nagini*)
Java, C, OpenCL, OpenMP (*VerCors*)	Smart contracts	RSL, FSL, FSL++
Secure information flow	Gradual verification	…

Overview of the Viper Language

Program Code	Assertion Language
Verification Features	**Mathematical Background**

Overview of the Viper Language

Program Code	Assertion Language
• Sequential, imperative language • Standard control structures • Basic type system • Built-in heap	
Verification Features	**Mathematical Background**

Overview of the Viper Language

Program Code	**Assertion Language**
• Sequential, imperative language • Standard control structures • Basic type system • Built-in heap	• Fractional permissions
Verification Features	**Mathematical Background**

Overview of the Viper Language

Program Code	Assertion Language
• Sequential, imperative language • Standard control structures • Basic type system • Built-in heap	• Fractional permissions • Inductive predicates
Verification Features	**Mathematical Background**

Overview of the Viper Language

Program Code	Assertion Language
• Sequential, imperative language • Standard control structures • Basic type system • Built-in heap	• Fractional permissions • Inductive predicates • Iterated separating conjunction
Verification Features	**Mathematical Background**

Overview of the Viper Language

Program Code	Assertion Language
• Sequential, imperative language • Standard control structures • Basic type system • Built-in heap	• Fractional permissions • Inductive predicates • Iterated separating conjunction • Magic wands
Verification Features	**Mathematical Background**

Overview of the Viper Language

Program Code	**Assertion Language**
• Sequential, imperative language • Standard control structures • Basic type system • Built-in heap	• Fractional permissions • Inductive predicates • Iterated separating conjunction • Magic wands • …
Verification Features	**Mathematical Background**

Overview of the Viper Language

Program Code	**Assertion Language**
• Sequential, imperative language • Standard control structures • Basic type system • Built-in heap	• Fractional permissions • Inductive predicates • Iterated separating conjunction • Magic wands • …
Verification Features	**Mathematical Background**
• Standard contract features • Inhale and exhale • …	

Overview of the Viper Language

Program Code	Assertion Language
• Sequential, imperative language • Standard control structures • Basic type system • Built-in heap	• Fractional permissions • Inductive predicates • Iterated separating conjunction • Magic wands • …
Verification Features	**Mathematical Background**
• Standard contract features • Inhale and exhale • …	• Predefined and user-defined datatypes • Uninterpreted functions • Axioms

Outline of the Talk

1. Overview of Viper

2. Inhale and Exhale: An Operational View of Separation Logic

3. Designed for Automation

4. Toward a Foundational Viper

Verification Primitives: Inhale and Exhale

Verification Primitives: Inhale and Exhale

inhale A

exhale A

Verification Primitives: Inhale and Exhale

SL Assertion

inhale A

exhale A

Verification Primitives: Inhale and Exhale

SL Assertion

inhale A

exhale A

Adds resources specified by A to the current context

Verification Primitives: Inhale and Exhale

SL Assertion

inhale A

exhale A

Adds resources specified by A to the current context

Removes resources specified by A from the current context

Verification Primitives: Inhale and Exhale

	inhale A	**exhale** A
Intuitive meaning	Adds resources specified by A to the current context	Removes resources specified by A from the current context
Logically		
Operationally		

Verification Primitives: Inhale and Exhale

	inhale A	**exhale** A
Intuitive meaning	Adds resources specified by A to the current context	Removes resources specified by A from the current context
Logically	⊢ {P} **inhale** A {P ∗ A}	
Operationally		

Verification Primitives: Inhale and Exhale

	inhale A	**exhale** A
Intuitive meaning	Adds resources specified by A to the current context	Removes resources specified by A from the current context
Logically	⊢ {P} **inhale** A {P ∗ A}	
Operationally		

Verification Primitives: Inhale and Exhale

	inhale A	**exhale** A
Intuitive meaning	Adds resources specified by A to the current context	Removes resources specified by A from the current context
Logically	$\vdash \{P\}$ **inhale** A $\{P \ast A\}$ wp (**inhale** A) $\{Q\} = A \mathrel{-\!\!\ast} Q$	
Operationally		

Verification Primitives: Inhale and Exhale

	inhale A	**exhale** A
Intuitive meaning	Adds resources specified by A to the current context	Removes resources specified by A from the current context
Logically	⊢ {P} **inhale** A {P ∗ A} wp (**inhale** A) {Q} = A −∗ Q	⊢ {P ∗ A} **exhale** A {P}
Operationally		

Verification Primitives: Inhale and Exhale

	inhale A	**exhale** A
Intuitive meaning	Adds resources specified by A to the current context	Removes resources specified by A from the current context
Logically	⊦ {P} **inhale** A {P ∗ A} wp (**inhale** A) {Q} = A −∗ Q	⊦ {P ∗ A} **exhale** A {P}
Operationally		

Verification Primitives: Inhale and Exhale

	inhale A	**exhale** A
Intuitive meaning	Adds resources specified by A to the current context	Removes resources specified by A from the current context
Logically	$\vdash$ {P} **inhale** A {P ∗ A} wp (**inhale** A) {Q} = A −∗ Q	$\vdash$ {P ∗ A} **exhale** A {P} wp (**exhale** A) {Q} = A ∗ Q
Operationally		

Verification Primitives: Inhale and Exhale

	inhale A	**exhale** A
Intuitive meaning	Adds resources specified by A to the current context	Removes resources specified by A from the current context
Logically	⊢ {P} **inhale** A {P ∗ A} wp (**inhale** A) {Q} = A −∗ Q	⊢ {P ∗ A} **exhale** A {P} wp (**exhale** A) {Q} = A ∗ Q
Operationally		

Acting on a SL state
(e.g., *Loc* ⇀ *(0, 1]* × *Val*)

Verification Primitives: Inhale and Exhale

	inhale A	**exhale** A
Intuitive meaning	Adds resources specified by A to the current context	Removes resources specified by A from the current context
Logically	$\vdash$ {P} **inhale** A {P $\ast$ A} wp (**inhale** A) {Q} = A $-\!\!\ast$ Q	$\vdash$ {P $\ast$ A} **exhale** A {P} wp (**exhale** A) {Q} = A $\ast$ Q
Operationally	• All resources required by A are obtained • All logical constraints are assumed	

Acting on a SL state
(e.g., *Loc* ⇀ *(0, 1]* × *Val*)

Verification Primitives: Inhale and Exhale

	inhale A	**exhale** A
Intuitive meaning	Adds resources specified by A to the current context	Removes resources specified by A from the current context
Logically	$\vdash \{P\}$ **inhale** A $\{P \ast A\}$ wp (**inhale** A) $\{Q\} = A \mathrel{-\!\!\ast} Q$	$\vdash \{P \ast A\}$ **exhale** A $\{P\}$ wp (**exhale** A) $\{Q\} = A \ast Q$
Operationally	• All resources required by A are obtained • All logical constraints are assumed	• All resources required by A are removed • All logical constraints are asserted

Acting on a SL state
(e.g., $Loc \rightharpoonup (0, 1] \times Val$)

Verification Primitives: Inhale and Exhale

	inhale A	**exhale** A
Intuitive meaning	Adds resources specified by A to the current context	Removes resources specified by A from the current context
Logically	⊢ {P} **inhale** A {P ∗ A} wp (**inhale** A) {Q} = A −∗ Q	⊢ {P ∗ A} **exhale** A {P} wp (**exhale** A) {Q} = A ∗ Q
Operationally	• All resources required by A are obtained • All logical constraints are assumed	• All resources required by A are removed • All logical constraints are asserted
SL analogue of	**assume** A	**assert** A

Verification Primitives: Inhale and Exhale

"A Basis for Verifying Multi-Threaded Programs" (Leino and Müller, ESOP 2009)

	inhale A	**exhale** A
Intuitive meaning	Adds resources specified by A to the current context	Removes resources specified by A from the current context
Logically	$\vdash \{P\}$ **inhale** A $\{P \ast A\}$ wp (**inhale** A) $\{Q\} = A \mathrel{-\!\!\ast} Q$	$\vdash \{P \ast A\}$ **exhale** A $\{P\}$ wp (**exhale** A) $\{Q\} = A \ast Q$
Operationally	• All resources required by A are obtained • All logical constraints are assumed	• All resources required by A are removed • All logical constraints are asserted
SL analogue of	**assume** A	**assert** A

Verification Primitives: Inhale and Exhale

“A Basis for Verifying Multi-Threaded Programs” (Leino and Müller, ESOP 2009)

Sometimes called **produce**

Sometimes called **consume**

	inhale A	**exhale** A
Intuitive meaning	Adds resources specified by A to the current context	Removes resources specified by A from the current context
Logically	$\vdash$ {P} **inhale** A {P ∗ A} wp (**inhale** A) {Q} = A −∗ Q	$\vdash$ {P ∗ A} **exhale** A {P} wp (**exhale** A) {Q} = A ∗ Q
Operationally	• All resources required by A are obtained • All logical constraints are assumed	• All resources required by A are removed • All logical constraints are asserted
SL analogue of	**assume** A	**assert** A

Example: Verifying a Parallel Composition (1/2)

Front-end language

```
q := new PointXY()

v := p.x      ||   q.y := p.x
q.x := v

free(p)
```

Example: Verifying a Parallel Composition (1/2)

Front-end language

```
q := new PointXY()

v := p.x      ||   q.y := p.x
q.x := v

free(p)
```

Object with fields *x* and *y*

Example: Verifying a Parallel Composition (1/2)

Front-end language

$\{\ p.x \mapsto \_ \ast p.y \mapsto \_\ \}$

q := **new** PointXY()

Object with fields *x* and *y*

v := p.x q.x := v	‖	q.y := p.x

free(p)

$\{\ q.x \mapsto v \ast q.y \mapsto v\ \}$

Example: Verifying a Parallel Composition (1/2)

Front-end language

$\{\ p.x \mapsto \_ \ast p.y \mapsto \_\ \}$

q := **new** PointXY()

Object with fields *x* and *y*

$\{ P_1 \}$	$\Vert$	$\{ P_2 \}$
v := p.x		
q.x := v		q.y := p.x
$\{ Q_1 \}$		$\{ Q_2 \}$

free(p)

$\{\ q.x \mapsto v \ast q.y \mapsto v\ \}$

Example: Verifying a Parallel Composition (1/2)

Front-end language

{ p.x ↦ _ ∗ p.y ↦ _ }
q := **new** PointXY()

Object with fields *x* and *y*

{ P_1 }	‖	{ P_2 }
v := p.x		
q.x := v		q.y := p.x
{ Q_1 }		{ Q_2 }

free(p)
{ q.x ↦ v ∗ q.y ↦ v }

$P_1 \triangleq (q.x \mapsto \_ \ast p.x \overset{1/2}{\mapsto} \_)$ $\quad$ $P_2 \triangleq (q.y \mapsto \_ \ast p.x \overset{1/2}{\mapsto} \_)$

Example: Verifying a Parallel Composition (1/2)

Front-end language

{ p.x ↦ _ ∗ p.y ↦ _ }
q := **new** PointXY()

Object with fields *x* and *y*

{ P_1 }	‖	{ P_2 }
v := p.x		
q.x := v		q.y := p.x
{ Q_1 }		{ Q_2 }

free(p)
{ q.x ↦ v ∗ q.y ↦ v }

$P_1 \triangleq (q.x \mapsto \_ \ast p.x \overset{1/2}{\mapsto} \_)$ $P_2 \triangleq (q.y \mapsto \_ \ast p.x \overset{1/2}{\mapsto} \_)$

$Q_1 \triangleq (q.x \mapsto v \ast p.x \overset{1/2}{\mapsto} v)$

Example: Verifying a Parallel Composition (1/2)

Front-end language

$\{\, p.x \mapsto \_ \ast p.y \mapsto \_ \,\}$

q := **new** PointXY()

Object with fields *x* and *y*

$\{\, P_1 \,\}$	‖	$\{\, P_2 \,\}$
v := p.x		
q.x := v		q.y := p.x
$\{\, Q_1 \,\}$		$\{\, Q_2 \,\}$

free(p)

$\{\, q.x \mapsto v \ast q.y \mapsto v \,\}$

$P_1 \triangleq (q.x \mapsto \_ \ast p.x \overset{½}{\mapsto} \_)$ $\quad P_2 \triangleq (q.y \mapsto \_ \ast p.x \overset{½}{\mapsto} \_)$

$Q_1 \triangleq (q.x \mapsto v \ast p.x \overset{½}{\mapsto} v)$ $\quad Q_2 \triangleq (\exists k.\ q.y \mapsto k \ast p.x \overset{½}{\mapsto} k)$

Example: Verifying a Parallel Composition (1/2)

Front-end language

{ p.x ↦ _ ∗ p.y ↦ _ }
q := **new** PointXY()

{ P_1 }	‖	{ P_2 }
v := p.x		
q.x := v		q.y := p.x
{ Q_1 }		{ Q_2 }

free(p)
{ q.x ↦ v ∗ q.y ↦ v }

$P_1 \triangleq (q.x \mapsto \_ \ast p.x \overset{1/2}{\mapsto} \_)$ $P_2 \triangleq (q.y \mapsto \_ \ast p.x \overset{1/2}{\mapsto} \_)$

$Q_1 \triangleq (q.x \mapsto v \ast p.x \overset{1/2}{\mapsto} v)$ $Q_2 \triangleq (\exists k.\ q.y \mapsto k \ast p.x \overset{1/2}{\mapsto} k)$

Example: Verifying a Parallel Composition (1/2)

Front-end language

$\{\ p.x \mapsto \_ \ast p.y \mapsto \_\ \}$

q := **new** PointXY()

$\{ P_1 \}$	‖	$\{ P_2 \}$
v := p.x		
q.x := v		q.y := p.x
$\{ Q_1 \}$		$\{ Q_2 \}$

free(p)

$\{\ q.x \mapsto v \ast q.y \mapsto v\ \}$

Starting in a state with no resources

VIPER

$P_1 \triangleq (q.x \mapsto \_ \ast p.x \overset{½}{\mapsto} \_)$ $\quad P_2 \triangleq (q.y \mapsto \_ \ast p.x \overset{½}{\mapsto} \_)$

$Q_1 \triangleq (q.x \mapsto v \ast p.x \overset{½}{\mapsto} v)$ $\quad Q_2 \triangleq (\exists k.\ q.y \mapsto k \ast p.x \overset{½}{\mapsto} k)$

Example: Verifying a Parallel Composition (1/2)

Front-end language

{ p.x ↦ _ ∗ p.y ↦ _ }
q := **new** PointXY()

{ P_1 }	‖	{ P_2 }
v := p.x		
q.x := v		q.y := p.x
{ Q_1 }		{ Q_2 }

free(p)
{ q.x ↦ v ∗ q.y ↦ v }

Starting in a state with no resources

VIPER

inhale p.x ↦ _ ∗ p.y ↦ _

$P_1 \triangleq (q.x \mapsto \_ \ast p.x \overset{1/2}{\mapsto} \_)$ $\quad P_2 \triangleq (q.y \mapsto \_ \ast p.x \overset{1/2}{\mapsto} \_)$

$Q_1 \triangleq (q.x \mapsto v \ast p.x \overset{1/2}{\mapsto} v)$ $\quad Q_2 \triangleq (\exists k.\ q.y \mapsto k \ast p.x \overset{1/2}{\mapsto} k)$

Example: Verifying a Parallel Composition (1/2)

Front-end language

{ p.x ↦ _ ∗ p.y ↦ _ }
q := **new** PointXY()

{ P_1 }	‖	{ P_2 }
v := p.x		
q.x := v		q.y := p.x
{ Q_1 }		{ Q_2 }

free(p)
{ q.x ↦ v ∗ q.y ↦ v }

(not actual Viper syntax)

Starting in a state with no resources

VIPER

inhale p.x ↦ _ ∗ p.y ↦ _

$P_1 \triangleq (q.x \mapsto \_ \ast p.x \overset{½}{\mapsto} \_)$ $P_2 \triangleq (q.y \mapsto \_ \ast p.x \overset{½}{\mapsto} \_)$

$Q_1 \triangleq (q.x \mapsto v \ast p.x \overset{½}{\mapsto} v)$ $Q_2 \triangleq (\exists k.\ q.y \mapsto k \ast p.x \overset{½}{\mapsto} k)$

Example: Verifying a Parallel Composition (1/2)

Front-end language

$\{ p.x \mapsto \_ \ast p.y \mapsto \_ \}$
q := **new** PointXY()

$\{ P_1 \}$	$\parallel$	$\{ P_2 \}$
v := p.x		
q.x := v		q.y := p.x
$\{ Q_1 \}$		$\{ Q_2 \}$

free(p)
$\{ q.x \mapsto v \ast q.y \mapsto v \}$

(not actual Viper syntax)

Starting in a state with no resources

inhale $p.x \mapsto \_ \ast p.y \mapsto \_$

havoc q
inhale $q.x \mapsto \_ \ast q.y \mapsto \_$

$P_1 \triangleq (q.x \mapsto \_ \ast p.x \overset{½}{\mapsto} \_)$ $P_2 \triangleq (q.y \mapsto \_ \ast p.x \overset{½}{\mapsto} \_)$

$Q_1 \triangleq (q.x \mapsto v \ast p.x \overset{½}{\mapsto} v)$ $Q_2 \triangleq (\exists k.\ q.y \mapsto k \ast p.x \overset{½}{\mapsto} k)$

Example: Verifying a Parallel Composition (1/2)

Front-end language

{ p.x ↦ _ ∗ p.y ↦ _ }
q := **new** PointXY()

$\{ P_1 \}$	‖	$\{ P_2 \}$
v := p.x		
q.x := v		q.y := p.x
$\{ Q_1 \}$		$\{ Q_2 \}$

free(p)
{ q.x ↦ v ∗ q.y ↦ v }

Starting in a state with no resources

VIPER

inhale p.x ↦ ∗ p.y ↦

Non-deterministic assignment

havoc q
inhale q.x ↦ _ ∗ q.y ↦ _

$P_1 \triangleq (q.x \mapsto \_ \ast p.x \overset{½}{\mapsto} \_)$ $P_2 \triangleq (q.y \mapsto \_ \ast p.x \overset{½}{\mapsto} \_)$

$Q_1 \triangleq (q.x \mapsto v \ast p.x \overset{½}{\mapsto} v)$ $Q_2 \triangleq (\exists k.\ q.y \mapsto k \ast p.x \overset{½}{\mapsto} k)$

Example: Verifying a Parallel Composition (1/2)

Front-end language

{ p.x ↦ _ ∗ p.y ↦ _ }
q := **new** PointXY()

{ P_1 }	‖	{ P_2 }
v := p.x		
q.x := v		q.y := p.x
{ Q_1 }		{ Q_2 }

free(p)
{ q.x ↦ v ∗ q.y ↦ v }

$$\frac{q \not\in fv(A)}{\vdash \{A\}\ q := \mathbf{new}\ PointXY()\ \{A \ast q.x \mapsto \_ \ast q.y \mapsto \_\}}$$

Starting in a state with no resources

inhale p.x ↦ ∗ p.y ↦

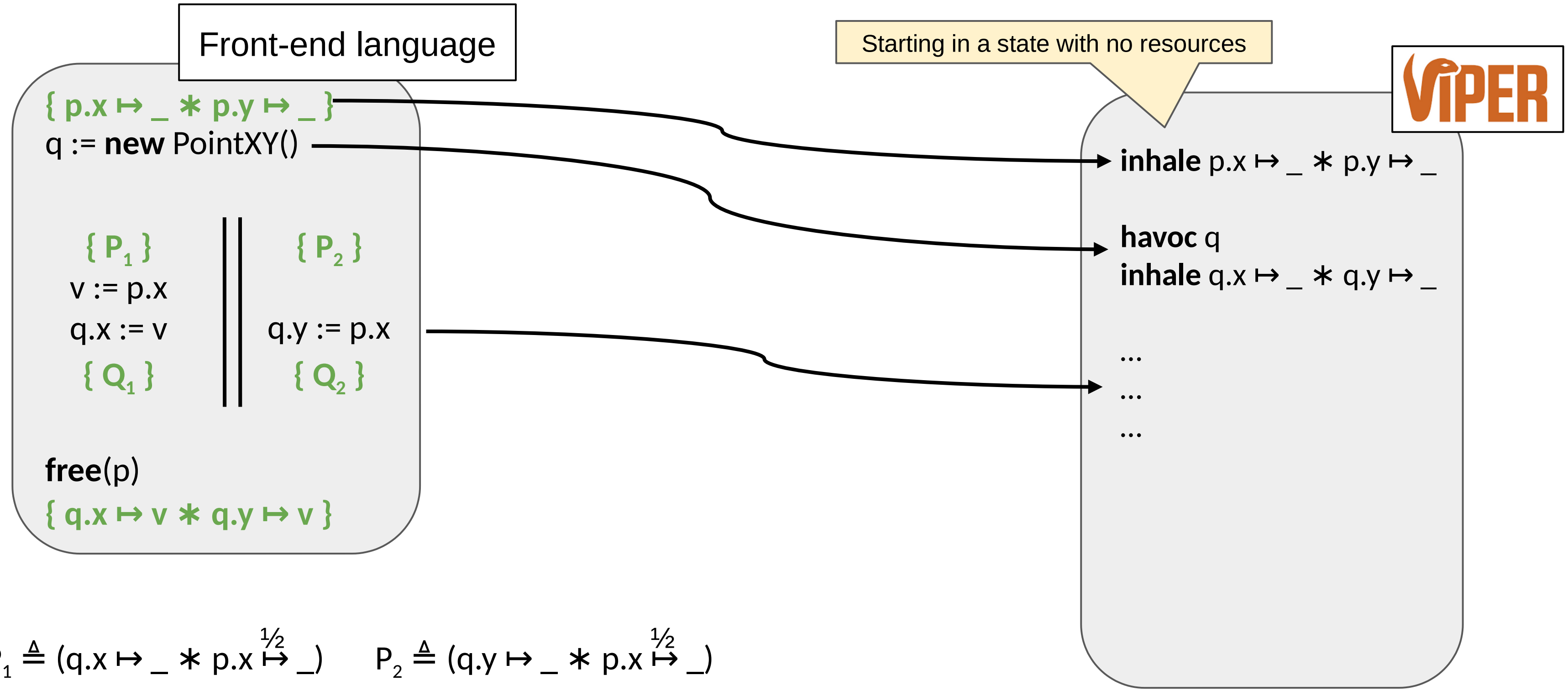
Non-deterministic assignment

havoc q
inhale q.x ↦ _ ∗ q.y ↦ _

$P_1 \triangleq (q.x \mapsto \_ \ast p.x \overset{½}{\mapsto} \_)$ $P_2 \triangleq (q.y \mapsto \_ \ast p.x \overset{½}{\mapsto} \_)$

$Q_1 \triangleq (q.x \mapsto v \ast p.x \overset{½}{\mapsto} v)$ $Q_2 \triangleq (\exists k.\ q.y \mapsto k \ast p.x \overset{½}{\mapsto} k)$

Example: Verifying a Parallel Composition (1/2)



$P_1 \triangleq (q.x \mapsto \_ \ast p.x \overset{1/2}{\mapsto} \_)$ $\quad P_2 \triangleq (q.y \mapsto \_ \ast p.x \overset{1/2}{\mapsto} \_)$

$Q_1 \triangleq (q.x \mapsto v \ast p.x \overset{1/2}{\mapsto} v)$ $\quad Q_2 \triangleq (\exists k.\ q.y \mapsto k \ast p.x \overset{1/2}{\mapsto} k)$

Example: Verifying a Parallel Composition (1/2)

Front-end language

$\{ p.x \mapsto \_ * p.y \mapsto \_ \}$

q := **new** PointXY()

$\{ P_1 \}$	$\parallel$	$\{ P_2 \}$
v := p.x		
q.x := v		q.y := p.x
$\{ Q_1 \}$		$\{ Q_2 \}$

free(p)

$\{ q.x \mapsto v * q.y \mapsto v \}$

Starting in a state with no resources

VIPER

inhale $p.x \mapsto \_ * p.y \mapsto \_$

havoc q

inhale $q.x \mapsto \_ * q.y \mapsto \_$

…

…

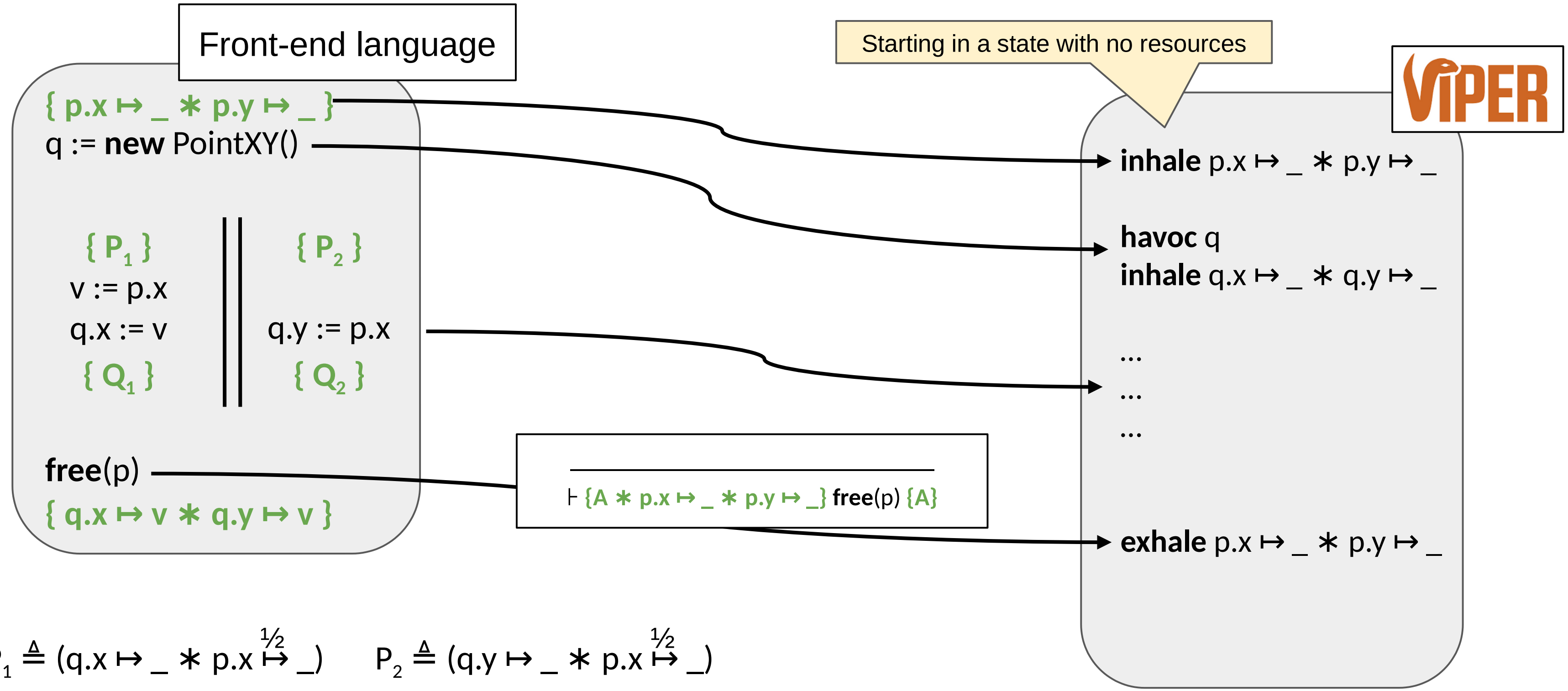
…

exhale $p.x \mapsto \_ * p.y \mapsto \_$

$P_1 \triangleq (q.x \mapsto \_ * p.x \overset{1/2}{\mapsto} \_)$ $\quad P_2 \triangleq (q.y \mapsto \_ * p.x \overset{1/2}{\mapsto} \_)$

$Q_1 \triangleq (q.x \mapsto v * p.x \overset{1/2}{\mapsto} v)$ $\quad Q_2 \triangleq (\exists k.\ q.y \mapsto k * p.x \overset{1/2}{\mapsto} k)$

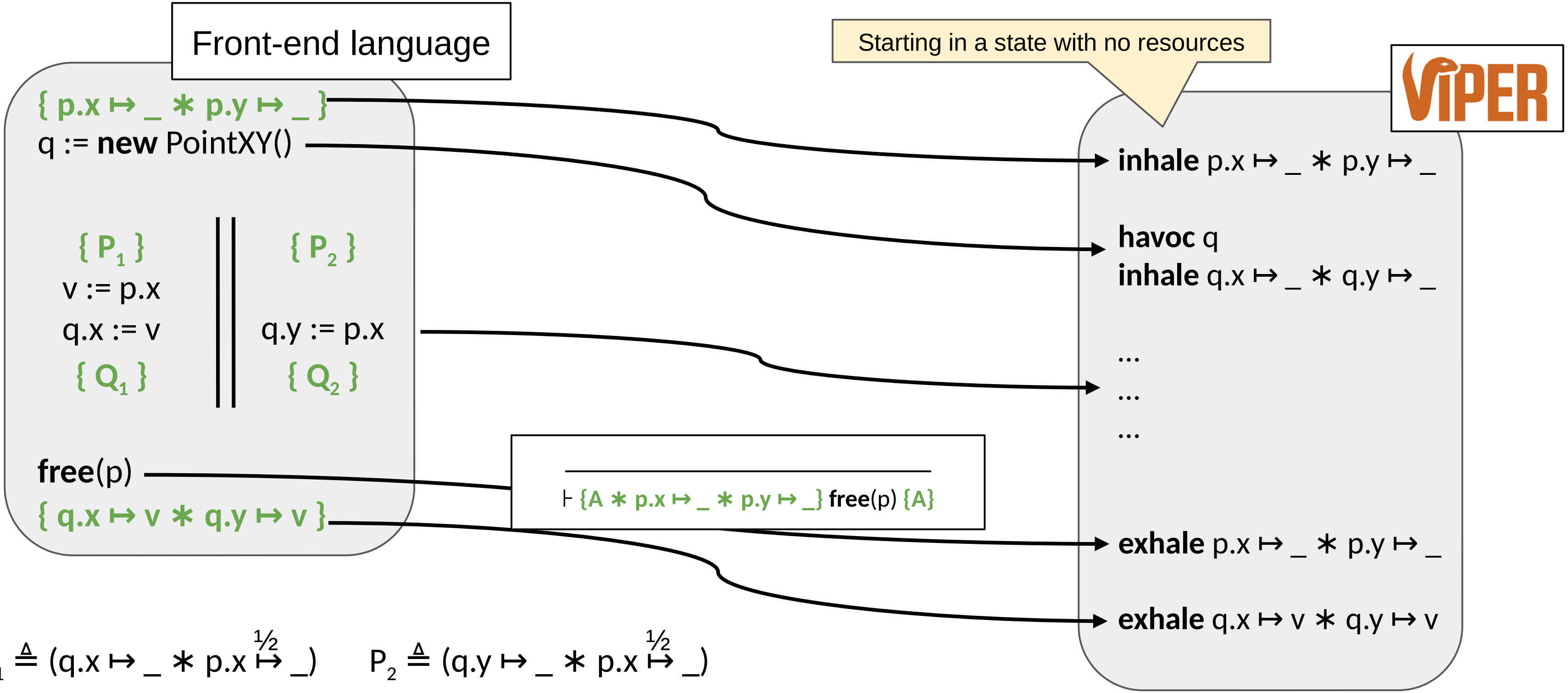
Example: Verifying a Parallel Composition (1/2)



$P_1 \triangleq (q.x \mapsto \_ * p.x \overset{1/2}{\mapsto} \_)$ $\quad P_2 \triangleq (q.y \mapsto \_ * p.x \overset{1/2}{\mapsto} \_)$

$Q_1 \triangleq (q.x \mapsto v * p.x \overset{1/2}{\mapsto} v)$ $\quad Q_2 \triangleq (\exists k.\ q.y \mapsto k * p.x \overset{1/2}{\mapsto} k)$

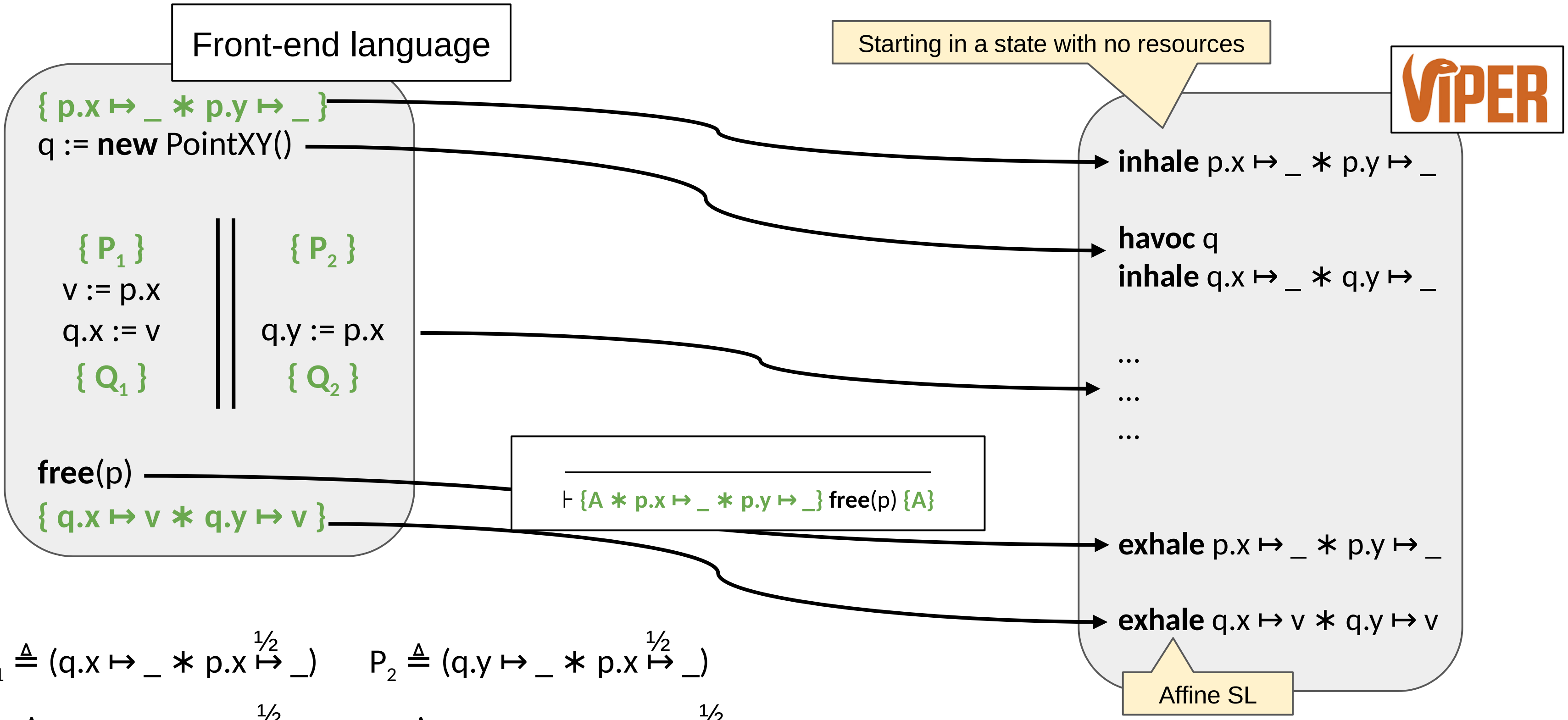
Example: Verifying a Parallel Composition (1/2)



$P_1 \triangleq (q.x \mapsto \_ \ast p.x \overset{1/2}{\mapsto} \_)$ $P_2 \triangleq (q.y \mapsto \_ \ast p.x \overset{1/2}{\mapsto} \_)$

$Q_1 \triangleq (q.x \mapsto v \ast p.x \overset{1/2}{\mapsto} v)$ $Q_2 \triangleq (\exists k.\ q.y \mapsto k \ast p.x \overset{1/2}{\mapsto} k)$

Example: Verifying a Parallel Composition (1/2)



$P_1 \triangleq (q.x \mapsto \_ * p.x \overset{1/2}{\mapsto} \_)$ $\quad$ $P_2 \triangleq (q.y \mapsto \_ * p.x \overset{1/2}{\mapsto} \_)$

$Q_1 \triangleq (q.x \mapsto v * p.x \overset{1/2}{\mapsto} v)$ $\quad$ $Q_2 \triangleq (\exists k.\ q.y \mapsto k * p.x \overset{1/2}{\mapsto} k)$

Example: Verifying a Parallel Composition (2/2)

Front-end language

{ p.x ↦ _ ∗ p.y ↦ _ }
q := **new** PointXY()

{ P_1 }	‖	{ P_2 }
v := p.x		
q.x := v		q.y := p.x
{ Q_1 }		{ Q_2 }

free(p)

{ q.x ↦ v ∗ q.y ↦ v }

VIPER

inhale p.x ↦ _ ∗ p.y ↦ _

havoc q
inhale q.x ↦ _ ∗ q.y ↦ _

…
…
…

exhale p.x ↦ _ ∗ p.y ↦ _

exhale q.x ↦ v ∗ q.y ↦ v

Example: Verifying a Parallel Composition (2/2)

Front-end language

{ p.x ↦ _ ∗ p.y ↦ _ }
q := **new** PointXY()

{ P_1 }	‖	{ P_2 }
v := p.x		
q.x := v		q.y := p.x
{ Q_1 }		{ Q_2 }

free(p)

{ q.x ↦ v ∗ q.y ↦ v }

VIPER

inhale p.x ↦ _ ∗ p.y ↦ _

havoc q
inhale q.x ↦ _ ∗ q.y ↦ _

…
…
…

exhale p.x ↦ _ ∗ p.y ↦ _

exhale q.x ↦ v ∗ q.y ↦ v

Example: Verifying a Parallel Composition (2/2)

Front-end language

{ p.x ↦ _ ∗ p.y ↦ _ }
q := **new** PointXY()

{ P_1 }	‖	{ P_2 }
v := p.x		
q.x := v		q.y := p.x
{ Q_1 }		{ Q_2 }

free(p)

{ q.x ↦ v ∗ q.y ↦ v }

$$\frac{\vdash \{P_1\}\ C_1\ \{Q_1\} \qquad \vdash \{P_2\}\ C_2\ \{Q_2\} \qquad fv(R) \cap wr(C_1) = \emptyset \qquad \ldots}{\vdash \{P_1 * P_2 * R\}\ C_1 \,||\, C_2\ \{Q_1 * Q_2 * R\}}$$

VIPER

inhale p.x ↦ _ ∗ p.y ↦ _

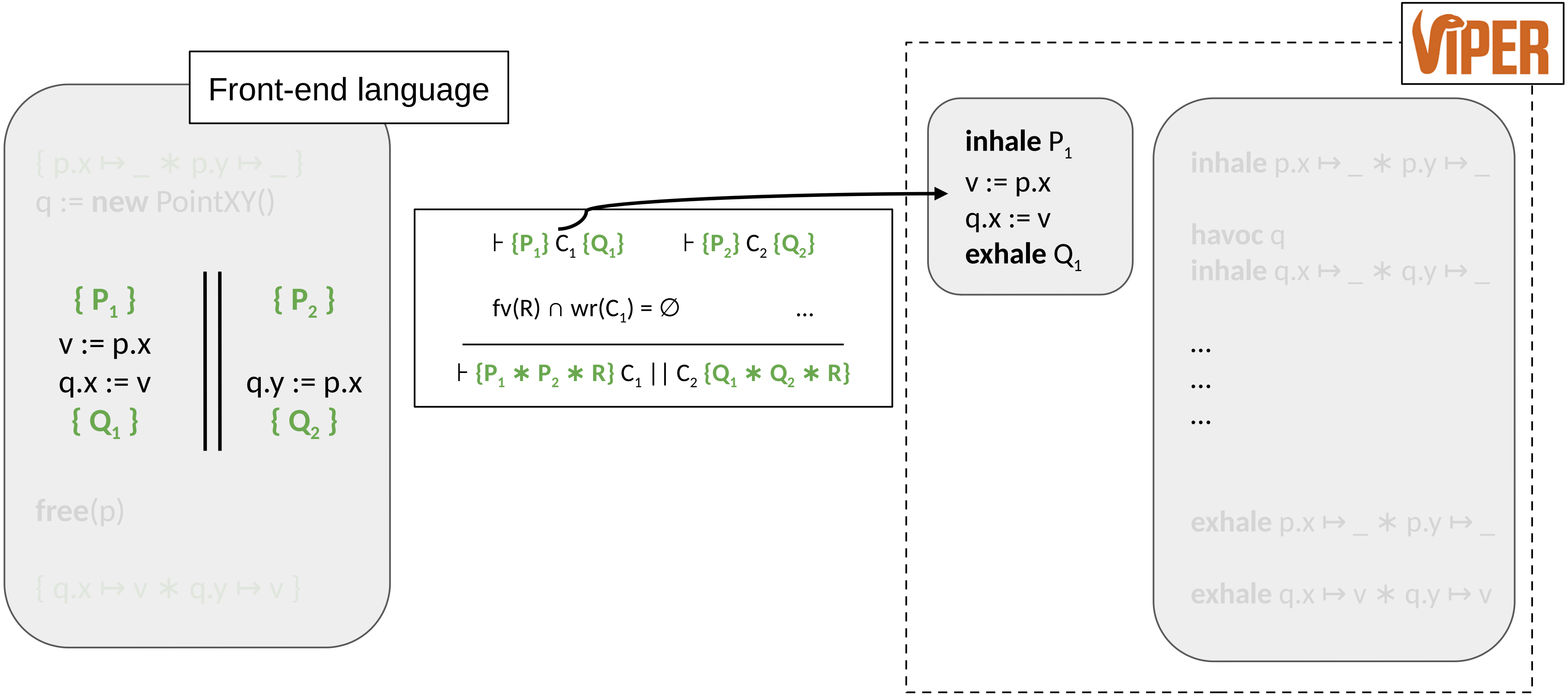
havoc q
inhale q.x ↦ _ ∗ q.y ↦ _

...
...
...

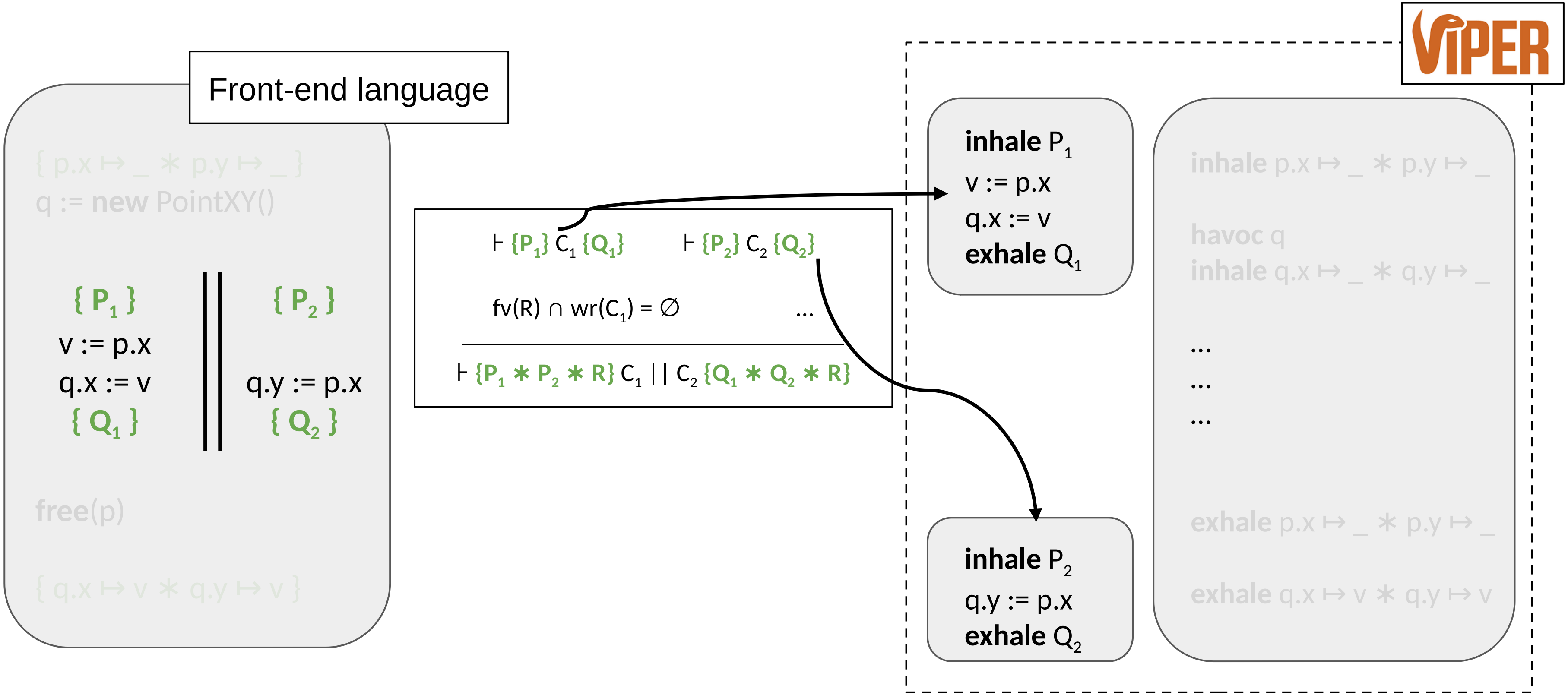
exhale p.x ↦ _ ∗ p.y ↦ _

exhale q.x ↦ v ∗ q.y ↦ v

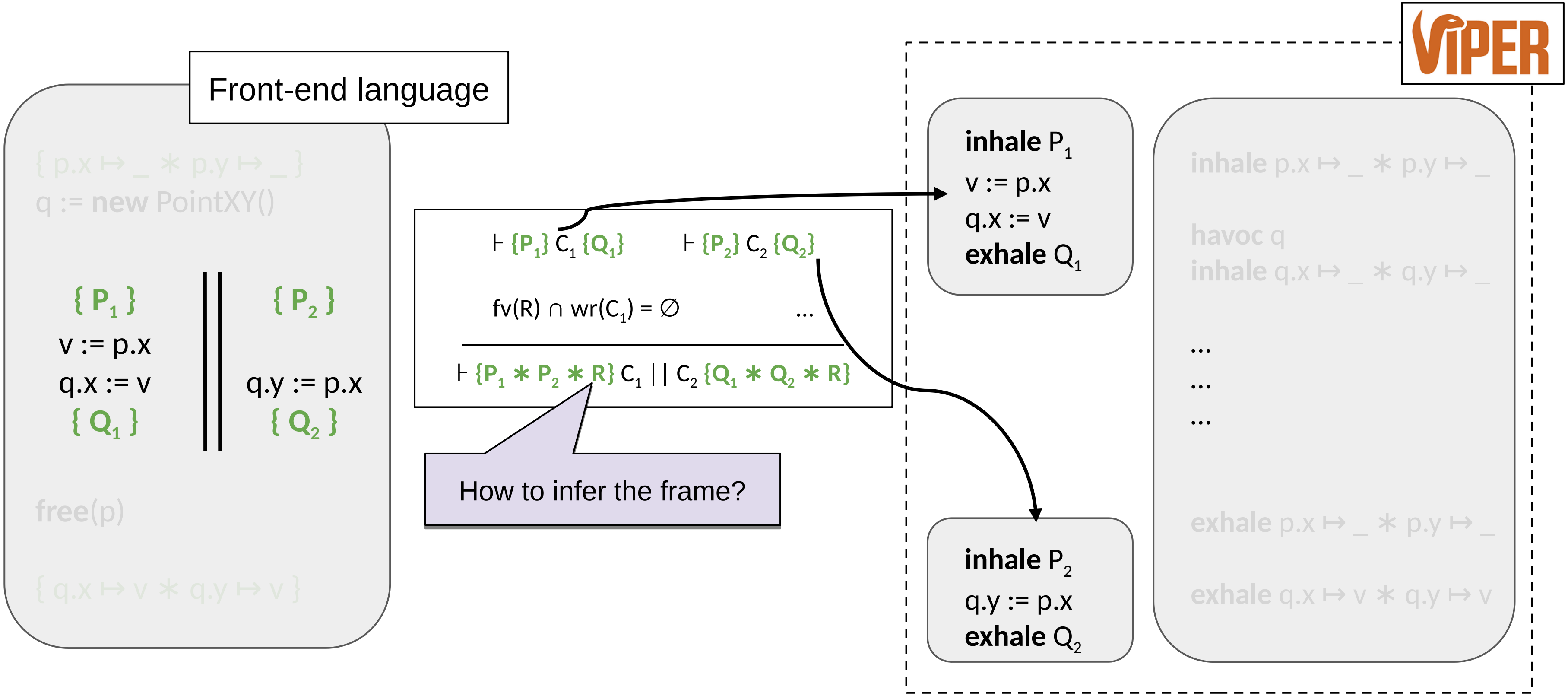
Example: Verifying a Parallel Composition (2/2)



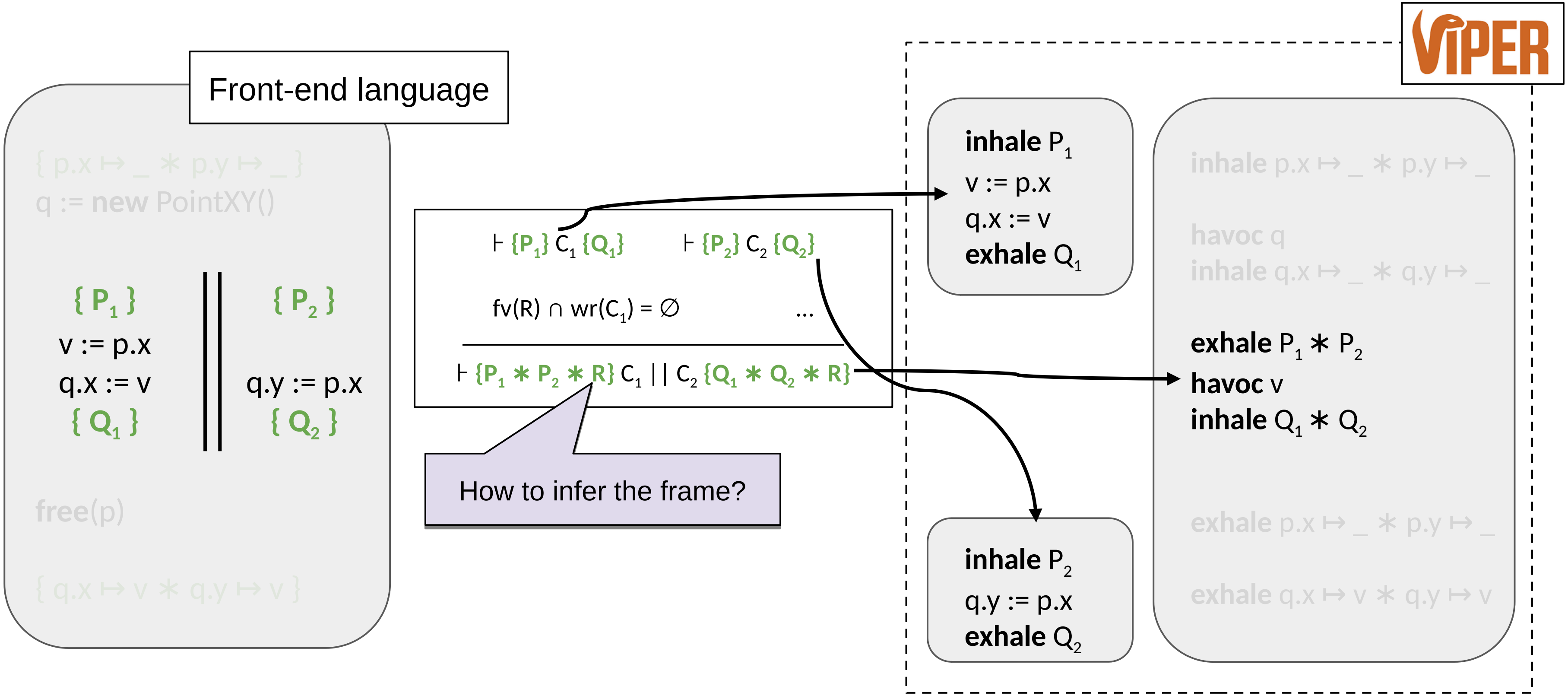
Example: Verifying a Parallel Composition (2/2)



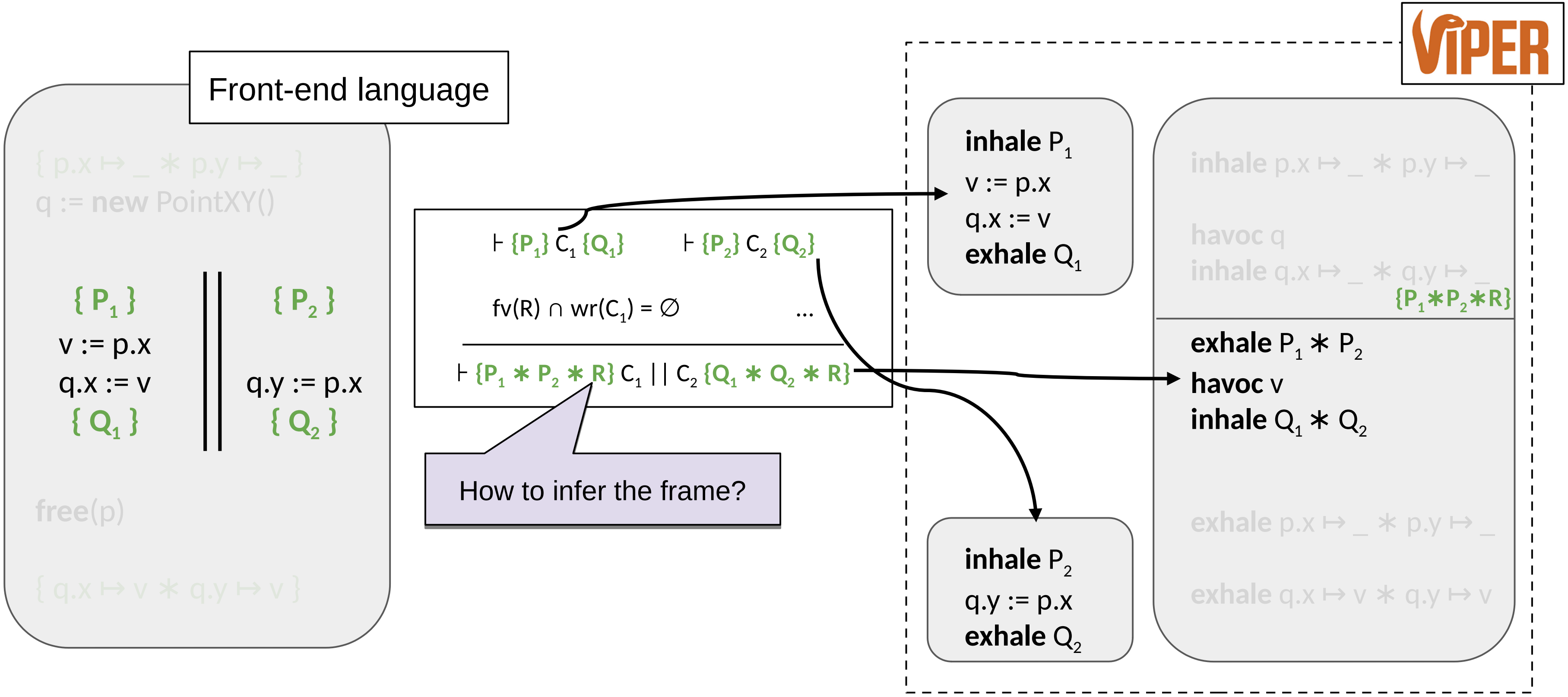
Example: Verifying a Parallel Composition (2/2)



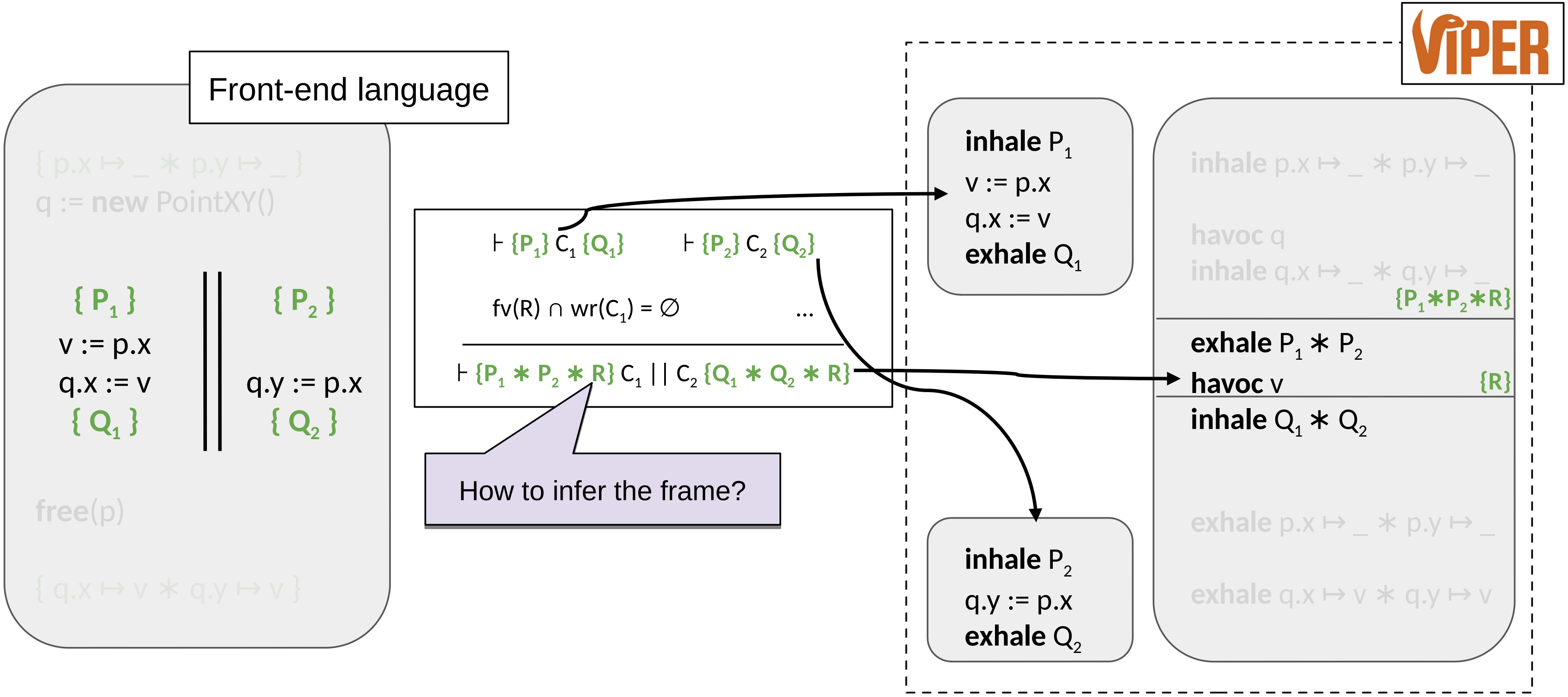
Example: Verifying a Parallel Composition (2/2)



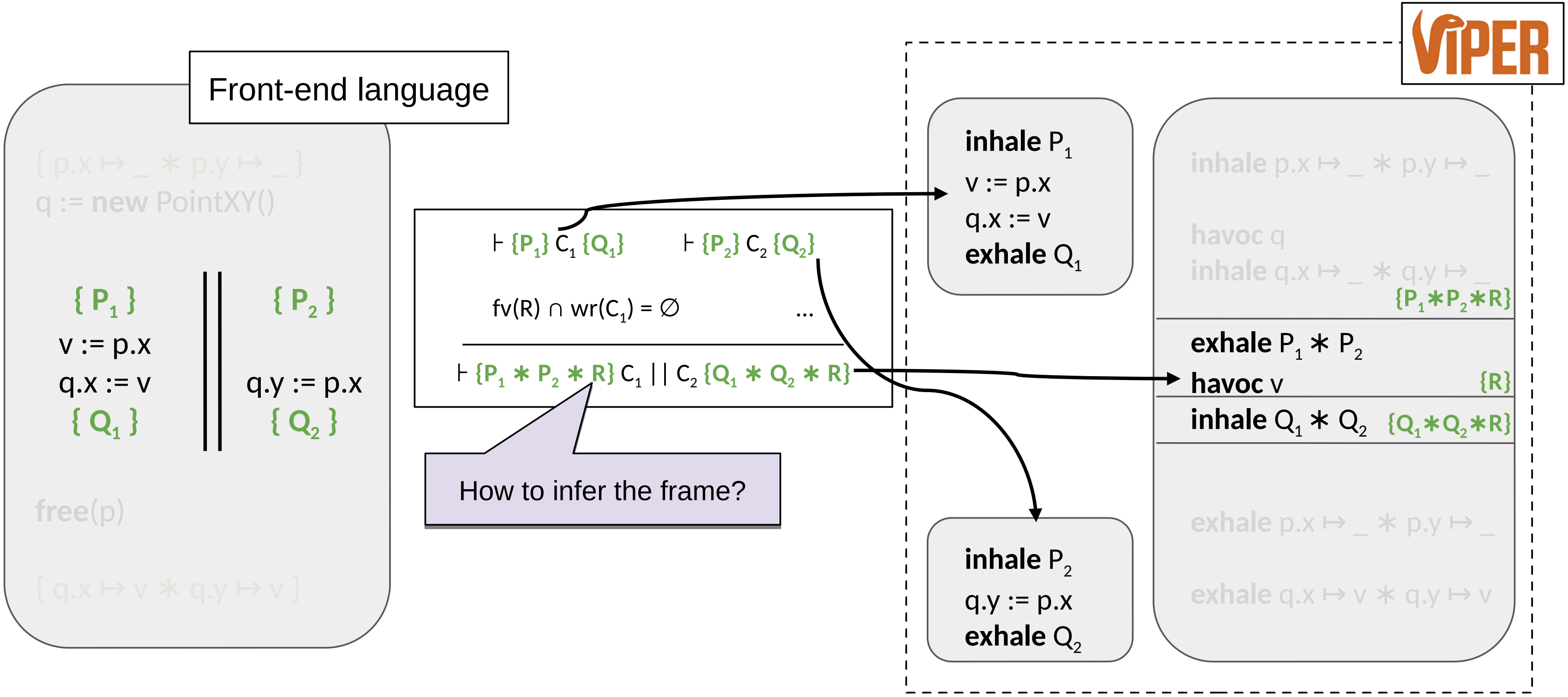
Example: Verifying a Parallel Composition (2/2)



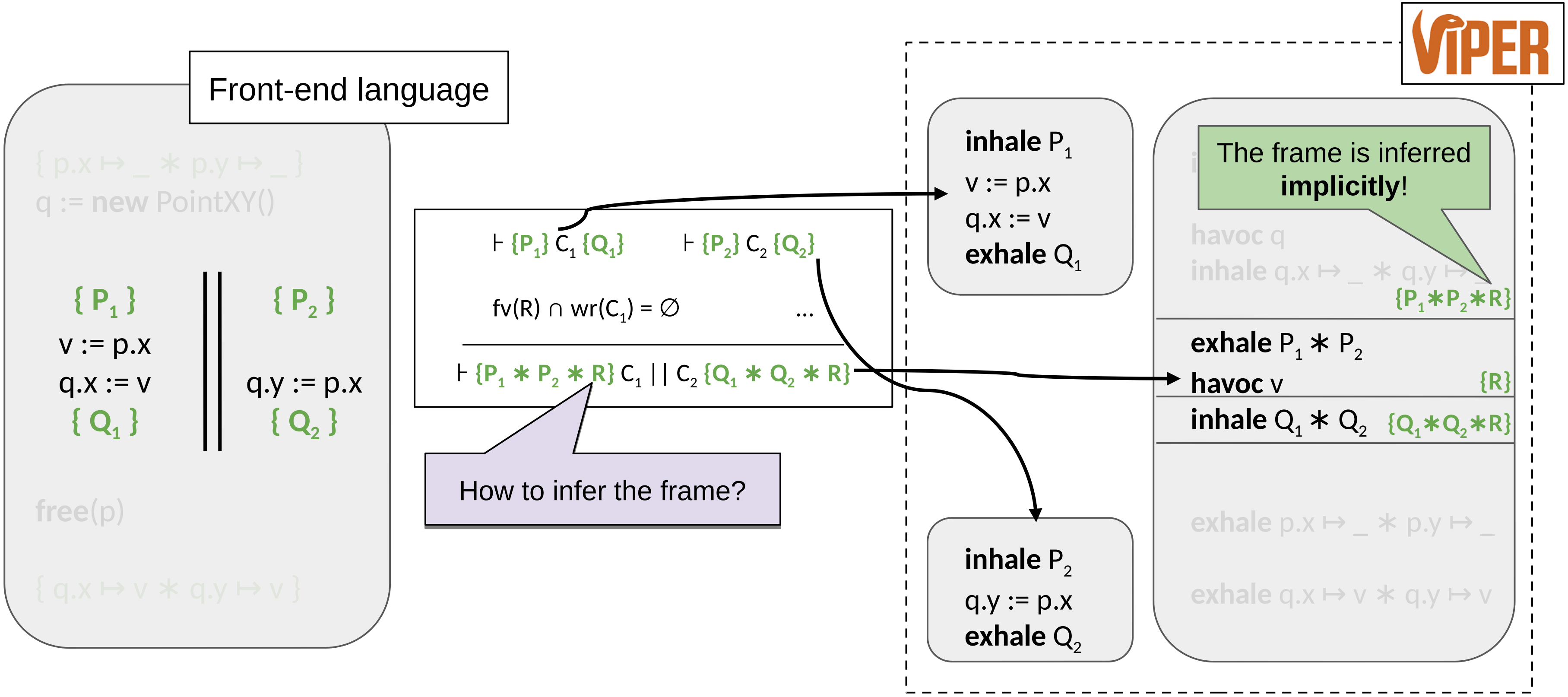
Example: Verifying a Parallel Composition (2/2)



Example: Verifying a Parallel Composition (2/2)



Example: Verifying a Parallel Composition (2/2)



Outline of the Talk

1. Overview of Viper

2. Inhale and Exhale: An Operational View of Separation Logic

3. Designed for Automation

4. Toward a Foundational Viper

Automating Separation Logic: Challenges

Automating Separation Logic: Challenges

Challenge 1

Existentials

Automating Separation Logic: Challenges

Challenge 1

Existentials

Challenge 2

Recursive predicates

Automating Separation Logic: Challenges

Challenge 1

Existentials

Challenge 2

Recursive predicates

Challenge 3

Magic wands

Automating Separation Logic: Challenges

Challenge 1

Existentials

Challenge 2

Recursive predicates

Challenge 3

Magic wands

Which resources to remove when exhaling $A -\!\!* \; B$?

Automating Separation Logic: Challenges

Challenge 1

Existentials

Challenge 2

Recursive predicates

Challenge 3

Magic wands

Which resources to remove when exhaling A −∗ B?

Challenge 4

Iterated separating conjunction

Automating Separation Logic: Challenges

Challenge 1

Existentials

Challenge 2

Recursive predicates

Challenge 3

Magic wands

Which resources to remove when exhaling A −∗ B?

Challenge 4

Iterated separating conjunction

…

Automating Separation Logic: Challenges

Challenge 1

Existentials

Challenge 2

Recursive predicates

Challenge 3

Magic wands

Which resources to remove when exhaling A −∗ B?

Challenge 4

Iterated separating conjunction

…

Challenge 1: Existentials

Challenge 1: Existentials

$$\textbf{exhale}\ \exists v.\ (p.x \overset{½}{\mapsto} v \ast v > 0)$$

Challenge 1: Existentials

exhale $\exists v.\ (p.x \overset{½}{\mapsto} v \ast v > 0)$

exhale $\exists s.\ list(l, s) \ast |s| > 0$

Challenge 1: Existentials

exhale $\exists v.\ (p.x \overset{½}{\mapsto} v \ast v > 0)$

exhale $\exists s.\ list(l, s) \ast |s| > 0$

Sequence of elements

Challenge 1: Existentials

Output parameters

exhale $\exists v. \; (p.x \overset{½}{\mapsto} v \ast v > 0)$

exhale $\exists s. \; list(l, s) \ast |s| > 0$

Sequence of elements

Challenge 1: Existentials

Output parameters

exhale $\exists v.\ (p.x \overset{½}{\mapsto} v \ast v > 0)$

exhale $\exists s.\ list(l, s) \ast |s| > 0$

exhale $\exists p.\ p.x \mapsto \_ \ast p.y \mapsto \_$

Challenge 1: Existentials

Output parameters

exhale ∃v. (p.x $\overset{½}{\mapsto}$ v ∗ v > 0)

exhale ∃s. list(l, s) ∗ |s| > 0

exhale ∃p. p.x ↦ _ ∗ p.y ↦ _

exhale ∃l. list(l, s) ∗ l ≠ null

Challenge 1: Existentials

Output parameters

exhale ∃v. (p.x $\overset{1/2}{\mapsto}$ v ∗ v > 0)

exhale ∃s. list(l, s) ∗ |s| > 0

Input parameters

exhale ∃p. p.x ↦ _ ∗ p.y ↦ _

exhale ∃l. list(l, s) ∗ l ≠ null

Challenge 1: Existentials

Output parameters

exhale ∃v. (p.x $\overset{½}{\mapsto}$ v ∗ v > 0)

exhale ∃s. list(l, s) ∗ |s| > 0

Input parameters

exhale ∃p. p.x ↦ _ ∗ p.y ↦ _

exhale ∃l. list(l, s) ∗ l ≠ null

Challenge 1: Existentials

❑ Avoid existentials in the SMT encoding

Output parameters

exhale ∃v. (p.x $\overset{½}{\mapsto}$ v ∗ v > 0)

exhale ∃s. list(l, s) ∗ |s| > 0

Input parameters

exhale ∃p. p.x ↦ _ ∗ p.y ↦ _

exhale ∃l. list(l, s) ∗ l ≠ null

Challenge 1: Existentials

- ❑ Avoid existentials in the SMT encoding
- ❑ Avoid backtracking

Output parameters

exhale ∃v. (p.x $\overset{½}{\mapsto}$ v ∗ v > 0)

exhale ∃s. list(l, s) ∗ |s| > 0

Input parameters

exhale ∃p. p.x ↦ _ ∗ p.y ↦ _

exhale ∃l. list(l, s) ∗ l ≠ null

Challenge 1: Existentials

- ❑ Avoid existentials in the SMT encoding
- ❑ Avoid backtracking
- ❑ Predictable automation

Output parameters

exhale ∃v. (p.x $\overset{½}{\mapsto}$ v ∗ v > 0)

exhale ∃s. list(l, s) ∗ |s| > 0

Input parameters

exhale ∃p. p.x ↦ _ ∗ p.y ↦ _

exhale ∃l. list(l, s) ∗ l ≠ null

Challenge 1: Existentials

- ❏ Avoid existentials in the SMT encoding
- ❏ Avoid backtracking
- ❏ Predictable automation

Output parameters

exhale ∃v. (p.x $\overset{½}{\mapsto}$ v ∗ v > 0)

exhale ∃s. list(l, s) ∗ |s| > 0

Input parameters

exhale ∃p. p.x ↦ _ ∗ p.y ↦ _

exhale ∃l. list(l, s) ∗ l ≠ null

Forbidden by Viper's syntax

Challenge 1: Existentials

- ❑ Avoid existentials in the SMT encoding
- ❑ Avoid backtracking
- ❑ Predictable automation

Output parameters

exhale ∃v. (p.x $\overset{½}{\mapsto}$ v ∗ v > 0)

exhale ∃s. list(l, s) ∗ |s| > 0

Input parameters

exhale ∃p. p.x ↦ _ ∗ p.y ↦ _

exhale ∃l. list(l, s) ∗ l ≠ null

Forbidden by Viper's syntax

Viper uses
implicit dynamic frames

Challenge 1: Existentials

- ❏ Avoid existentials in the SMT encoding
- ❏ Avoid backtracking
- ❏ Predictable automation

Output parameters

exhale ∃v. (p.x $\overset{½}{\mapsto}$ v ∗ v > 0)

exhale ∃s. list(l, s) ∗ |s| > 0

Input parameters

exhale ∃p. p.x ↦ _ ∗ p.y ↦ _

exhale ∃l. list(l, s) ∗ l ≠ null

Forbidden by Viper's syntax

Viper uses
implicit dynamic frames

≈ SL with **heap-dependent** expressions and functions

Challenge 1: Existentials

- ❑ Avoid existentials in the SMT encoding
- ❑ Avoid backtracking
- ❑ Predictable automation

Output parameters

exhale ∃v. (p.x $\overset{½}{\mapsto}$ v ∗ v > 0)

exhale ∃s. list(l, s) ∗ |s| > 0

is written as

exhale acc(p.x, 1/2) ∗ p.x > 0

Input parameters

exhale ∃p. p.x ↦ _ ∗ p.y ↦ _

exhale ∃l. list(l, s) ∗ l ≠ null

Forbidden by Viper's syntax

Viper uses
implicit dynamic frames

≈ SL with **heap-dependent** expressions and functions

Challenge 1: Existentials

- ❑ Avoid existentials in the SMT encoding
- ❑ Avoid backtracking
- ❑ Predictable automation

Output parameters

exhale ∃v. (p.x $\overset{½}{\mapsto}$ v ∗ v > 0)

exhale ∃s. list(l, s) ∗ |s| > 0

is written as →

exhale acc(p.x, 1/2) ∗ p.x > 0

Avoids existential

Input parameters

exhale ∃p. p.x ↦ _ ∗ p.y ↦ _

exhale ∃l. list(l, s) ∗ l ≠ null

Forbidden by Viper's syntax

Viper uses
implicit dynamic frames

≈ SL with **heap-dependent** expressions and functions

Challenge 1: Existentials

- ❑ Avoid existentials in the SMT encoding
- ❑ Avoid backtracking
- ❑ Predictable automation

Output parameters

exhale ∃v. (p.x $\overset{1/2}{\mapsto}$ v ∗ v > 0) — is written as → **exhale** acc(p.x, 1/2) ∗ p.x > 0 (Avoids existential)

exhale ∃s. list(l, s) ∗ |s| > 0 — is written as → **exhale** list(l) ∗ len(l) > 0

Input parameters

exhale ∃p. p.x ↦ _ ∗ p.y ↦ _

exhale ∃l. list(l, s) ∗ l ≠ null

Forbidden by Viper's syntax

Viper uses
implicit dynamic frames

≈ SL with **heap-dependent** expressions and functions

Challenge 1: Existentials

- ❑ Avoid existentials in the SMT encoding
- ❑ Avoid backtracking
- ❑ Predictable automation

Output parameters

exhale ∃v. (p.x $\overset{½}{\mapsto}$ v ∗ v > 0) — is written as → **exhale** acc(p.x, 1/2) ∗ p.x > 0 (Avoids existential)

exhale ∃s. list(l, s) ∗ |s| > 0 — is written as → **exhale** list(l) ∗ len(l) > 0 (Heap-dependent function)

Input parameters

exhale ∃p. p.x ↦ _ ∗ p.y ↦ _

exhale ∃l. list(l, s) ∗ l ≠ null

Forbidden by Viper's syntax

Viper uses **implicit dynamic frames**

≈ SL with **heap-dependent** expressions and functions

Challenge 1: Existentials

- ❑ Avoid existentials in the SMT encoding
- ❑ Avoid backtracking
- ❑ Predictable automation

Output parameters

exhale ∃v. (p.x $\overset{½}{\mapsto}$ v ∗ v > 0) — is written as → **exhale** acc(p.x, 1/2) ∗ p.x > 0

exhale ∃s. list(l, s) ∗ |s| > 0 — is written as → **exhale** list(l) ∗ len(l) > 0

Avoids existential

Heap-dependent function

Input parameters

exhale ∃p. p.x ↦ _ ∗ p.y ↦ _

exhale ∃l. list(l, s) ∗ l ≠ null

Forbidden by Viper's syntax

Viper uses
implicit dynamic frames

≈ SL with **heap-dependent** expressions and functions

Allows writing code and specifications in the same language

Challenge 1: Existentials

- ❑ Avoid existentials in the SMT encoding
- ❑ Avoid backtracking
- ❑ Predictable automation

Output parameters

exhale ∃v. (p.x $\overset{½}{\mapsto}$ v ∗ v > 0) — is written as → **exhale** acc(p.x, 1/2) ∗ p.x > 0

exhale ∃s. list(l, s) ∗ |s| > 0 — is written as → **exhale** list(l) ∗ len(l) > 0

Input parameters

exhale ∃p. p.x ↦ _ ∗ p.y ↦ _

exhale ∃l. list(l, s) ∗ l ≠ null

Forbidden by Viper's syntax

Viper uses
implicit dynamic frames

≈ SL with **heap-dependent** expressions and functions

Allows writing code and specifications in the same language

inhale A ∗ B
exhale A ∗ B

Challenge 1: Existentials

- ❑ Avoid existentials in the SMT encoding
- ❑ Avoid backtracking
- ❑ Predictable automation

Output parameters

exhale ∃v. (p.x $\overset{½}{\mapsto}$ v ∗ v > 0) — is written as → **exhale** acc(p.x, 1/2) ∗ p.x > 0

exhale ∃s. list(l, s) ∗ |s| > 0 — is written as → **exhale** list(l) ∗ len(l) > 0

Input parameters

exhale ∃p. p.x ↦ _ ∗ p.y ↦ _

exhale ∃l. list(l, s) ∗ l ≠ null

Forbidden by Viper's syntax

Viper uses
implicit dynamic frames

≈ SL with **heap-dependent** expressions and functions

Allows writing code and specifications in the same language

Implicit existential quantification

inhale A ∗ B
exhale A ∗ B

Challenge 1: Existentials

- ❑ Avoid existentials in the SMT encoding
- ❑ Avoid backtracking
- ❑ Predictable automation

Output parameters

exhale ∃v. (p.x $\overset{1/2}{\mapsto}$ v ∗ v > 0) — is written as → **exhale** acc(p.x, 1/2) ∗ p.x > 0

exhale ∃s. list(l, s) ∗ |s| > 0 — is written as → **exhale** list(l) ∗ len(l) > 0

Input parameters

exhale ∃p. p.x ↦ _ ∗ p.y ↦ _

exhale ∃l. list(l, s) ∗ l ≠ null

Forbidden by Viper's syntax

Viper uses
implicit dynamic frames

≈ SL with **heap-dependent** expressions and functions

Allows writing code and specifications in the same language

Implicit existential quantification

inhale A ∗ B — operationally equivalent to → **inhale** A; **inhale** B

exhale A ∗ B

Challenge 1: Existentials

- ❑ Avoid existentials in the SMT encoding
- ❑ Avoid backtracking
- ❑ Predictable automation

Output parameters

exhale $\exists v.\ (p.x \overset{1/2}{\mapsto} v * v > 0)$ — is written as → **exhale** acc(p.x, 1/2) $*$ p.x > 0

exhale $\exists s.\ \text{list}(l, s) * |s| > 0$ — is written as → **exhale** list(l) $*$ len(l) > 0

Input parameters

exhale $\exists p.\ p.x \mapsto \_ * p.y \mapsto \_$

exhale $\exists l.\ \text{list}(l, s) * l \neq \text{null}$

Forbidden by Viper's syntax

Viper uses
implicit dynamic frames

≈ SL with **heap-dependent** expressions and functions

Allows writing code and specifications in the same language

Implicit existential quantification

Analogous for **exhale**

inhale A $*$ B — operationally equivalent to → **inhale** A; **inhale** B

exhale A $*$ B

Viper’s Expression and Assertion Language

Viper's Expression and Assertion Language

e ::= e.x | $f(e_1, \ldots, e_n)$ | **old**[l](e) | $e_1 + e_2$ | e_1 / e_2 | n | v | b ? $e_1 : e_2$ | …

Viper's Expression and Assertion Language

Field access

$e ::= \;\; e.x \mid f(e_1, ..., e_n) \mid \textbf{old}[l](e) \mid e_1 + e_2 \mid e_1 / e_2 \mid n \mid v \mid b \mathbin{?} e_1 : e_2 \mid ...$

Viper's Expression and Assertion Language

Field access

Heap-dependent functions

$e ::= \; e.x \mid f(e_1, \ldots, e_n) \mid \textbf{old}[l](e) \mid e_1 + e_2 \mid e_1 / e_2 \mid n \mid v \mid b \mathbin{?} e_1 : e_2 \mid \ldots$

Viper's Expression and Assertion Language

Field access

Heap-dependent functions

$e ::= \; e.x \mid f(e_1, \ldots, e_n) \mid \textbf{old}[l](e) \mid e_1 + e_2 \mid e_1 / e_2 \mid n \mid v \mid b \; ? \; e_1 : e_2 \mid \ldots$

$b ::= \; e_1 = e_2 \mid \neg b \mid b_1 \Rightarrow b_2 \mid b_1 \vee b_2 \mid b_1 \wedge b_2 \mid \forall v.\, b \mid \ldots$

Viper's Expression and Assertion Language

Field access

Heap-dependent functions

$e ::= \; e.x \mid f(e_1, \ldots, e_n) \mid \textbf{old}[l](e) \mid e_1 + e_2 \mid e_1 / e_2 \mid n \mid v \mid b \; ? \; e_1 : e_2 \mid \ldots$

Pure expressions

$b ::= \; e_1 = e_2 \mid \neg b \mid b_1 \Rightarrow b_2 \mid b_1 \vee b_2 \mid b_1 \wedge b_2 \mid \forall v.\, b \mid \ldots$

Viper’s Expression and Assertion Language

Field access

Heap-dependent functions

$$e ::= \; e.x \mid f(e_1, \ldots, e_n) \mid \textbf{old}[l](e) \mid e_1 + e_2 \mid e_1 / e_2 \mid n \mid v \mid b \; ? \; e_1 : e_2 \mid \ldots$$

Pure expressions

$$b ::= \; e_1 = e_2 \mid \neg b \mid b_1 \Rightarrow b_2 \mid b_1 \vee b_2 \mid b_1 \wedge b_2 \mid \forall v.\, b \mid \ldots$$

$$A ::= \; b \mid \textbf{acc}(e_1.x, e_2) \mid \textbf{acc}(P(e_1, \ldots, e_n), e) \mid A_1 \ast A_2 \mid A_1 - \!\ast\, A_2 \mid \circledast v.\, A \mid b \Rightarrow A \mid \ldots$$

Viper's Expression and Assertion Language

Field access

Heap-dependent functions

$e ::= \; e.x \mid f(e_1, \ldots, e_n) \mid \textbf{old}[l](e) \mid e_1 + e_2 \mid e_1 / e_2 \mid n \mid v \mid b \; ? \; e_1 : e_2 \mid \ldots$

Pure expressions

$b ::= \; e_1 = e_2 \mid \neg b \mid b_1 \Rightarrow b_2 \mid b_1 \vee b_2 \mid b_1 \wedge b_2 \mid \forall v.\, b \mid \ldots$

$A ::= \; b \mid \textbf{acc}(e_1.x, e_2) \mid \textbf{acc}(P(e_1, \ldots, e_n), e) \mid A_1 \ast A_2 \mid A_1 \mathbin{-\!\ast} A_2 \mid \circledast v.\, A \mid b \Rightarrow A \mid \ldots$

Fractional permissions for heap locations

Viper's Expression and Assertion Language

Field access

Heap-dependent functions

$$e ::= \; e.x \mid f(e_1, \ldots, e_n) \mid \textbf{old}[l](e) \mid e_1 + e_2 \mid e_1 / e_2 \mid n \mid v \mid b \; ? \; e_1 : e_2 \mid \ldots$$

Pure expressions

$$b ::= \; e_1 = e_2 \mid \neg b \mid b_1 \Rightarrow b_2 \mid b_1 \lor b_2 \mid b_1 \land b_2 \mid \forall v.\, b \mid \ldots$$

$$A ::= \; b \mid \textbf{acc}(e_1.x, e_2) \mid \textbf{acc}(P(e_1, \ldots, e_n), e) \mid A_1 \ast A_2 \mid A_1 -\!\!\ast A_2 \mid \circledast v.\, A \mid b \Rightarrow A \mid \ldots$$

Fractional permissions for heap locations

Inductive predicates with fractional permissions

Viper's Expression and Assertion Language

Field access

Heap-dependent functions

$$e ::= \; e.x \mid f(e_1, \ldots, e_n) \mid \mathbf{old}[l](e) \mid e_1 + e_2 \mid e_1 / e_2 \mid n \mid v \mid b \,?\, e_1 : e_2 \mid \ldots$$

Pure expressions

$$b ::= \; e_1 = e_2 \mid \neg b \mid b_1 \Rightarrow b_2 \mid b_1 \vee b_2 \mid b_1 \wedge b_2 \mid \forall v.\, b \mid \ldots$$

$$A ::= \; b \mid \mathbf{acc}(e_1.x, e_2) \mid \mathbf{acc}(P(e_1, \ldots, e_n), e) \mid A_1 \ast A_2 \mid A_1 \mathrel{-\!\!\ast} A_2 \mid \circledast v.\, A \mid b \Rightarrow A \mid \ldots$$

Fractional permissions for heap locations

Inductive predicates with fractional permissions

Iterated separating conjunction

Viper's Expression and Assertion Language

Design choice

- No impure existential
- No impure disjunction
- No impure implication
- No impure negation
- No impure logical conjunction

Field access

Heap-dependent functions

$e ::= \; e.x \mid f(e_1, \ldots, e_n) \mid \textbf{old}[l](e) \mid e_1 + e_2 \mid e_1 / e_2 \mid n \mid v \mid b \; ? \; e_1 : e_2 \mid \ldots$

Pure expressions

$b ::= \; e_1 = e_2 \mid \neg b \mid b_1 \Rightarrow b_2 \mid b_1 \lor b_2 \mid b_1 \land b_2 \mid \forall v.\, b \mid \ldots$

$A ::= \; b \mid \textbf{acc}(e_1.x, e_2) \mid \textbf{acc}(P(e_1, \ldots, e_n), e) \mid A_1 \ast A_2 \mid A_1 -\!\!\ast A_2 \mid \circledast v.\, A \mid b \Rightarrow A \mid \ldots$

Fractional permissions for heap locations

Inductive predicates with fractional permissions

Iterated separating conjunction

Viper’s Expression and Assertion Language

Design choice

- No impure existential
- No impure disjunction
- No impure implication
- No impure negation
- No impure logical conjunction

Field access

Heap-dependent functions

$e ::= \; e.x \mid f(e_1, \ldots, e_n) \mid \textbf{old}[l](e) \mid e_1 + e_2 \mid e_1 / e_2 \mid n \mid v \mid b \; ? \; e_1 : e_2 \mid \ldots$

Pure expressions

$b ::= \; e_1 = e_2 \mid \neg b \mid b_1 \Rightarrow b_2 \mid b_1 \lor b_2 \mid b_1 \land b_2 \mid \forall v.\, b \mid \ldots$

$A ::= \; b \mid \textbf{acc}(e_1.x, e_2) \mid \textbf{acc}(P(e_1, \ldots, e_n), e) \mid A_1 * A_2 \mid A_1 -\!\!* A_2 \mid \circledast v.\, A \mid b \Rightarrow A \mid \ldots$

Fractional permissions for heap locations

Inductive predicates with fractional permissions

Iterated separating conjunction

Challenge 2: Inductive Predicates

Challenge 2: Inductive Predicates

$$\text{list}(l) \triangleq \big(l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) * \textbf{acc}(l.\text{next}) * \text{list}(l.\text{next}))\big)$$

Challenge 2: Inductive Predicates

Input parameters only

$$\text{list}(l) \triangleq \big(l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) * \textbf{acc}(l.\text{next}) * \text{list}(l.\text{next}))\big)$$

Challenge 2: Inductive Predicates

Input parameters only

$$\text{list}(l) \triangleq \big(l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) \ast \textbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next}))\big)$$

1. Existence of a (least) fixed-point?

Challenge 2: Inductive Predicates

Input parameters only

$$\text{list}(l) \triangleq \left(l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) \ast \textbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next}))\right)$$

1. Existence of a (least) fixed-point?
2. How to automate **inhale** list(l) and **exhale** list(l)?

Challenge 2: Inductive Predicates

Input parameters only

$$\text{list}(l) \triangleq \big(l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) \ast \textbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next}))\big)$$

1. Existence of a (least) fixed-point?
2. How to automate **inhale** list(l) and **exhale** list(l)?
3. How to reason about output parameters?

Challenge 2: Inductive Predicates

Input parameters only

$$\text{list}(l) \triangleq \big(l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) \ast \textbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next}))\big)$$

1. Existence of a (least) fixed-point?
2. How to automate **inhale** list(l) and **exhale** list(l)?
3. How to reason about output parameters?
4. How to know when to **unfold** the definition?

Challenge 2: Inductive Predicates

Input parameters only

unfold

$$\mathrm{list}(l) \triangleq \big(l \neq \mathrm{null} \Rightarrow (\mathbf{acc}(l.\mathrm{value}) \ast \mathbf{acc}(l.\mathrm{next}) \ast \mathrm{list}(l.\mathrm{next}))\big)$$

1. Existence of a (least) fixed-point?
2. How to automate **inhale** list(l) and **exhale** list(l)?
3. How to reason about output parameters?
4. How to know when to **unfold** the definition?

Challenge 2: Inductive Predicates

Input parameters only

unfold

$$\text{list}(l) \triangleq (l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) * \textbf{acc}(l.\text{next}) * \text{list}(l.\text{next})))$$

fold

1. Existence of a (least) fixed-point?
2. How to automate **inhale** list(l) and **exhale** list(l)?
3. How to reason about output parameters?
4. How to know when to **unfold** the definition?

Challenge 2: Inductive Predicates

Input parameters only

unfold

$$\text{list}(l) \triangleq (l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) * \textbf{acc}(l.\text{next}) * \text{list}(l.\text{next})))$$

fold

1. Existence of a (least) fixed-point?
2. How to automate **inhale** list(l) and **exhale** list(l)?
3. How to reason about output parameters?
4. How to know when to **unfold** the definition?

Viper's approach: Treat predicates **isorecursively**

Isorecursive Predicates

unfold

$$\text{list}(l) \triangleq \big(l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) \ast \textbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next}))\big)$$

fold

Isorecursive Predicates

Not an equality!

unfold

$$\text{list}(l) \triangleq \big(l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) \ast \textbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next}))\big)$$

fold

Isorecursive Predicates

Not an equality!

unfold

$$\text{list}(l) \triangleq \big(l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) \ast \textbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next}))\big)$$

fold

(simplified) Viper's state model: $(\text{Loc} \rightharpoonup (0, 1] \times \text{Val})$

Isorecursive Predicates

Not an equality!

unfold

$$\text{list}(l) \triangleq \big(l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) \ast \textbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next}))\big)$$

fold

(simplified) Viper's state model: $(\text{Loc} \rightharpoonup (0, 1] \times \text{Val}) \times (\text{PredLoc} \rightarrow [0, +\infty))$

Isorecursive Predicates

Not an equality!

unfold

$$\mathrm{list}(l) \triangleq \big(l \neq \mathrm{null} \Rightarrow (\mathbf{acc}(l.\mathrm{value}) * \mathbf{acc}(l.\mathrm{next}) * \mathrm{list}(l.\mathrm{next}))\big)$$

fold

(simplified) Viper's state model: $(\mathrm{Loc} \rightharpoonup (0, 1] \times \mathrm{Val}) \times (\mathrm{PredLoc} \rightarrow [0, +\infty))$

Predicate mask

Isorecursive Predicates

Not an equality!

unfold

$$\text{list}(l) \triangleq \big(l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) \ast \textbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next}))\big)$$

fold

VIPER

inhale list(l)

exhale list(l)

(simplified) Viper's state model: $(\text{Loc} \rightharpoonup (0, 1] \times \text{Val}) \times (\text{PredLoc} \rightarrow [0, +\infty))$

Predicate mask

Isorecursive Predicates

Acts on the predicate mask

Not an equality!

unfold

$\text{list}(l) \triangleq (l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) \ast \textbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next})))$

fold

inhale list(l)

exhale list(l)

(simplified) Viper's state model: $(\text{Loc} \rightharpoonup (0, 1] \times \text{Val}) \times (\text{PredLoc} \rightarrow [0, +\infty))$

Predicate mask

Isorecursive Predicates

Acts on the predicate mask

```
inhale list(l)
if (l != null) {

    l.value := l.value + 1


}
exhale list(l)
```

Not an equality!

unfold

$\text{list}(l) \triangleq (l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) \ast \textbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next})))$

fold

(simplified) Viper's state model: $(\text{Loc} \rightharpoonup (0, 1] \times \text{Val}) \times (\text{PredLoc} \to [0, +\infty))$

Predicate mask

Isorecursive Predicates

Not an equality!

unfold

$$\text{list}(l) \triangleq (l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) \ast \textbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next})))$$

fold

Acts on the predicate mask

```
inhale list(l)
if (l != null) {
    unfold list(l)
    l.value := l.value + 1
    fold list(l)

}
exhale list(l)
```

(simplified) Viper's state model: $(\text{Loc} \rightharpoonup (0, 1] \times \text{Val}) \times (\text{PredLoc} \to [0, +\infty))$

Predicate mask

Isorecursive Predicates

Acts on the predicate mask

```
inhale list(l)
if (l != null) {
    unfold list(l)
    l.value := l.value + 1
    fold list(l)

}
exhale list(l)
```

Front-end has to emit fold/unfold statements

Not an equality!

unfold

$$\mathrm{list}(l) \triangleq \big(l \neq \mathrm{null} \Rightarrow (\mathbf{acc}(l.\mathrm{value}) \ast \mathbf{acc}(l.\mathrm{next}) \ast \mathrm{list}(l.\mathrm{next}))\big)$$

fold

(simplified) Viper's state model: $(\mathrm{Loc} \rightharpoonup (0, 1] \times \mathrm{Val}) \times (\mathrm{PredLoc} \rightarrow [0, +\infty))$

Predicate mask

Isorecursive Predicates

Acts on the predicate mask

```
inhale list(l)
if (l != null) {
    unfold list(l)
    l.value := l.value + 1
    fold list(l)

}
exhale list(l)
```

Front-end has to emit fold/unfold statements

Not an equality!

unfold ≈ **exhale** list(l); **inhale** body[list(l)]

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

fold

(simplified) Viper's state model: $(\text{Loc} \rightharpoonup (0, 1] \times \text{Val}) \times (\text{PredLoc} \to [0, +\infty))$

Predicate mask

Isorecursive Predicates

Acts on the predicate mask

```
inhale list(l)
if (l != null) {
  unfold list(l)
  l.value := l.value + 1
  fold list(l)

}
exhale list(l)
```

Front-end has to emit fold/unfold statements

Not an equality!

unfold ≈ **exhale** list(l); **inhale** body[list(l)]

$\text{list(l)} \triangleq (\text{l} \neq \text{null} \Rightarrow (\textbf{acc}(\text{l.value}) \ast \textbf{acc}(\text{l.next}) \ast \text{list(l.next)}))$

fold ≈ **exhale** body[list(l)]; **inhale** list(l)

(simplified) Viper's state model: $(\text{Loc} \rightharpoonup (0, 1] \times \text{Val}) \times (\text{PredLoc} \to [0, +\infty))$

Predicate mask

Isorecursive Predicates

Not an equality!

unfold ≈ **exhale** list(l); **inhale** body[list(l)]

$$\text{list}(l) \triangleq (l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) \ast \textbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next})))$$

fold ≈ **exhale** body[list(l)]; **inhale** list(l)

Acts on the predicate mask

```
inhale list(l)
if (l != null) {
    unfold list(l)
    l.value := l.value + 1
    fold list(l)
    v := unfolding list(l) in l.value
}
exhale list(l)
```

Front-end has to emit fold/unfold statements

Temporary unfold

(simplified) Viper's state model: (Loc ⇀ (0, 1] × Val) × (PredLoc → [0, +∞))

Predicate mask

Isorecursive Predicates

Acts on the predicate mask

```
inhale list(l)
if (l != null) {
    unfold list(l)
    l.value := l.value + 1
    fold list(l)
    v := unfolding list(l) in l.value
}
exhale list(l)
```

Front-end has to emit fold/unfold statements

Temporary unfold

Not an equality!

unfold ≈ **exhale** list(l); **inhale** body[list(l)]

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

fold ≈ **exhale** body[list(l)]; **inhale** list(l)

(simplified) Viper's state model: (Loc ⇀ (0, 1] × Val) × (PredLoc → [0, +∞)) × … × … × …

Predicate mask

Isorecursive Predicates

Not an equality!

unfold ≈ **exhale** list(l); **inhale** body[list(l)]

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

fold ≈ **exhale** body[list(l)]; **inhale** list(l)

Acts on the predicate mask

```
inhale list(l)
if (l != null) {
    unfold list(l)
    l.value := l.value + 1
    fold list(l)
    v := unfolding list(l) in l.value
}
exhale list(l)
```

Front-end has to emit fold/unfold statements

Temporary unfold

(simplified) Viper's state model: (Loc ⇀ (0, 1] × Val) × (PredLoc → [0, +∞)) × ... × ... × ...

Information about predicates' contents

Predicate mask

Isorecursive Predicates

Not an equality!

unfold ≈ **exhale** list(l); **inhale** body[list(l)]

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

fold ≈ **exhale** body[list(l)]; **inhale** list(l)

Acts on the predicate mask

Front-end has to emit fold/unfold statements

```
inhale list(l)
if (l != null) {
    unfold list(l)
    l.value := l.value + 1
    fold list(l)
    v := unfolding list(l) in l.value
}
exhale list(l)
```

Temporary unfold

(simplified) Viper's state model: (Loc ⇀ (0, 1] × Val) × (PredLoc → [0, +∞)) × … × … × …

Information about predicates' contents

Predicate mask

Magic wands

Isorecursive Predicates

Not an equality!

unfold ≈ **exhale** list(l); **inhale** body[list(l)]

$\text{list}(l) \triangleq (l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) * \textbf{acc}(l.\text{next}) * \text{list}(l.\text{next})))$

fold ≈ **exhale** body[list(l)]; **inhale** list(l)

Acts on the predicate mask

```
inhale list(l)
if (l != null) {
    unfold list(l)
    l.value := l.value + 1
    fold list(l)
    v := unfolding list(l) in l.value
}
exhale list(l)
```

Front-end has to emit fold/unfold statements

Temporary unfold

(simplified) Viper's state model: $(\text{Loc} \rightharpoonup (0, 1] \times \text{Val}) \times (\text{PredLoc} \rightarrow [0, +\infty)) \times \ldots \times \ldots \times \ldots$

Information about predicates' contents

Information about old states

Predicate mask

Magic wands

Existence of a Fixed-Point

unfold

$$\mathrm{list}(l) \triangleq \big(l \neq \mathrm{null} \Rightarrow (\mathbf{acc}(l.\mathrm{value}) \ast \mathbf{acc}(l.\mathrm{next}) \ast \mathrm{list}(l.\mathrm{next}))\big)$$

fold

Existence of a Fixed-Point

unfold

Fixed-point?

$$\mathsf{list}(l) \triangleq \big(l \neq \mathsf{null} \Rightarrow (\mathbf{acc}(l.\mathsf{value}) \ast \mathbf{acc}(l.\mathsf{next}) \ast \mathsf{list}(l.\mathsf{next}))\big)$$

fold

Existence of a Fixed-Point

unfold

Fixed-point?

$$\mathrm{list}(l) \triangleq \big(l \neq \mathrm{null} \Rightarrow (\mathbf{acc}(l.\mathrm{value}) \ast \mathbf{acc}(l.\mathrm{next}) \ast \mathrm{list}(l.\mathrm{next}))\big)$$

fold

$A ::= \; b \mid \mathbf{acc}(e_1.x, e_2) \mid \mathbf{acc}(P(e_1, \ldots, e_n), e) \mid A_1 \ast A_2 \mid A_1 -\!\!\ast\, A_2 \mid \circledast v.\, A \mid b \Rightarrow A \mid \ldots$

Existence of a Fixed-Point

unfold

$$\text{list}(l) \triangleq \big(l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) \ast \textbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next}))\big)$$

fold

Fixed-point?

Magic wands are not allowed in recursive definitions

$$A ::= \; b \mid \textbf{acc}(e_1.x, e_2) \mid \textbf{acc}(P(e_1, \ldots, e_n), e) \mid A_1 \ast A_2 \mid A_1 \mathrel{-\!\!\ast} A_2 \mid \circledast v.\, A \mid b \Rightarrow A \mid \ldots$$

Existence of a Fixed-Point

unfold

Fixed-point?

$$\mathrm{list}(l) \triangleq \big(l \neq \mathrm{null} \Rightarrow (\mathbf{acc}(l.\mathrm{value}) \ast \mathbf{acc}(l.\mathrm{next}) \ast \mathrm{list}(l.\mathrm{next}))\big)$$

fold

Magic wands are not allowed in recursive definitions

$$A ::= b \mid \mathbf{acc}(e_1.x, e_2) \mid \mathbf{acc}(P(e_1, \dots, e_n), e) \mid A_1 \ast A_2 \mid A_1 \mathrel{-\!\!\ast} A_2 \mid \circledast v.\ A \mid b \Rightarrow A \mid \dots$$

Scott-continuity

Existence of a Fixed-Point

unfold

Fixed-point?

$$\mathrm{list}(l) \triangleq \big(l \neq \mathrm{null} \Rightarrow (\mathbf{acc}(l.\mathrm{value}) \ast \mathbf{acc}(l.\mathrm{next}) \ast \mathrm{list}(l.\mathrm{next}))\big)$$

fold

Magic wands are not allowed in recursive definitions

$$A ::= \; b \mid \mathbf{acc}(e_1.x, e_2) \mid \mathbf{acc}(P(e_1, \ldots, e_n), e) \mid A_1 \ast A_2 \mid A_1 -\!\!\ast A_2 \mid \circledast v.\, A \mid b \Rightarrow A \mid \ldots$$

Scott-continuity

Existence of a least fixed point

Existence of a Fixed-Point

unfold

Fixed-point?

$$\text{list}(l) \triangleq (l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) \ast \textbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next})))$$

fold

Magic wands are not allowed in recursive definitions

$$A ::= b \mid \textbf{acc}(e_1.x, e_2) \mid \textbf{acc}(P(e_1, \ldots, e_n), e) \mid A_1 \ast A_2 \mid A_1 -\!\!\ast A_2 \mid \circledast v.\ A \mid b \Rightarrow A \mid \ldots$$

Scott-continuity

Existence of a least fixed point

Kleene's theorem

Any given predicate instance has a **finite** number of predicate instances folded within it

Existence of a Fixed-Point

unfold

Fixed-point?

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

fold

Magic wands are not allowed in recursive definitions

A ::= b | **acc**(e_1.x, e_2) | **acc**(P(e_1, ..., e_n), e) | A_1 ∗ A_2 | A_1 −∗ A_2 | ⊛v. A | b ⇒ A | ...

Scott-continuity

Existence of a least fixed point

Kleene's theorem

Can be used as a termination measure (e.g., for heap-dependent functions)

Any given predicate instance has a **finite** number of predicate instances folded within it

Heap-Dependent Functions

$\text{list}(l) \triangleq \big(l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) \ast \textbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next}))\big)$

Heap-Dependent Functions

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

```
function len(l): Int
   requires list(l)

{ l == null ? 0 : unfolding list(l) in 1 + len(l.next) }
```

Heap-Dependent Functions

$$list(l) \triangleq (l \neq null \Rightarrow (\textbf{acc}(l.value) \ast \textbf{acc}(l.next) \ast list(l.next)))$$

```
function len(l): Int
  requires list(l)

{ l == null ? 0 : unfolding list(l) in 1 + len(l.next) }
```

Heap-Dependent Functions

$\text{list}(l) \triangleq \big(l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) * \textbf{acc}(l.\text{next}) * \text{list}(l.\text{next}))\big)$

VIPER

```
function len(l): Int
  requires list(l)
  ensures result >= 0
{ l == null ? 0 : unfolding list(l) in 1 + len(l.next) }
```

Heap-Dependent Functions

$\text{list}(l) \triangleq (l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) \ast \textbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next})))$

```
function len(l): Int
  requires list(l)
  ensures result >= 0
{ l == null ? 0 : unfolding list(l) in 1 + len(l.next) }
```

Automatically proven by induction

VIPER

Heap-Dependent Functions

$$\text{list}(l) \triangleq \big(l \neq \text{null} \Rightarrow (\mathbf{acc}(l.\text{value}) \ast \mathbf{acc}(l.\text{next}) \ast \text{list}(l.\text{next}))\big)$$

Automatically proven by induction

```
function len(l): Int
  requires list(l)
  ensures result >= 0
{ l == null ? 0 : unfolding list(l) in 1 + len(l.next) }
```

Heap-Dependent Functions

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

```
function len(l): Int
  requires list(l)
  ensures result >= 0
{ l == null ? 0 : unfolding list(l) in 1 + len(l.next) }
```

Automatically proven by induction

Well-founded order

VIPER

Heap-Dependent Functions

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

```
function len(l): Int
   requires list(l)
   ensures result >= 0
{ l == null ? 0 : unfolding list(l) in 1 + len(l.next) }
```

Automatically proven by induction

Well-founded order

Heap-Dependent Functions

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

Automatically proven by induction

VIPER

```
function len(l): Int
  requires list(l)
  ensures result >= 0
{ l == null ? 0 : unfolding list(l) in 1 + len(l.next) }
```

Well-founded order

VIPER

```
function sum(l): Int
  requires list(l)
{ l == null ? 0 : unfolding list(l) in l.value + sum(l.next) }
```

Heap-Dependent Functions

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

Automatically proven by induction

```
function len(l): Int
  requires list(l)
  ensures result >= 0
{ l == null ? 0 : unfolding list(l) in 1 + len(l.next) }
```

Well-founded order

How deep should the definition be unfolded?

```
function sum(l): Int
  requires list(l)
{ l == null ? 0 : unfolding list(l) in l.value + sum(l.next) }
```

Heap-Dependent Functions

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

Automatically proven by induction

```
function len(l): Int
  requires list(l)
  ensures result >= 0
{ l == null ? 0 : unfolding list(l) in 1 + len(l.next) }
```

Well-founded order

How deep should the definition be unfolded?

```
function sum(l): Int
  requires list(l)
{ l == null ? 0 : unfolding list(l) in l.value + sum(l.next) }
```

```
method main(l: Ref)
    requires list(l) ∗ len(l) >= 2
    ensures list(l) ∗ sum(l) == old(sum(l)) + 5
{


    l.next.value := l.next.value + 5


}
```

Heap-Dependent Functions

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

Automatically proven by induction

```
function len(l): Int
   requires list(l)
   ensures result >= 0
{ l == null ? 0 : unfolding list(l) in 1 + len(l.next) }
```

Well-founded order

How deep should the definition be unfolded?

```
function sum(l): Int
   requires list(l)
{ l == null ? 0 : unfolding list(l) in l.value + sum(l.next) }
```

```
method main(l: Ref)
    requires list(l) ∗ len(l) >= 2
    ensures list(l) ∗ sum(l) == old(sum(l)) + 5
{
    unfold list(l)
    unfold list(l.next)
    l.next.value := l.next.value + 5
    fold list(l.next)
    fold list(l)
}
```

Heap-Dependent Functions

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

```
function len(l): Int
  requires list(l)
  ensures result >= 0
{ l == null ? 0 : unfolding list(l) in 1 + len(l.next) }
```

Automatically proven by induction

Well-founded order

How deep should the definition be unfolded?

```
function sum(l): Int
  requires list(l)
{ l == null ? 0 : unfolding list(l) in l.value + sum(l.next) }
```

```
method main(l: Ref)
    requires list(l) ∗ len(l) >= 2
    ensures list(l) ∗ sum(l) == old(sum(l)) + 5
{
    unfold list(l)
    unfold list(l.next)
    l.next.value := l.next.value + 5
    fold list(l.next)
    fold list(l)
}
```

sum(l) = l.value + sum(l.next)

Heap-Dependent Functions

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

Automatically proven by induction

```
function len(l): Int
  requires list(l)
  ensures result >= 0
{ l == null ? 0 : unfolding list(l) in 1 + len(l.next) }
```

Well-founded order

How deep should the definition be unfolded?

```
function sum(l): Int
  requires list(l)
{ l == null ? 0 : unfolding list(l) in l.value + sum(l.next) }
```

```
method main(l: Ref)
  requires list(l) ∗ len(l) >= 2
  ensures list(l) ∗ sum(l) == old(sum(l)) + 5
{
  unfold list(l)
  unfold list(l.next)
  l.next.value := l.next.value + 5
  fold list(l.next)
  fold list(l)
}
```

sum(l) = l.value + sum(l.next)

sum(l) = l.value + l.next.value + sum(l.next.next)

Heap-Dependent Functions

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

Automatically proven by induction

```
function len(l): Int
  requires list(l)
  ensures result >= 0
{ l == null ? 0 : unfolding list(l) in 1 + len(l.next) }
```

Well-founded order

How deep should the definition be unfolded?

```
function sum(l): Int
  requires list(l)
{ l == null ? 0 : unfolding list(l) in l.value + sum(l.next) }
```

```
method main(l: Ref)
    requires list(l) ∗ len(l) >= 2
    ensures list(l) ∗ sum(l) == old(sum(l)) + 5
{
    unfold list(l)
    unfold list(l.next)
    l.next.value := l.next.value + 5
    fold list(l.next)
    fold list(l)
}
```

sum(l) = l.value + sum(l.next)

sum(l) = l.value + l.next.value + sum(l.next.next)

Output parameter of list(l.next.next)

Heap-Dependent Functions

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

Automatically proven by induction

```
function len(l): Int
  requires list(l)
  ensures result >= 0
{ l == null ? 0 : unfolding list(l) in 1 + len(l.next) }
```

Well-founded order

How deep should the definition be unfolded?

```
function sum(l): Int
  requires list(l)
{ l == null ? 0 : unfolding list(l) in l.value + sum(l.next) }
```

```
method main(l: Ref)
  requires list(l) ∗ len(l) >= 2
  ensures list(l) ∗ sum(l) == old(sum(l)) + 5
{
  unfold list(l)
  unfold list(l.next)
  l.next.value := l.next.value + 5
  fold list(l.next)
  fold list(l)
}
```

sum(l) = l.value + sum(l.next)

sum(l) = l.value + l.next.value + sum(l.next.next)

Output parameter of list(l.next.next)

Fractional (Recursive) Predicates

Fractional (Recursive) Predicates

list(l)

Fractional (Recursive) Predicates

split

list(l) **acc**(list(l), 0.5) ∗ **acc**(list(l), 0.5)

Fractional (Recursive) Predicates

Enables concurrent reads

split

list(l) → **acc**(list(l), 0.5) ∗ **acc**(list(l), 0.5)

Fractional (Recursive) Predicates

Enables concurrent reads

split

list(l)   **acc**(list(l), 0.5) ∗ **acc**(list(l), 0.5)

combine

Fractional (Recursive) Predicates

split

list(l)   **acc**(list(l), 0.5) ∗ **acc**(list(l), 0.5)

combine

Enables concurrent reads

inhale list(l)
…
exhale acc(list(l), 0.5)
…
inhale acc(list(l), 0.5)
…
exhale list(l)

Fractional (Recursive) Predicates

split

list(l) ⟷ **acc**(list(l), 0.5) ∗ **acc**(list(l), 0.5)

combine

Enables concurrent reads

Acts on the predicate mask

inhale list(l)
…
exhale acc(list(l), 0.5)
…
inhale acc(list(l), 0.5)
…
exhale list(l)

Fractional (Recursive) Predicates

split

list(l)   **acc**(list(l), 0.5) ∗ **acc**(list(l), 0.5)

combine

Enables concurrent reads

Acts on the predicate mask

inhale list(l)
...
exhale acc(list(l), 0.5)
...
inhale acc(list(l), 0.5)
...
exhale list(l)

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

Fractional (Recursive) Predicates

split

list(l) → **acc**(list(l), 0.5) ∗ **acc**(list(l), 0.5)

combine

Enables concurrent reads

Acts on the predicate mask

inhale list(l)
…
exhale acc(list(l), 0.5)
…
inhale acc(list(l), 0.5)
…
exhale list(l)

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

automatically derived (syntactically)

acc(list(l), 0.5) ≜ (l ≠ null ⇒ (**acc**(l.value, 0.5) ∗ **acc**(l.next, 0.5) ∗ **acc**(list(l.next, 0.5))))

Fractional (Recursive) Predicates

split

list(l)   **acc**(list(l), 0.5) ∗ **acc**(list(l), 0.5)

combine

Enables concurrent reads

inhale list(l)
…
exhale acc(list(l), 0.5)
…
inhale acc(list(l), 0.5)
…
exhale list(l)

Acts on the predicate mask

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

automatically derived (syntactically)

acc(list(l), 0.5) ≜ (l ≠ null ⇒ (**acc**(l.value, 0.5) ∗ **acc**(l.next, 0.5) ∗ **acc**(list(l.next, 0.5))))

Fractional (Recursive) Predicates

split

list(l) **acc**(list(l), 0.5) ∗ **acc**(list(l), 0.5)

combine

Enables concurrent reads

inhale list(l)
…
exhale acc(list(l), 0.5)
…
inhale acc(list(l), 0.5)
…
exhale list(l)

Acts on the predicate mask

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

automatically derived (syntactically)

unfold

acc(list(l), 0.5) ≜ (l ≠ null ⇒ (**acc**(l.value, 0.5) ∗ **acc**(l.next, 0.5) ∗ **acc**(list(l.next, 0.5))))

fold

Fractional (Recursive) Predicates

split

list(l) $\quad$ **acc**(list(l), 0.5) ∗ **acc**(list(l), 0.5)

combine

Enables concurrent reads

inhale list(l)
...
exhale acc(list(l), 0.5)
...
inhale acc(list(l), 0.5)
...
exhale list(l)

Acts on the predicate mask

$$\text{list}(l) \triangleq (l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}) * \textbf{acc}(l.\text{next}) * \text{list}(l.\text{next})))$$

automatically derived (syntactically)

unfold

$$\textbf{acc}(\text{list}(l), 0.5) \triangleq (l \neq \text{null} \Rightarrow (\textbf{acc}(l.\text{value}, 0.5) * \textbf{acc}(l.\text{next}, 0.5) * \textbf{acc}(\text{list}(l.\text{next}, 0.5))))$$

fold

Sound for Viper (recursive) predicate definitions!

Fractional (Recursive) Predicates

split

list(l) → **acc**(list(l), 0.5) ∗ **acc**(list(l), 0.5)

combine

Enables concurrent reads

inhale list(l)
…
exhale acc(list(l), 0.5)
…
inhale acc(list(l), 0.5)
…
exhale list(l)

Acts on the predicate mask

list(l) ≜ (l ≠ null ⇒ (**acc**(l.value) ∗ **acc**(l.next) ∗ list(l.next)))

automatically derived (syntactically)

unfold

acc(list(l), 0.5) ≜ (l ≠ null ⇒ (**acc**(l.value, 0.5) ∗ **acc**(l.next, 0.5) ∗ **acc**(list(l.next, 0.5))))

fold

Sound for Viper (recursive) predicate definitions!

Theoretical foundations in our OOPSLA’22 paper
“Fractional Resources in Unbounded Separation Logic”

Outline of the Talk

1. Overview of Viper

2. Inhale and Exhale: An Operational View of Separation Logic

3. Designed for Automation

4. Toward a Foundational Viper

Toward a Foundational Viper

Toward a Foundational Viper

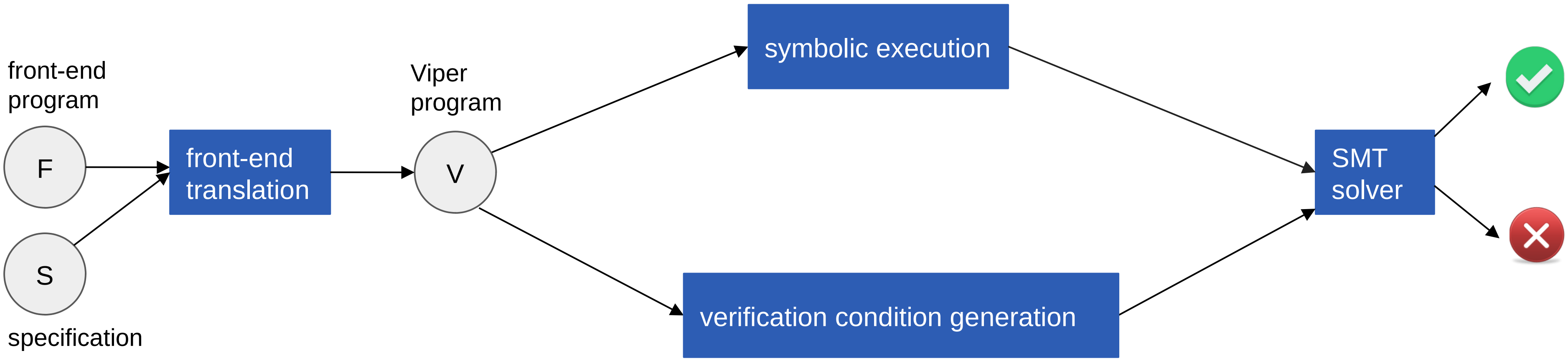
Iris from the ground up: A modular foundation for higher-order concurrent separation logic
Ralf Jung, Robert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, Derek Dreyer

Toward a Foundational Viper

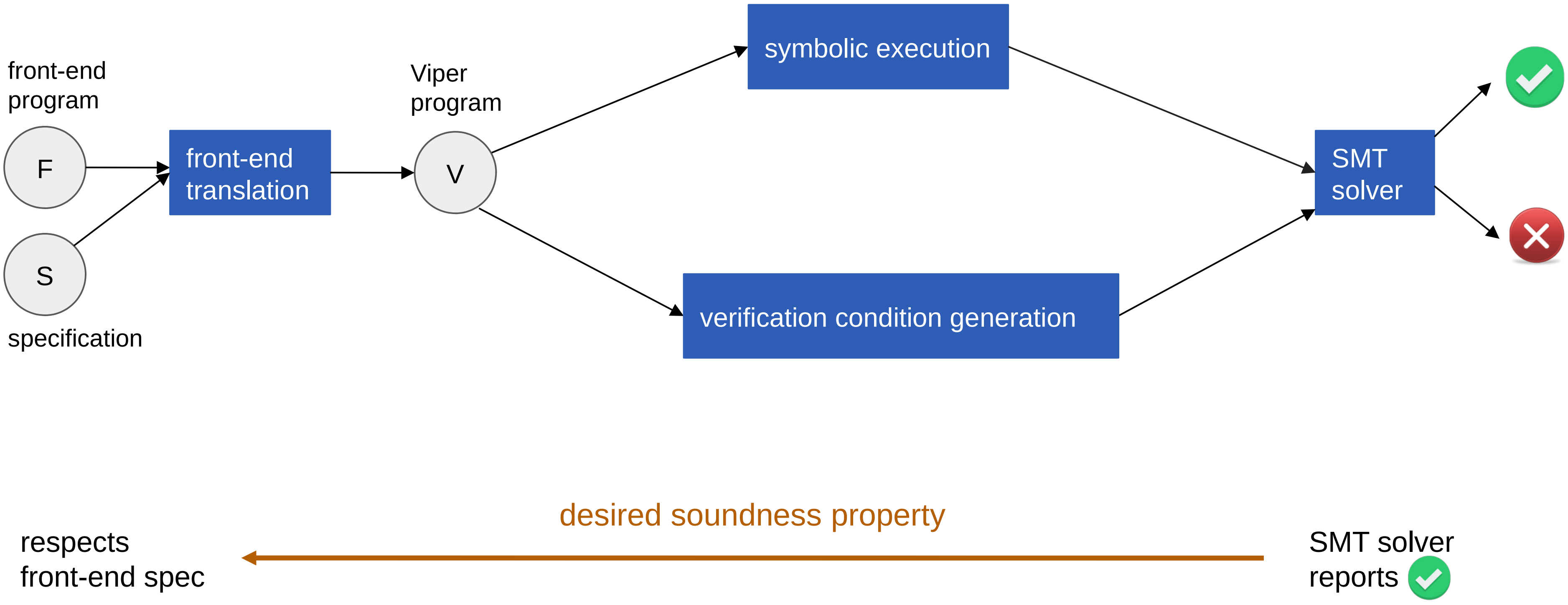
"This [foundational approach] is in contrast to tools like [...] **Viper**, which have much larger trusted computing bases because they assume the soundness of non-trivial extensions of Hoare logic and do not produce independently checkable proof terms."

Iris from the ground up: A modular foundation for higher-order concurrent separation logic
Ralf Jung, Robert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, Derek Dreyer

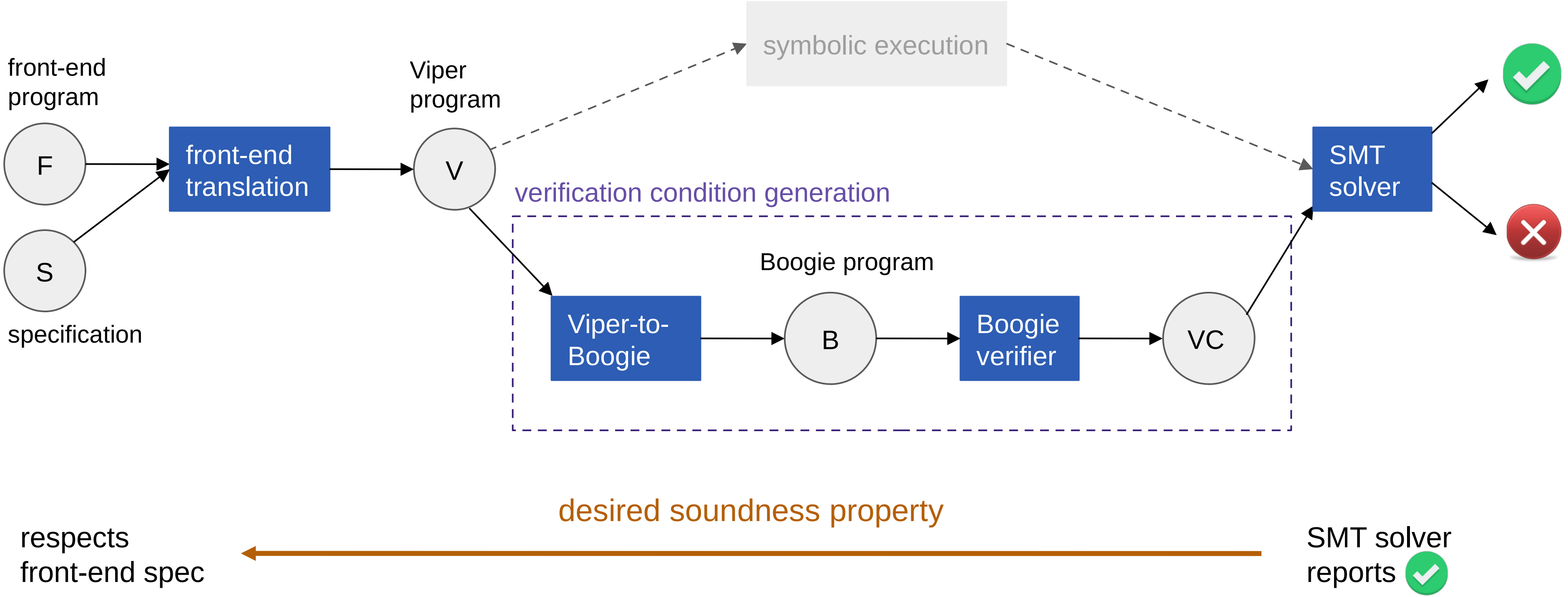
Soundness


front-end
program
F
S
specification
front-end
translation
Viper
program
V
symbolic execution
verification condition generation
SMT
solver

Soundness

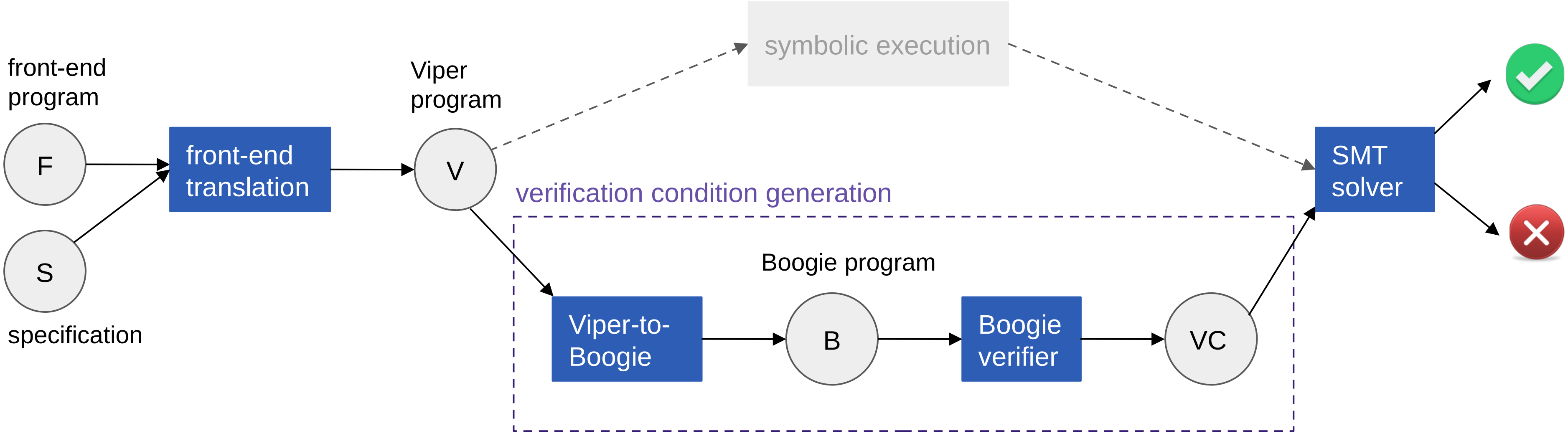

front-end
program
F
S
specification
front-end
translation
Viper
program
V
symbolic execution
verification condition generation
SMT
solver
desired soundness property
respects
front-end spec
SMT solver
reports

Soundness

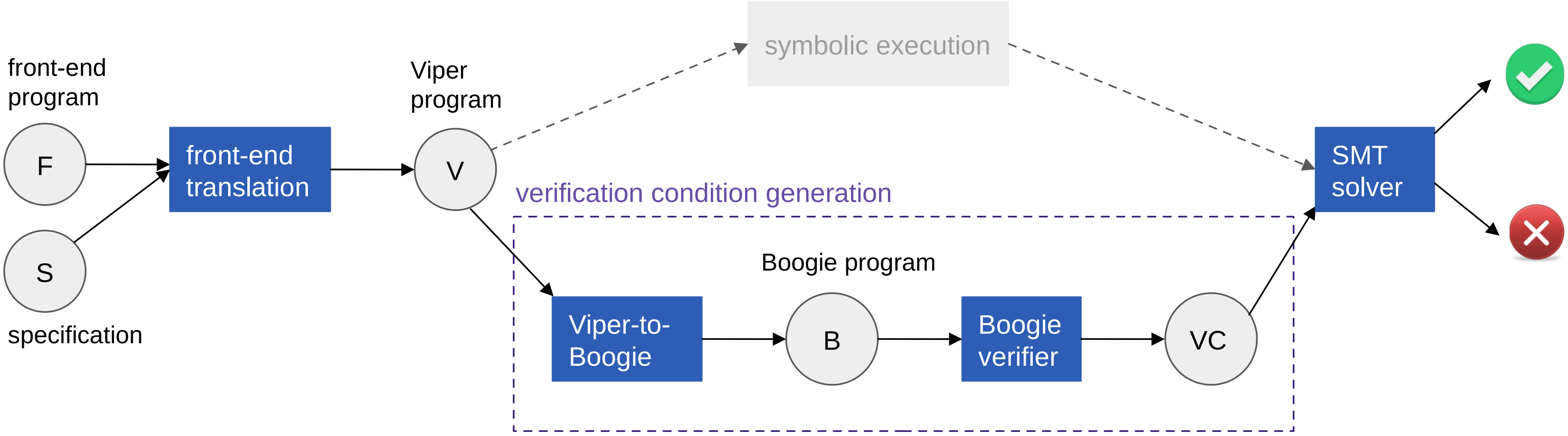


front-end program
F
S
specification
front-end translation
Viper program
V
symbolic execution
verification condition generation
Boogie program
Viper-to-Boogie
B
Boogie verifier
VC
SMT solver
desired soundness property
respects front-end spec
SMT solver reports

Soundness: Proof Strategy

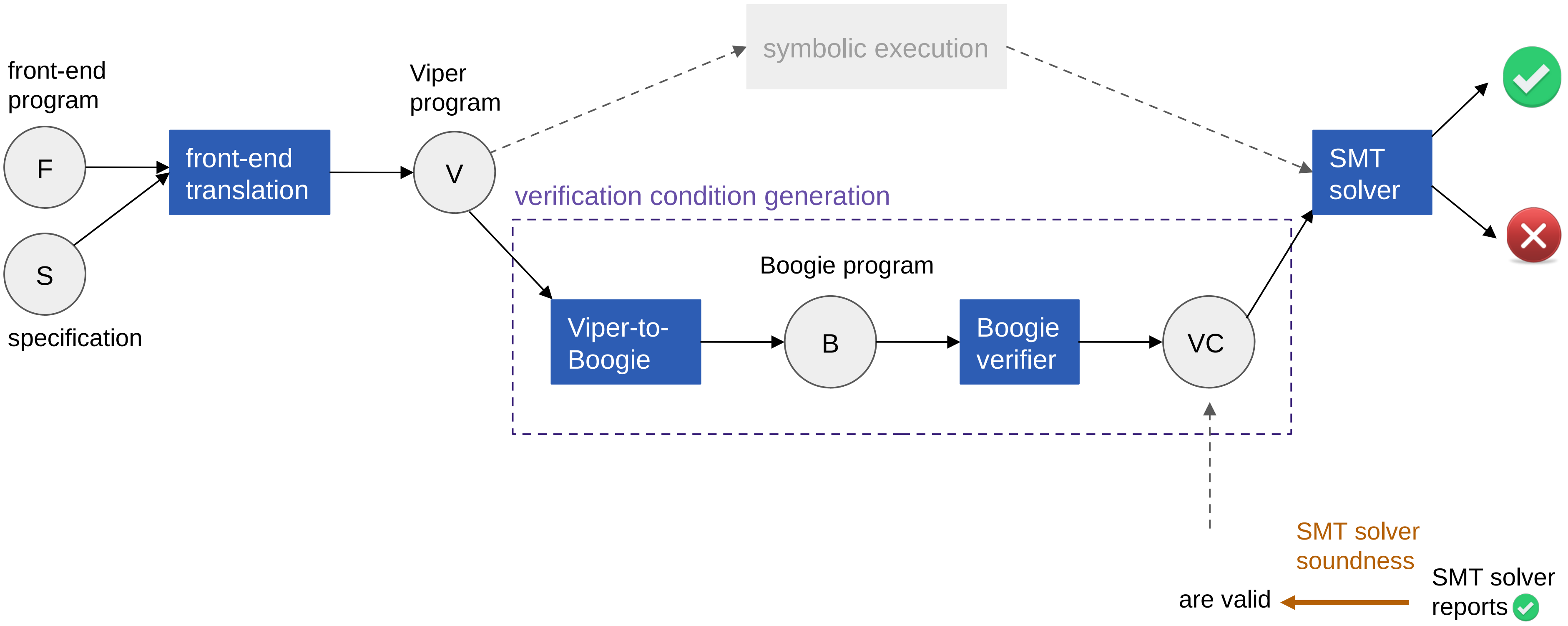


Soundness: Proof Strategy

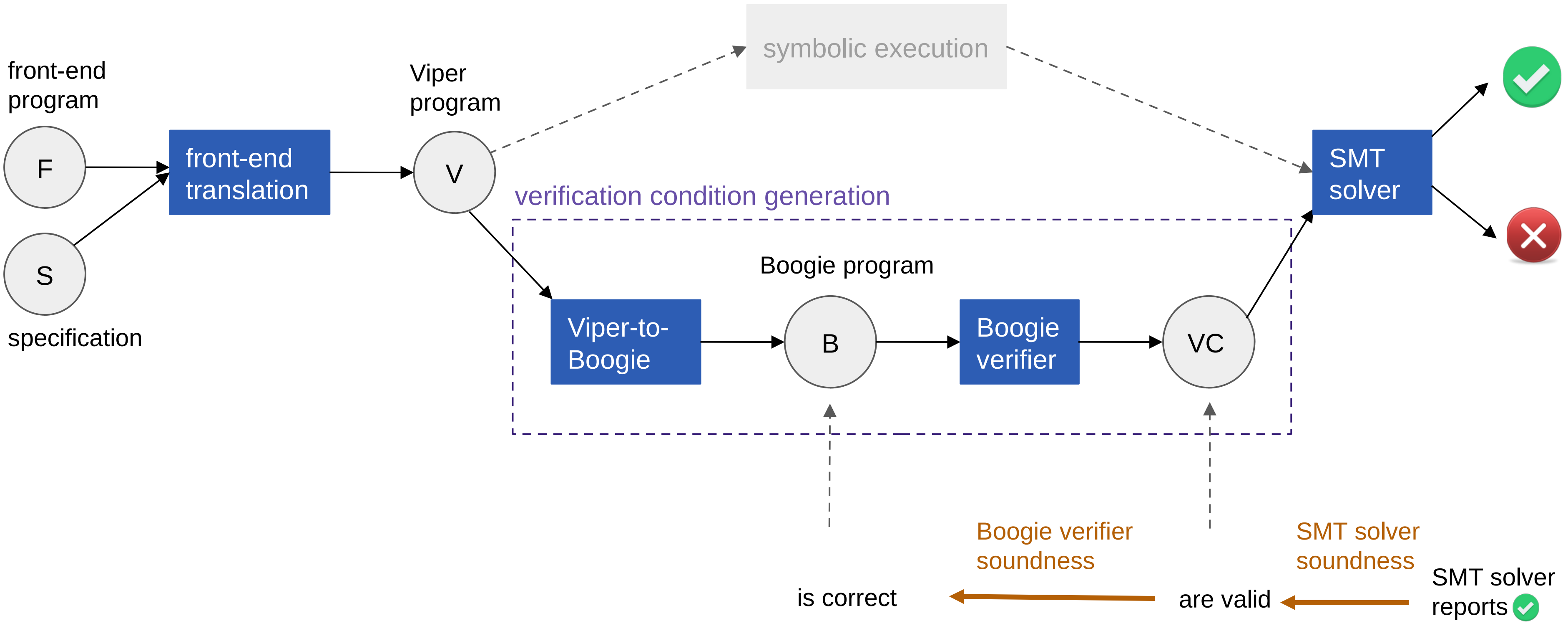


SMT solver reports

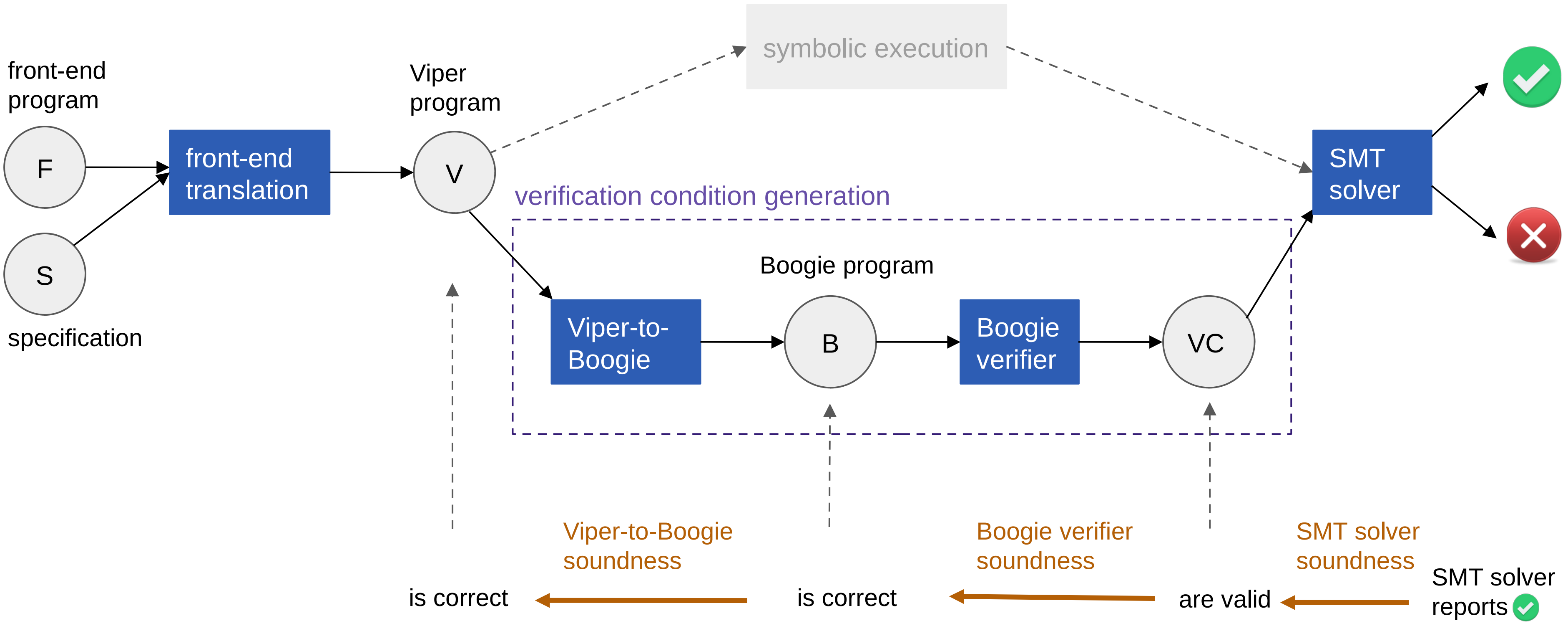
Soundness: Proof Strategy



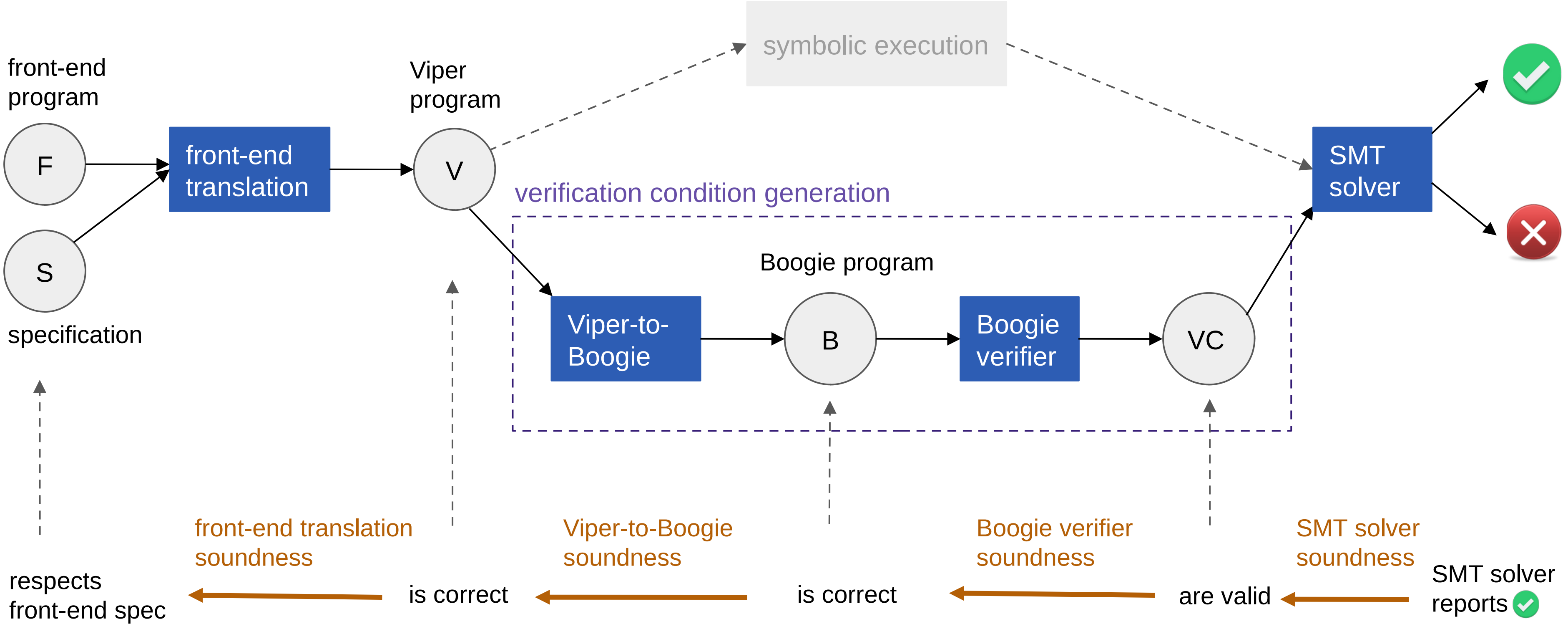
Soundness: Proof Strategy



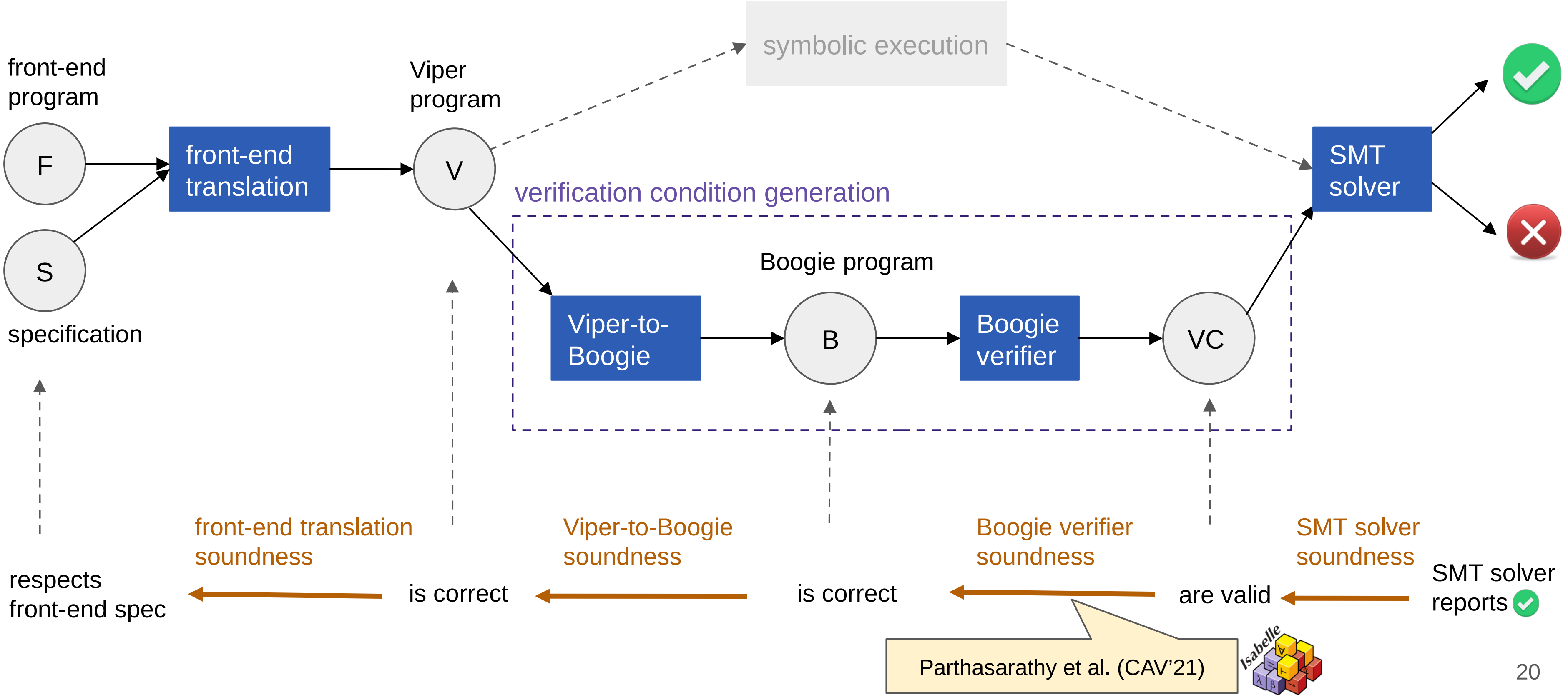
Soundness: Proof Strategy



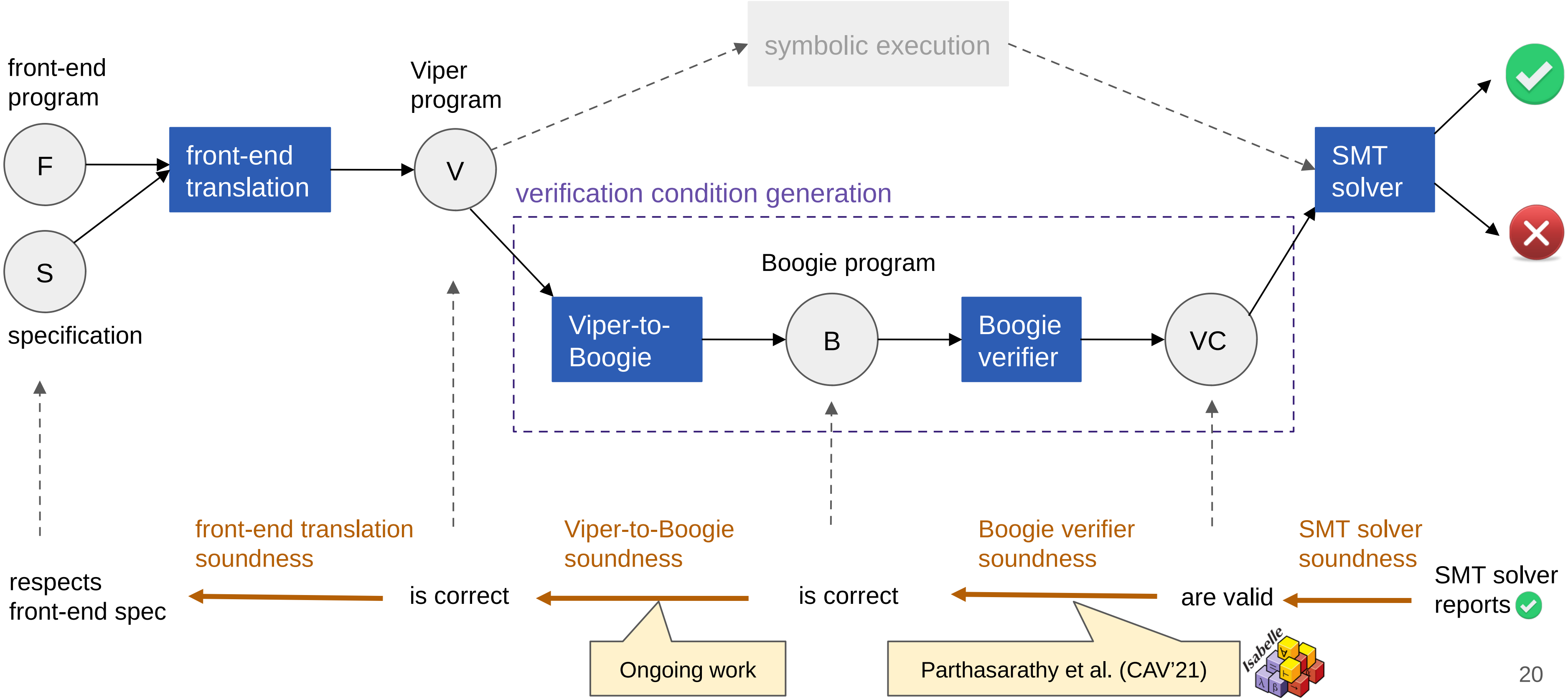
Soundness: Proof Strategy



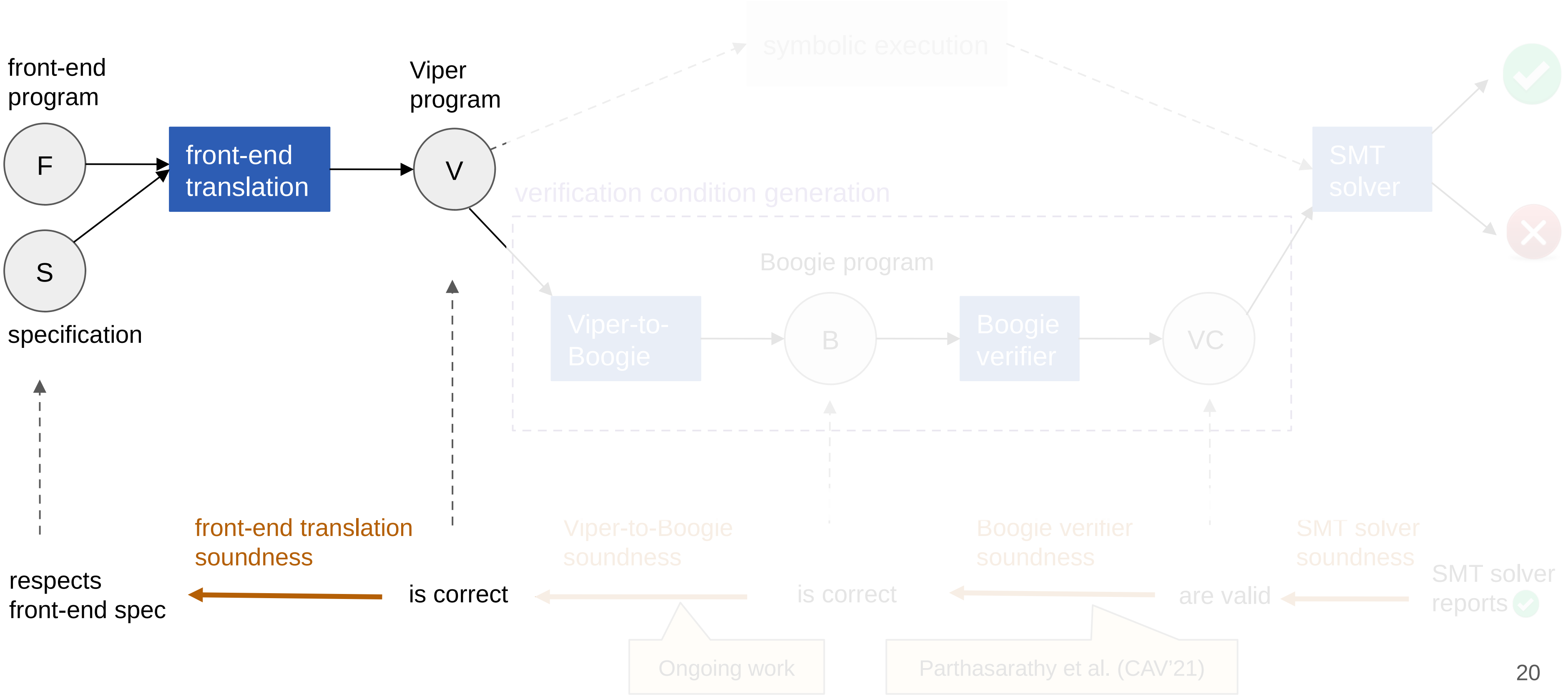
Soundness: Proof Strategy



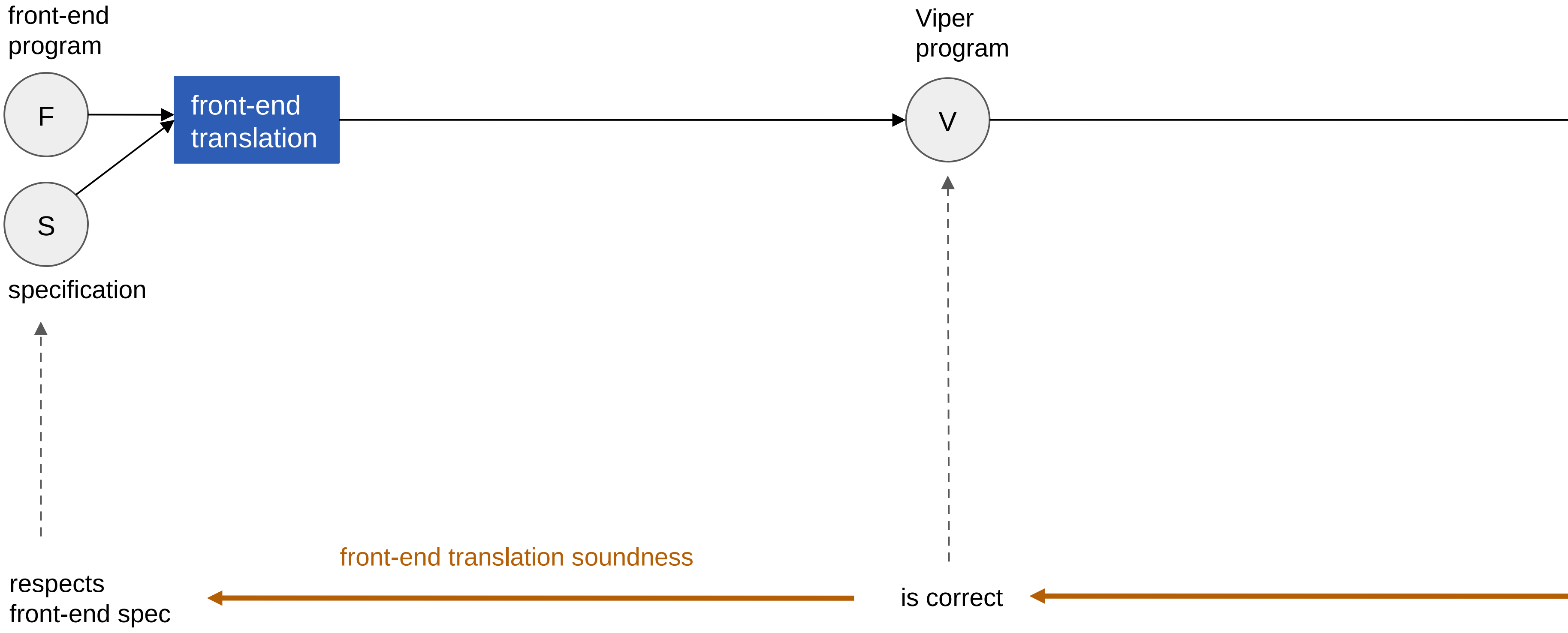
Soundness: Proof Strategy



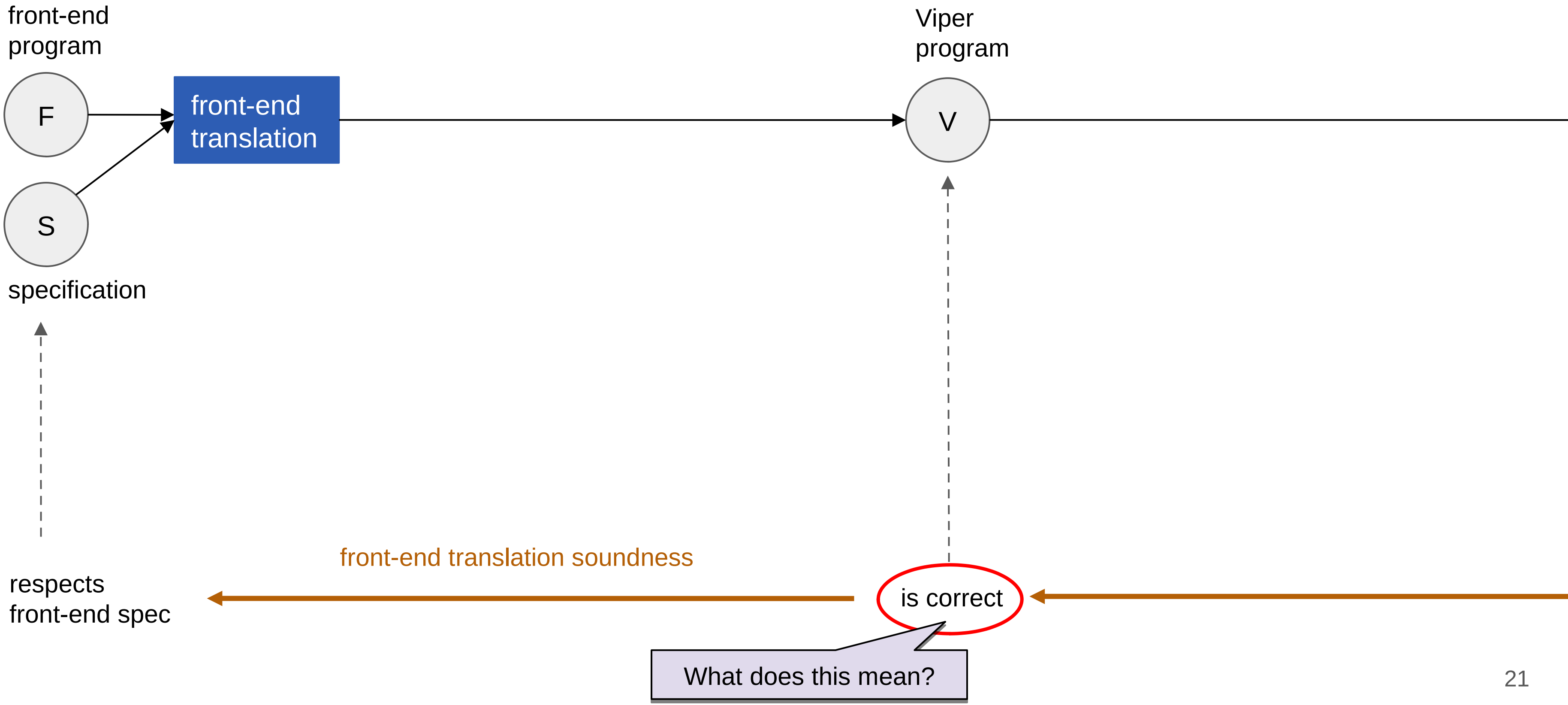
Soundness: Proof Strategy



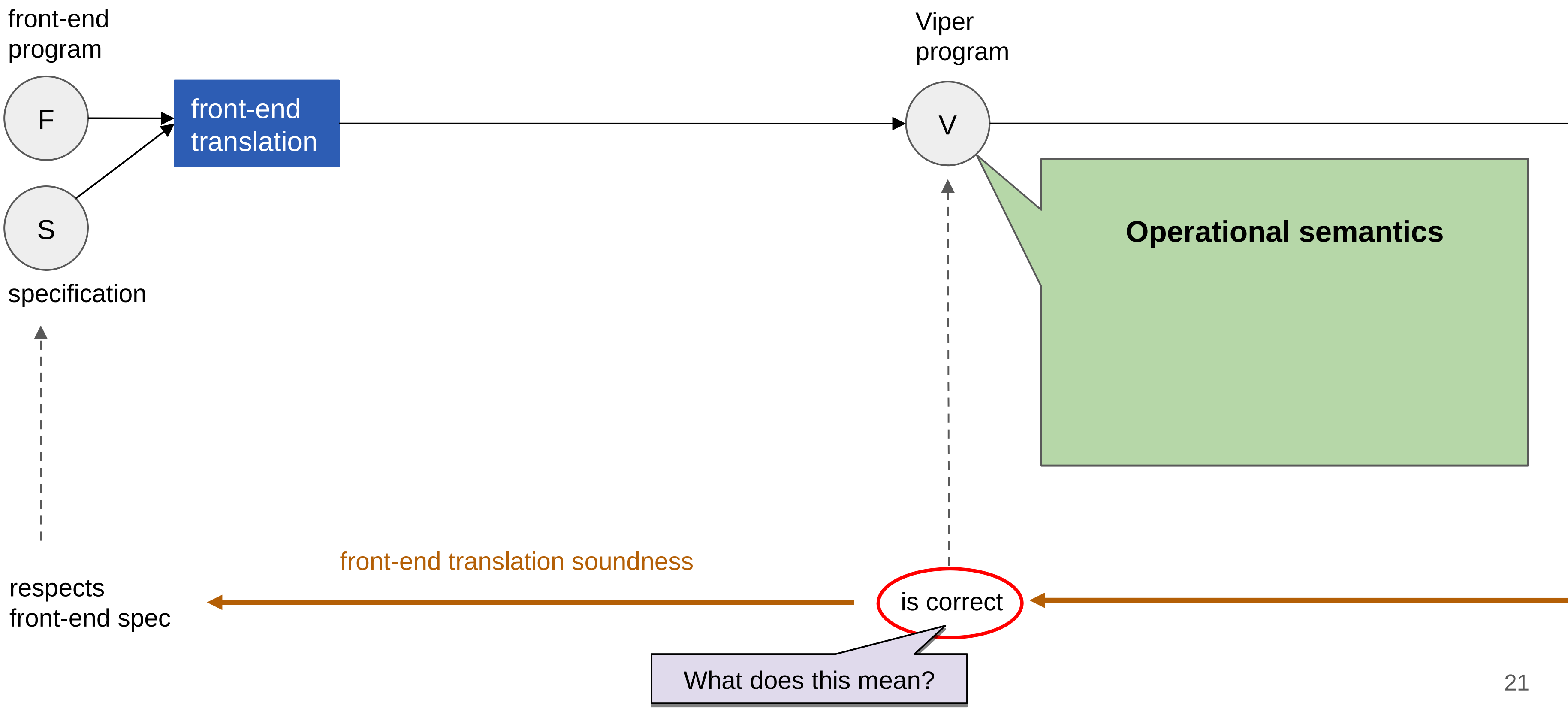
Operational Semantics and Adequacy Theorem



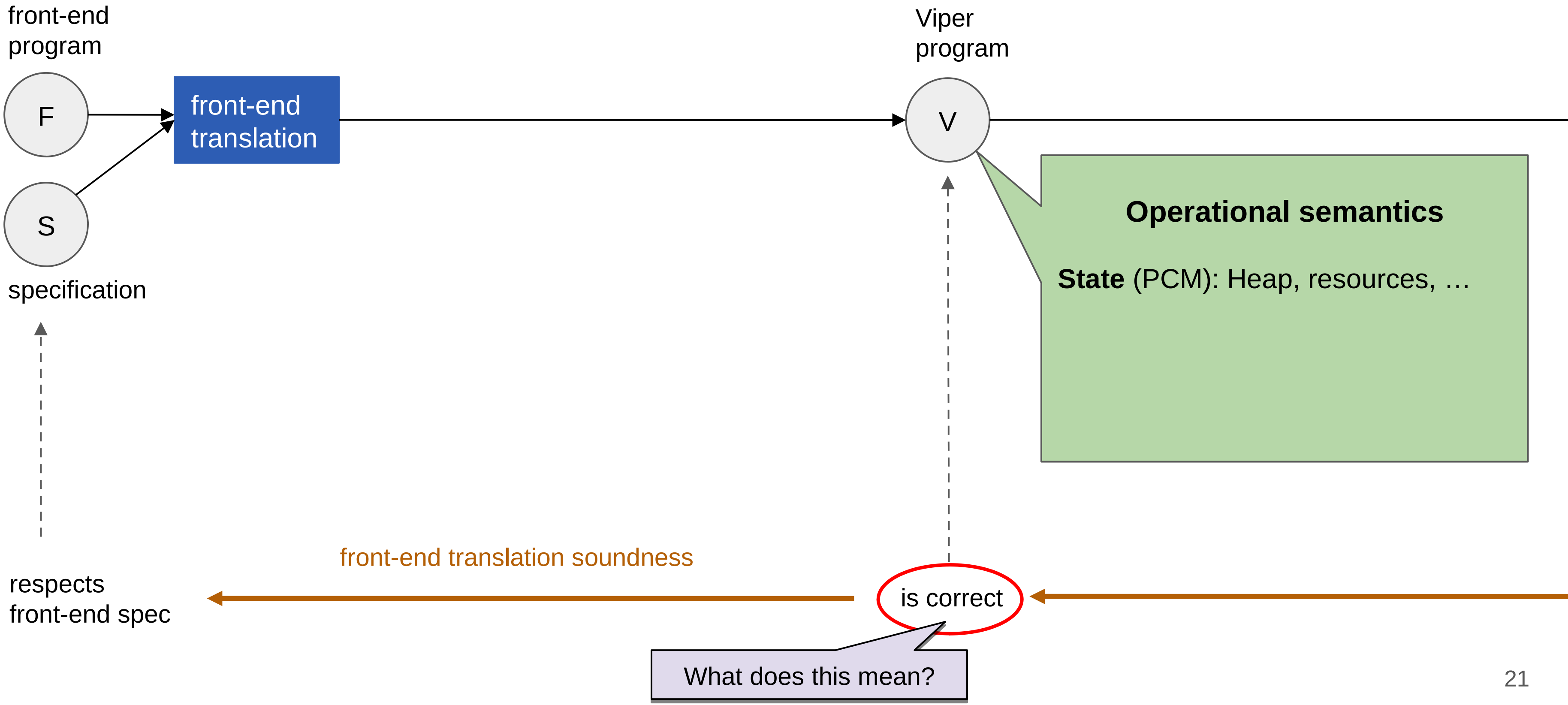
Operational Semantics and Adequacy Theorem



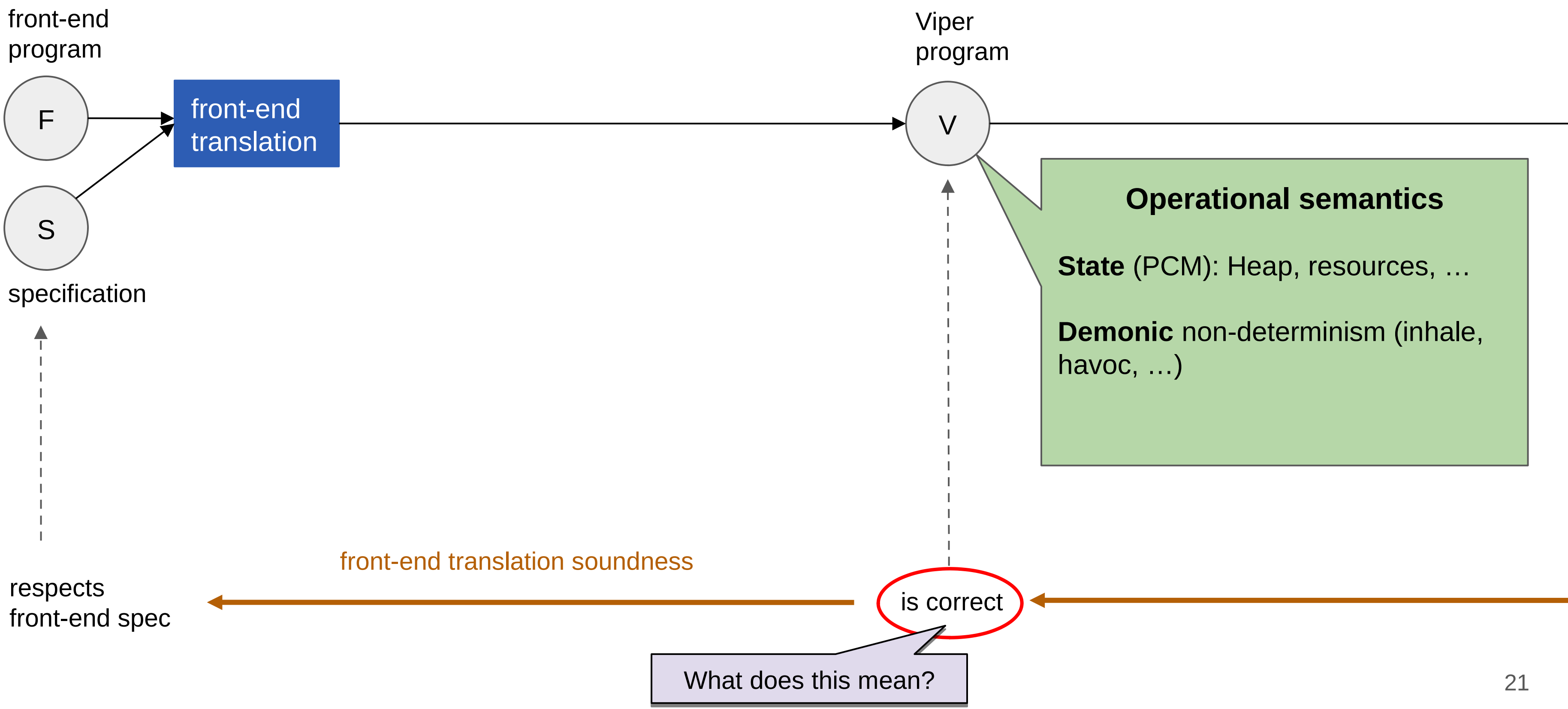
Operational Semantics and Adequacy Theorem



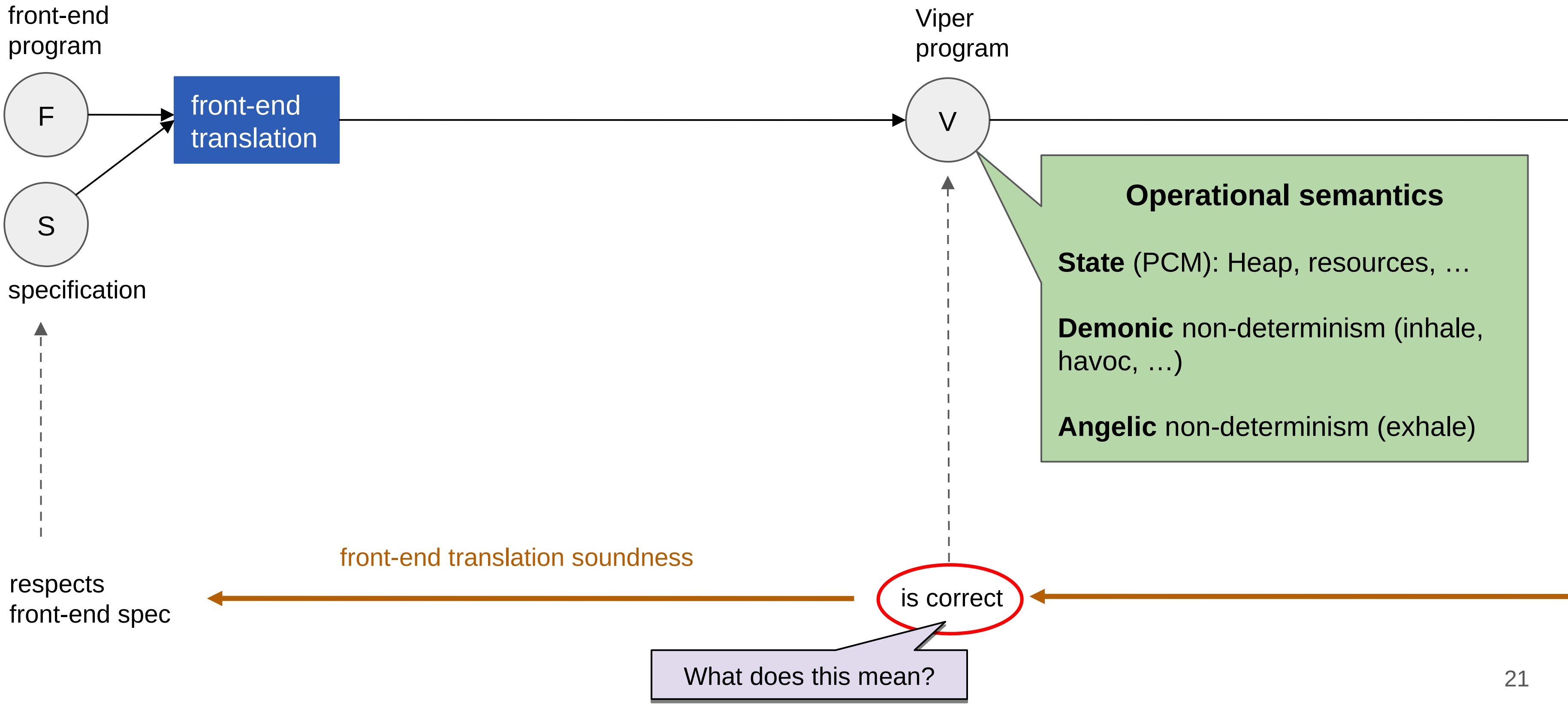
Operational Semantics and Adequacy Theorem



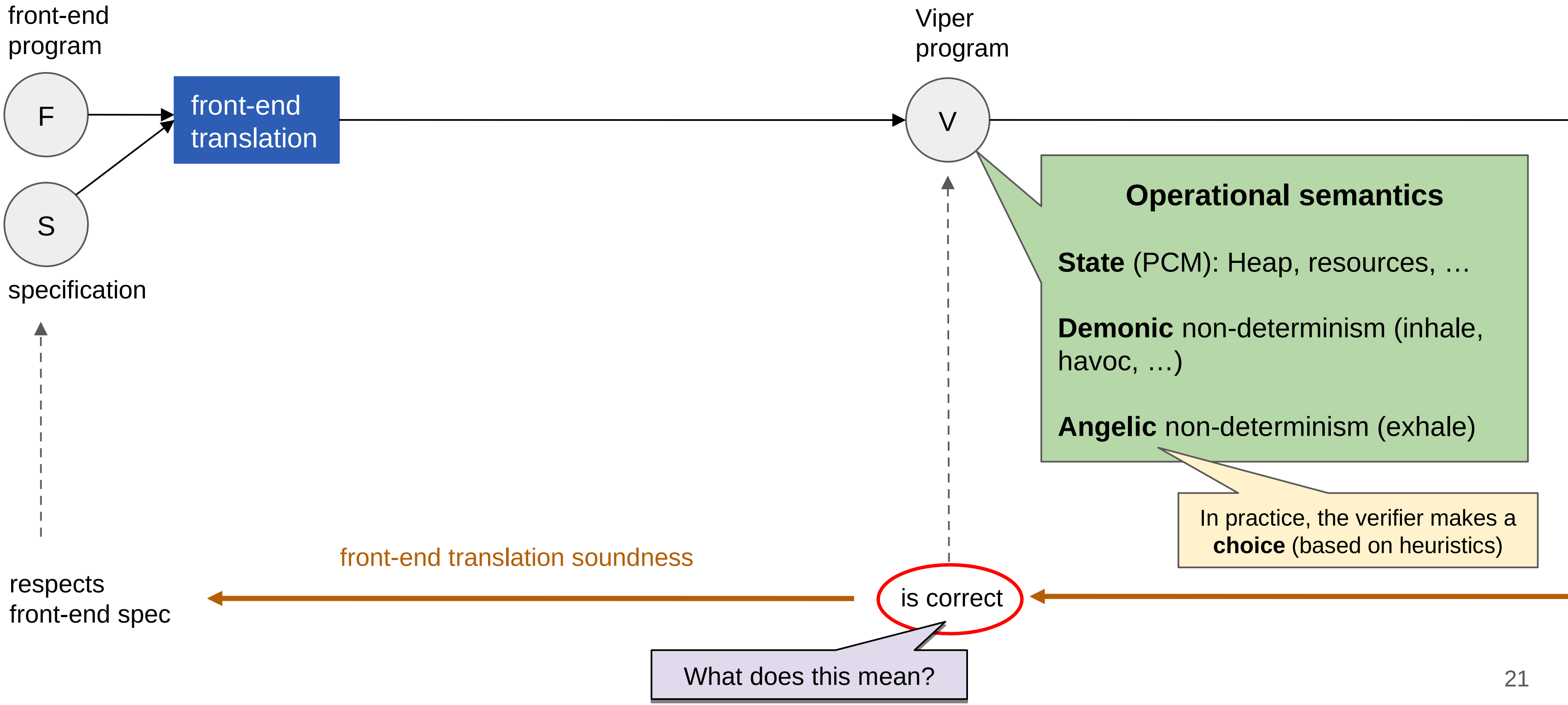
Operational Semantics and Adequacy Theorem



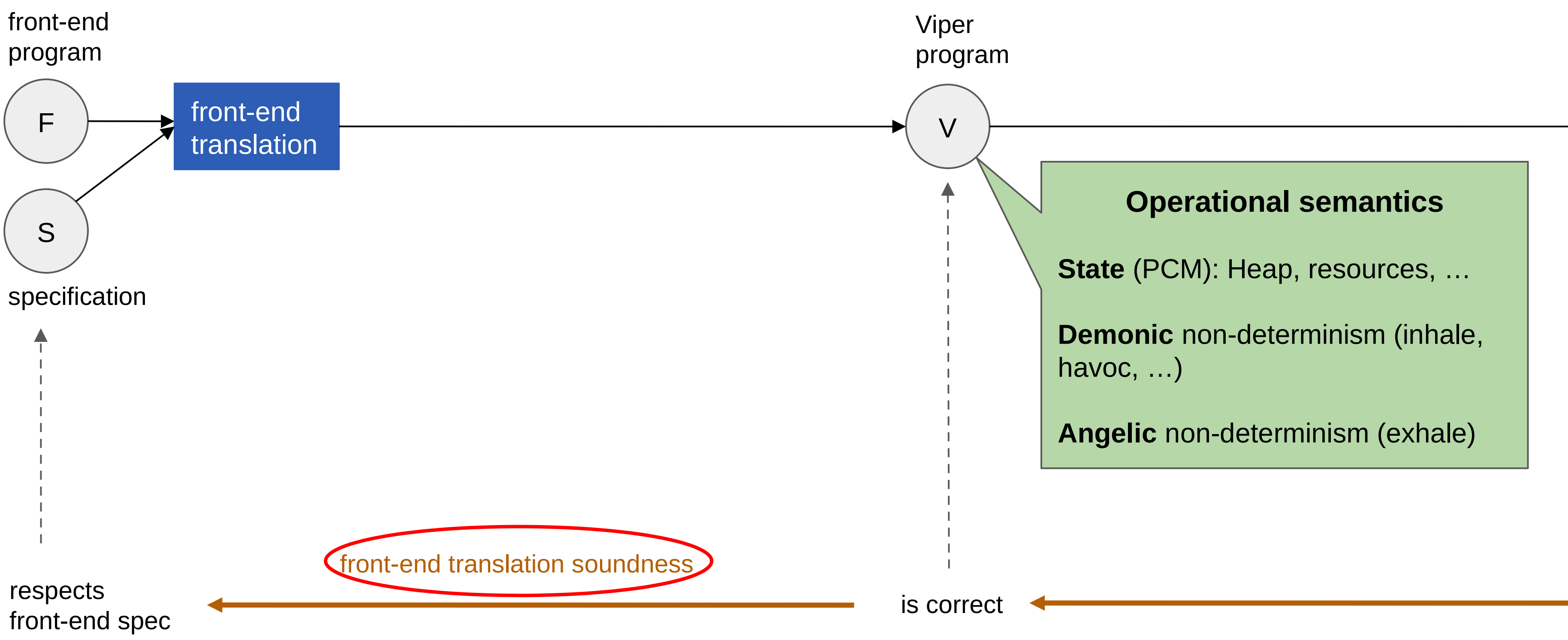
Operational Semantics and Adequacy Theorem



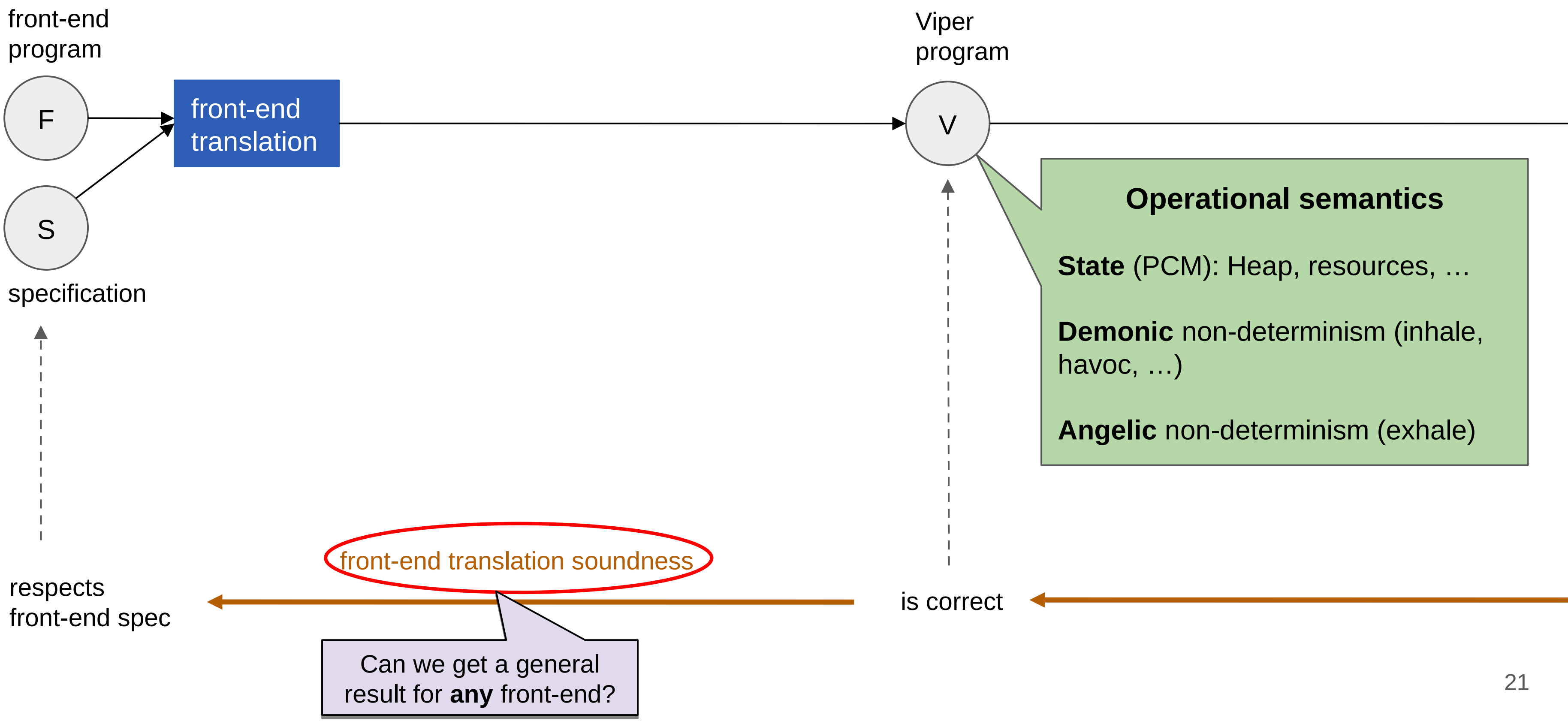
Operational Semantics and Adequacy Theorem



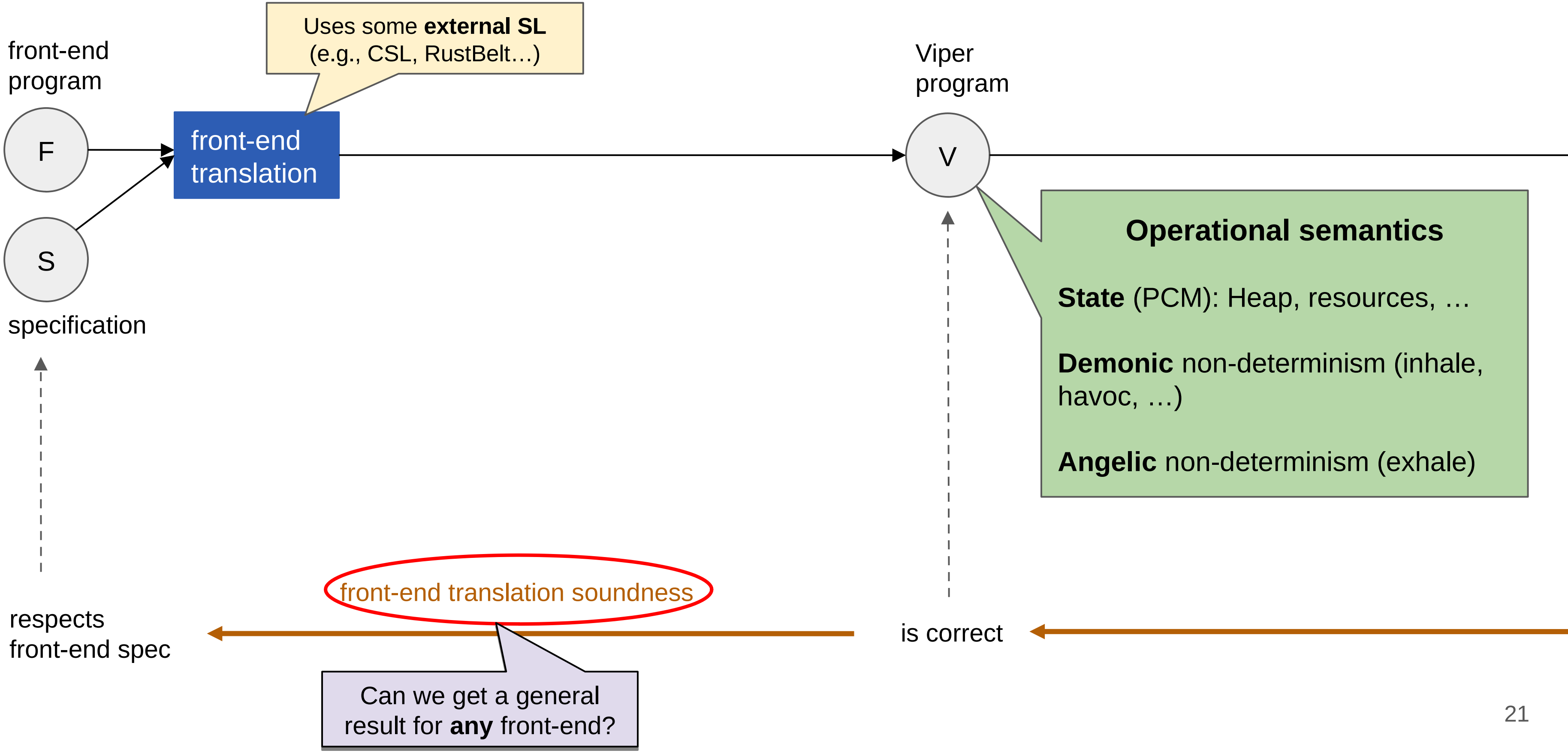
Operational Semantics and Adequacy Theorem



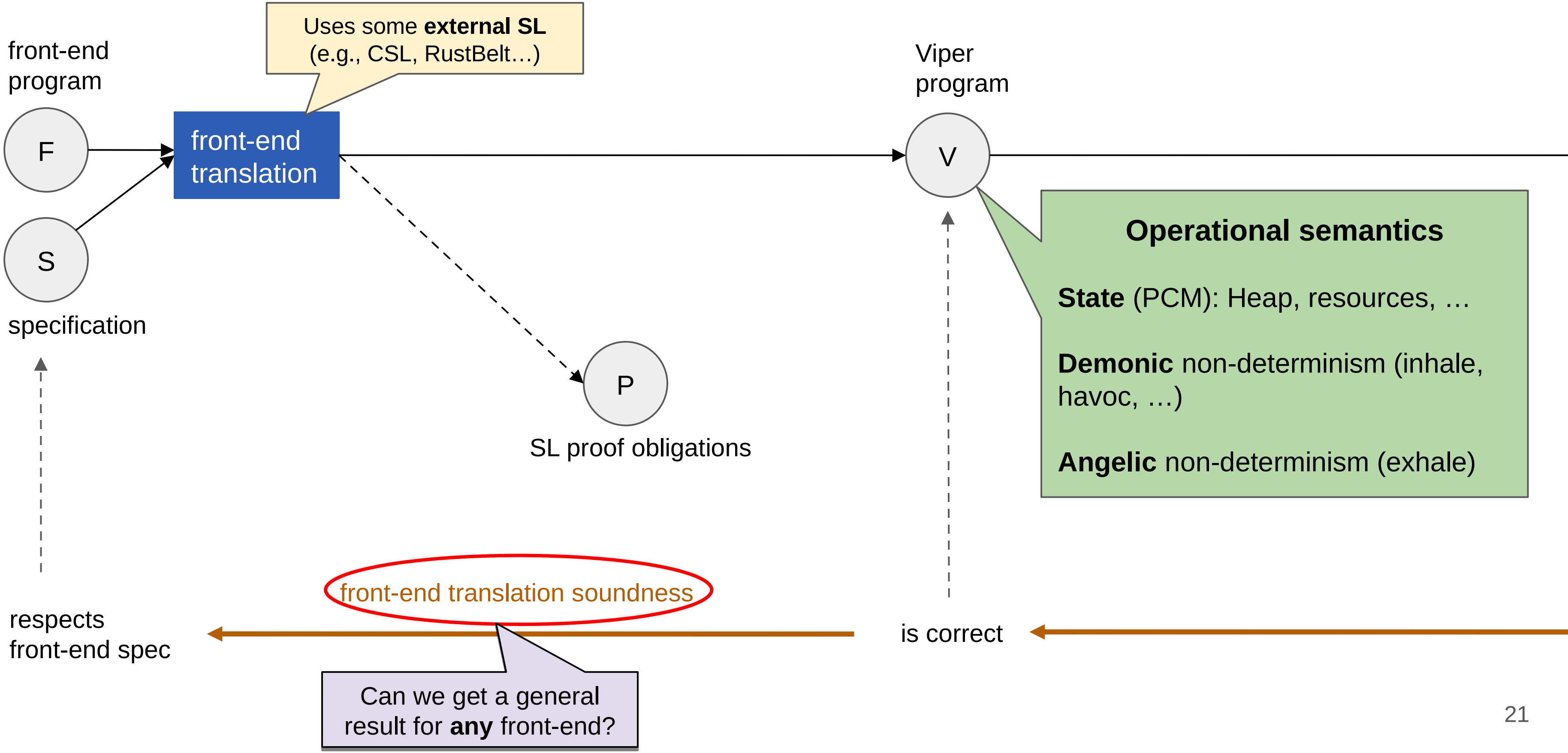
Operational Semantics and Adequacy Theorem



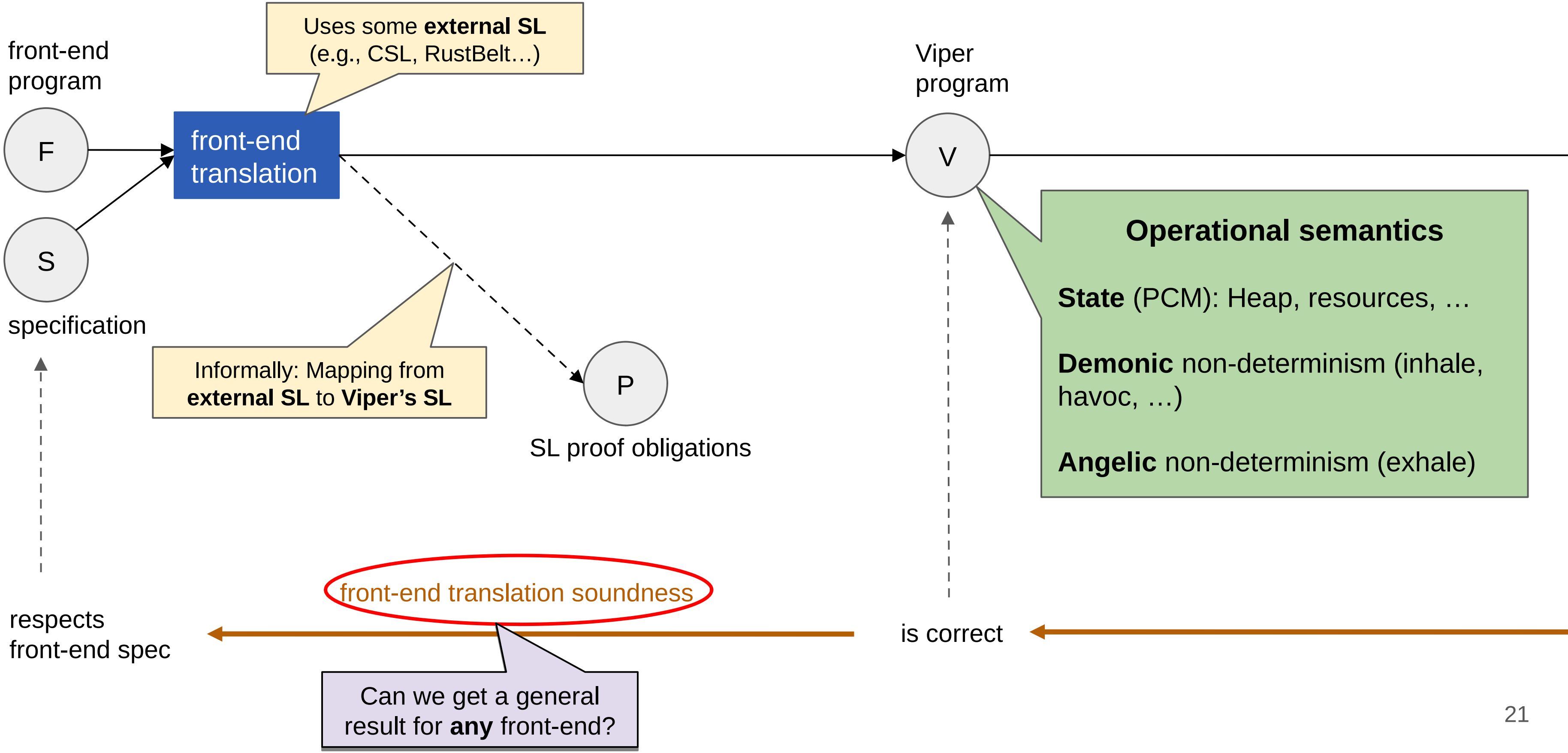
Operational Semantics and Adequacy Theorem



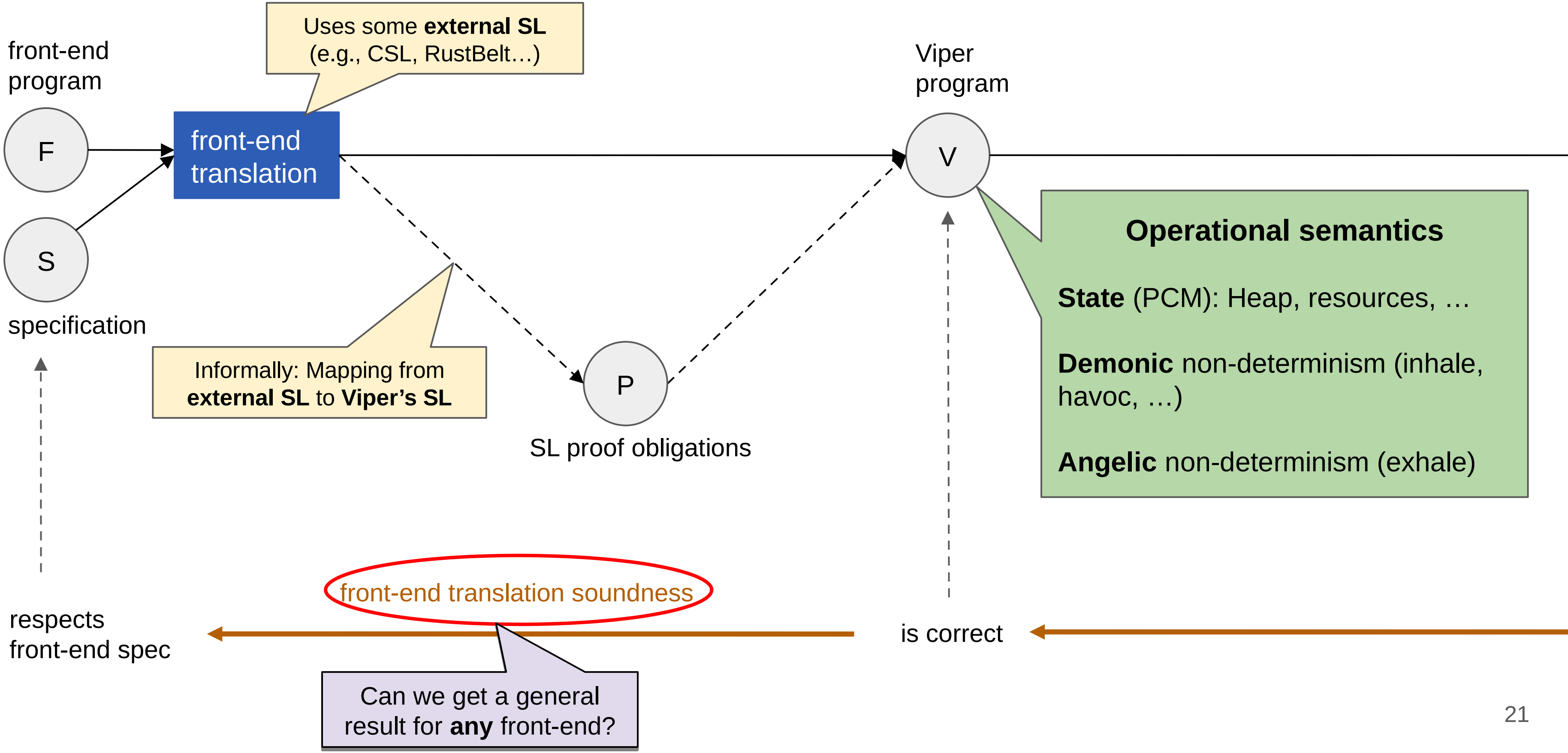
Operational Semantics and Adequacy Theorem



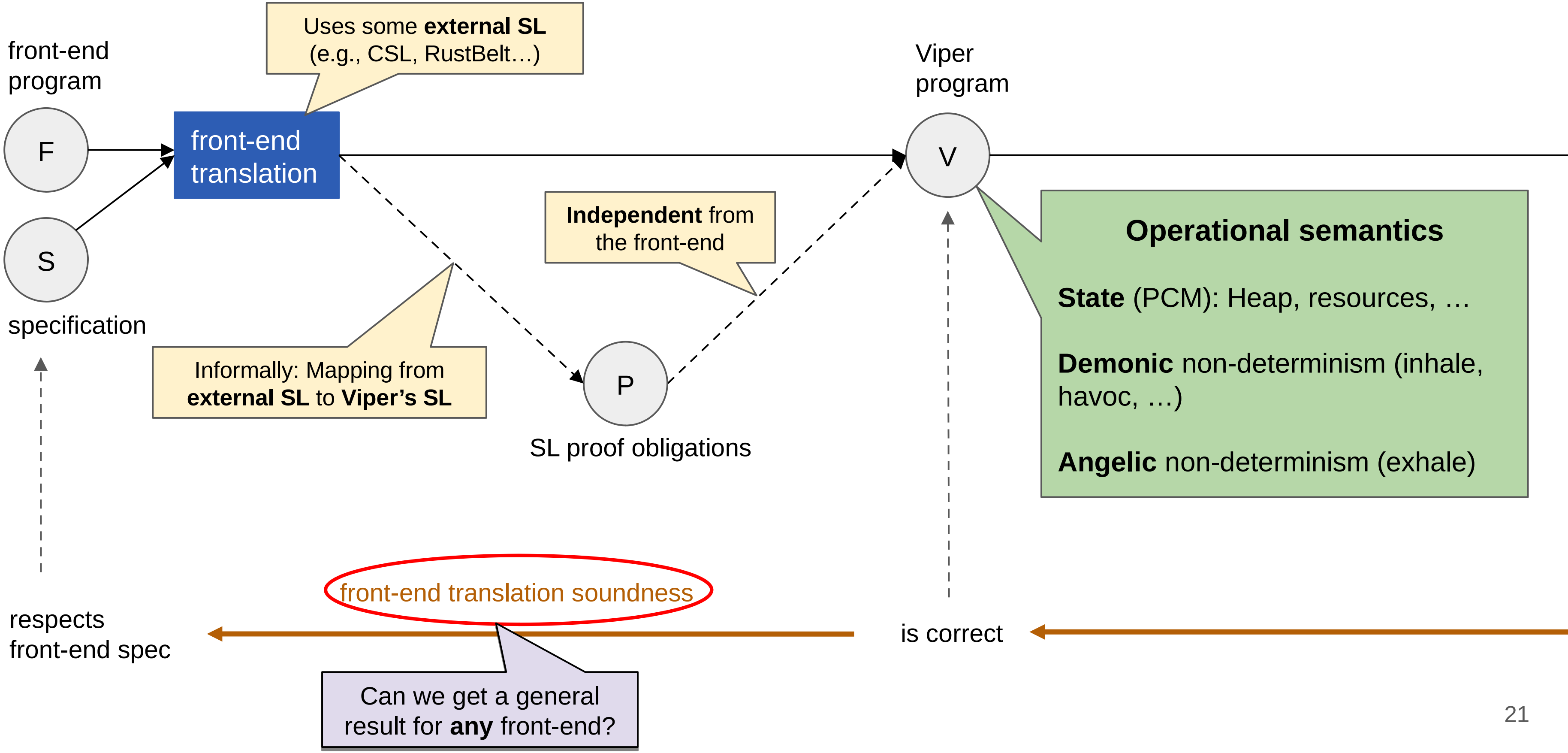
Operational Semantics and Adequacy Theorem



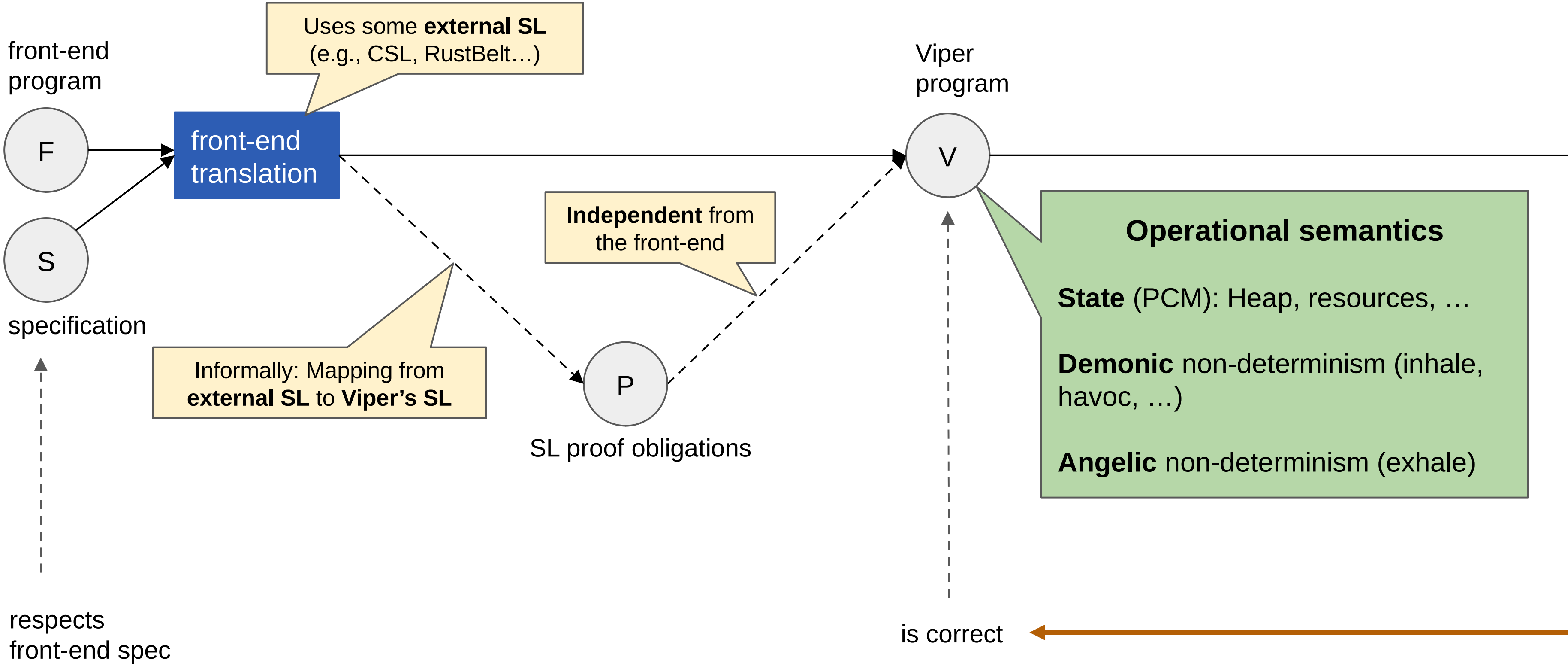
Operational Semantics and Adequacy Theorem



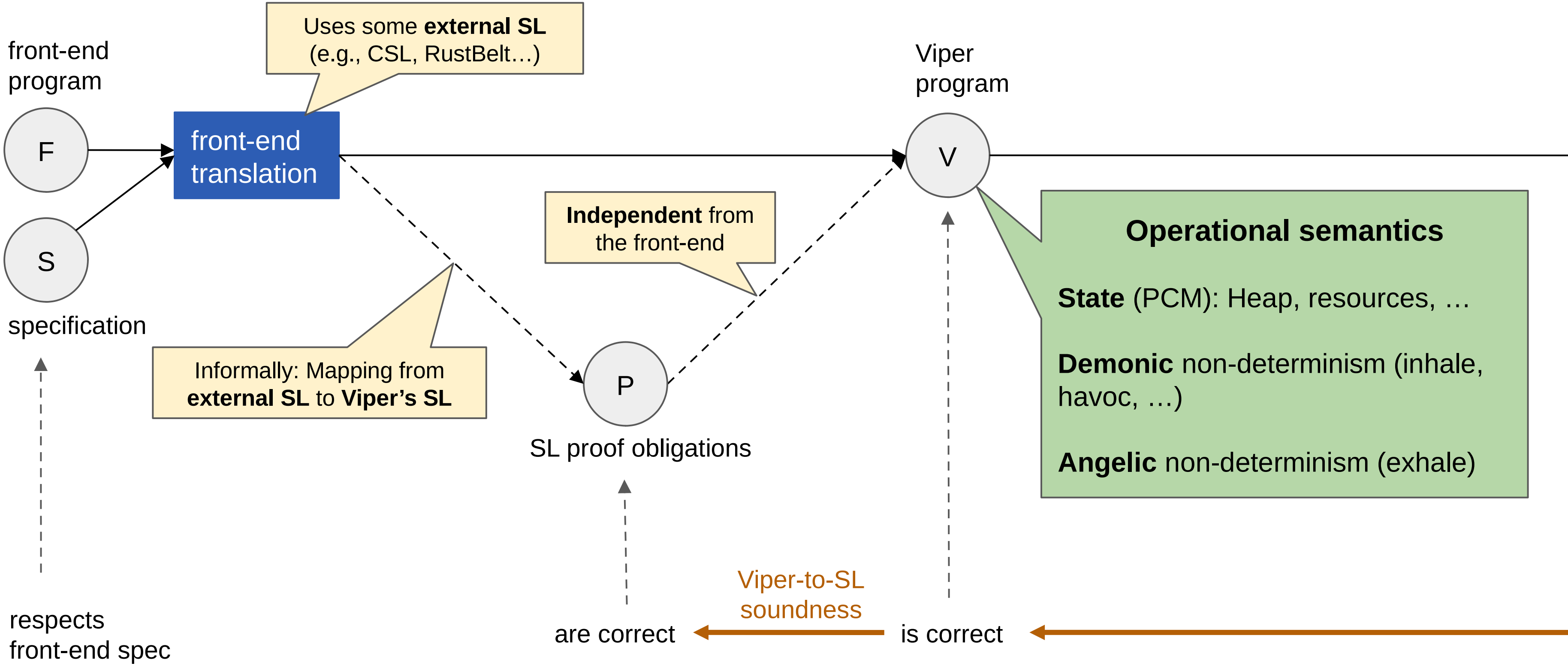
Operational Semantics and Adequacy Theorem



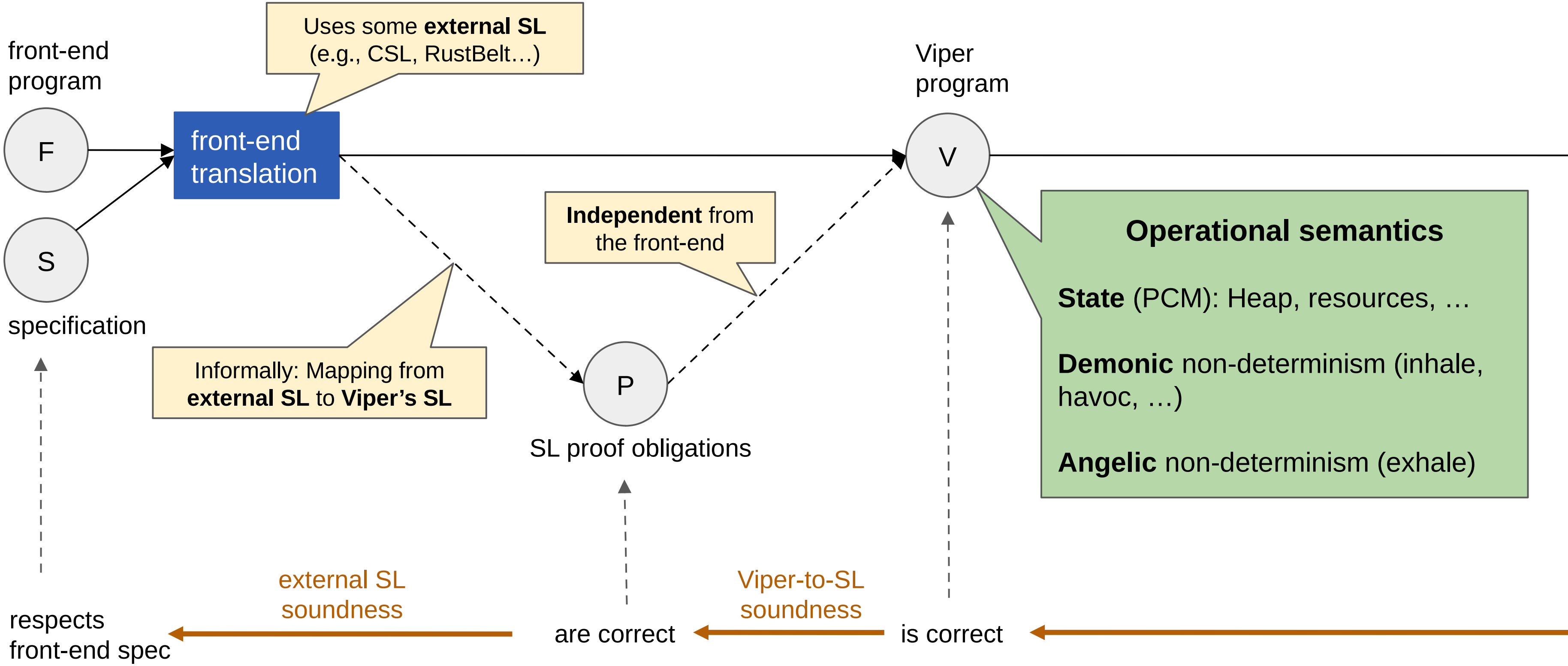
Operational Semantics and Adequacy Theorem



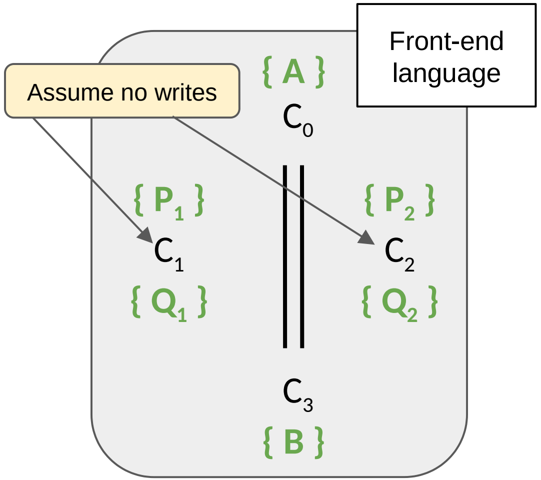
Operational Semantics and Adequacy Theorem



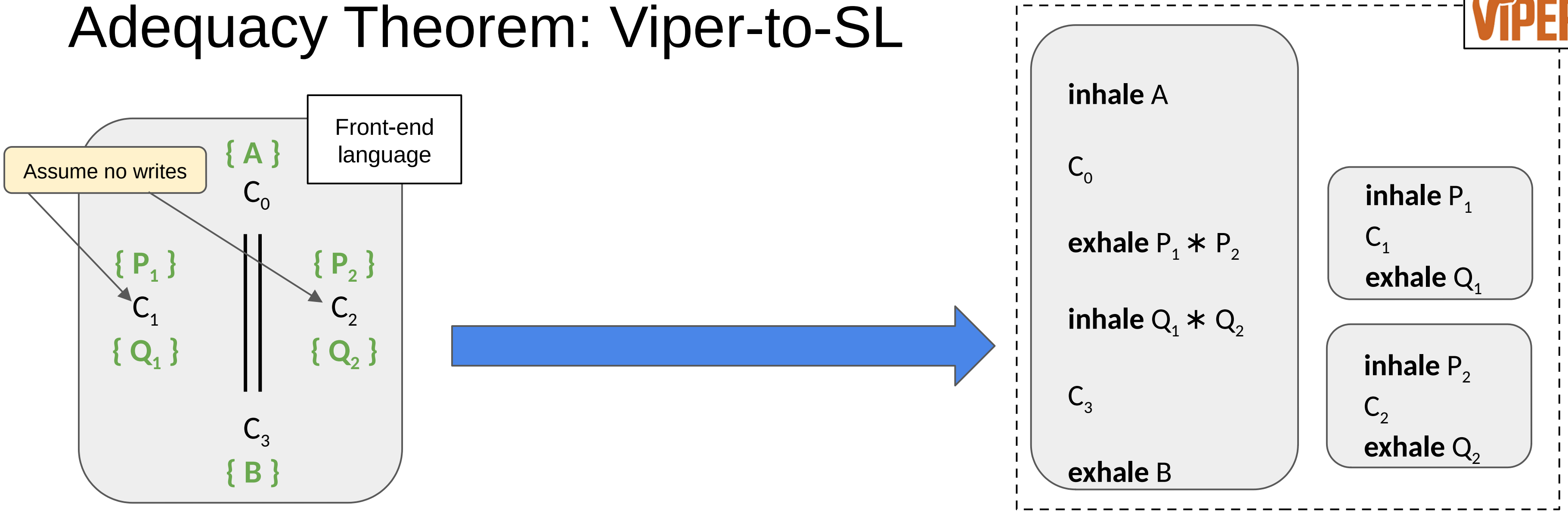
Operational Semantics and Adequacy Theorem



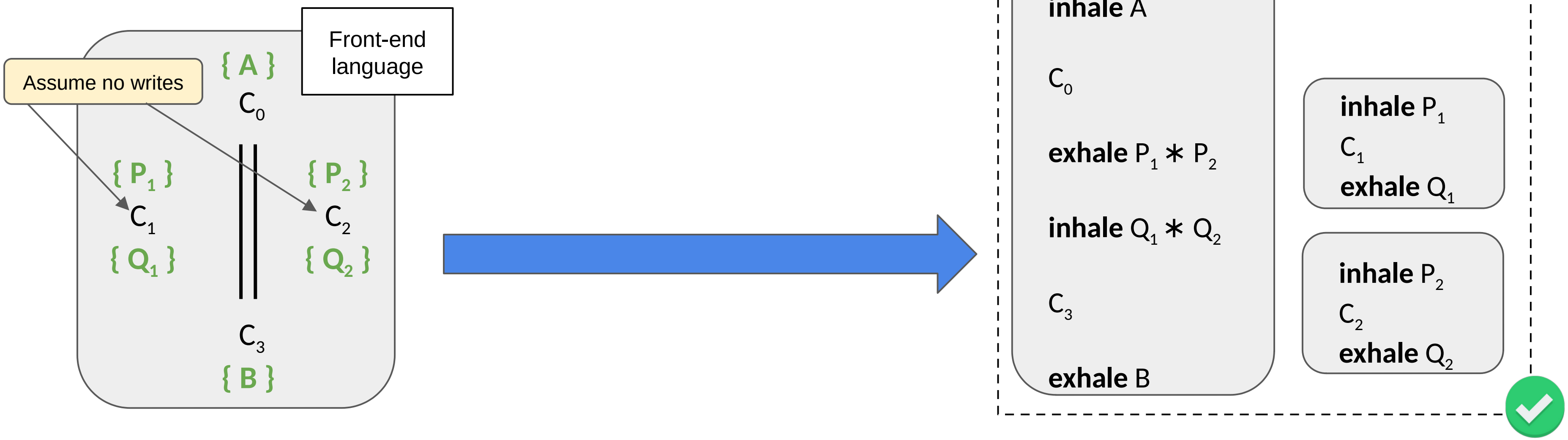
Adequacy Theorem: Viper-to-SL



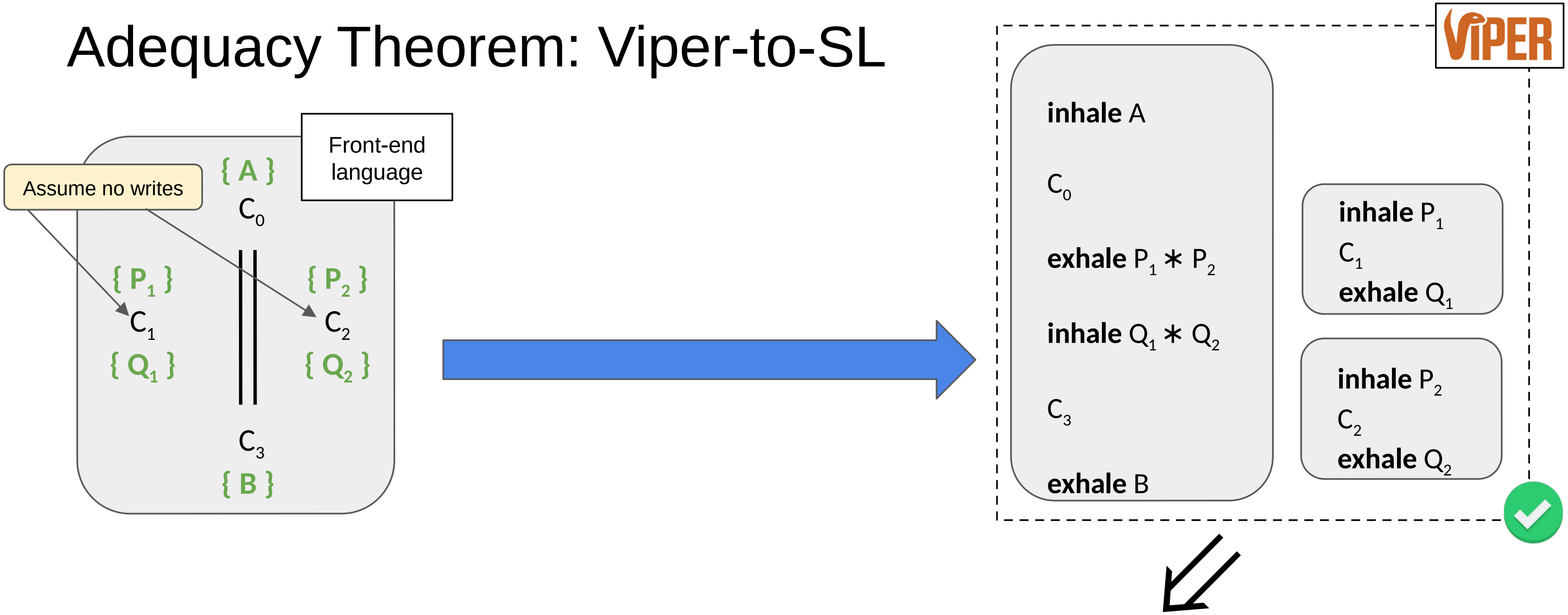
Adequacy Theorem: Viper-to-SL



Adequacy Theorem: Viper-to-SL

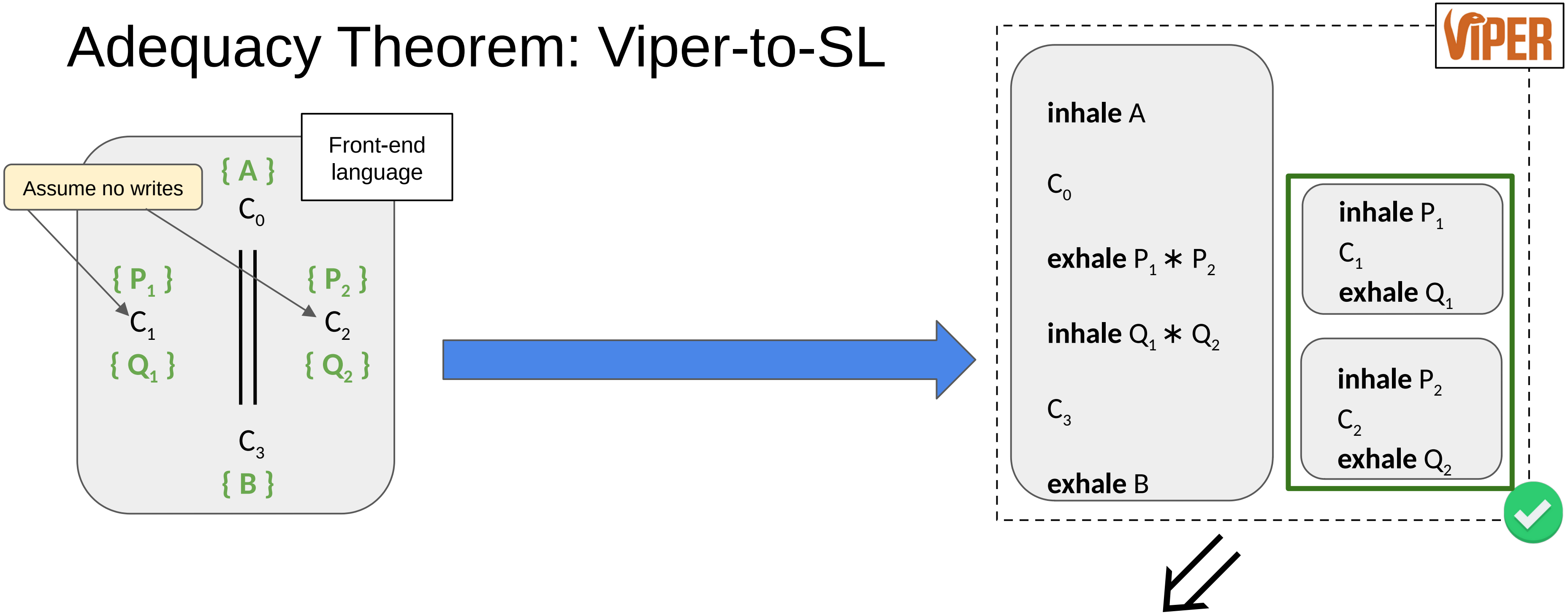


Adequacy Theorem: Viper-to-SL



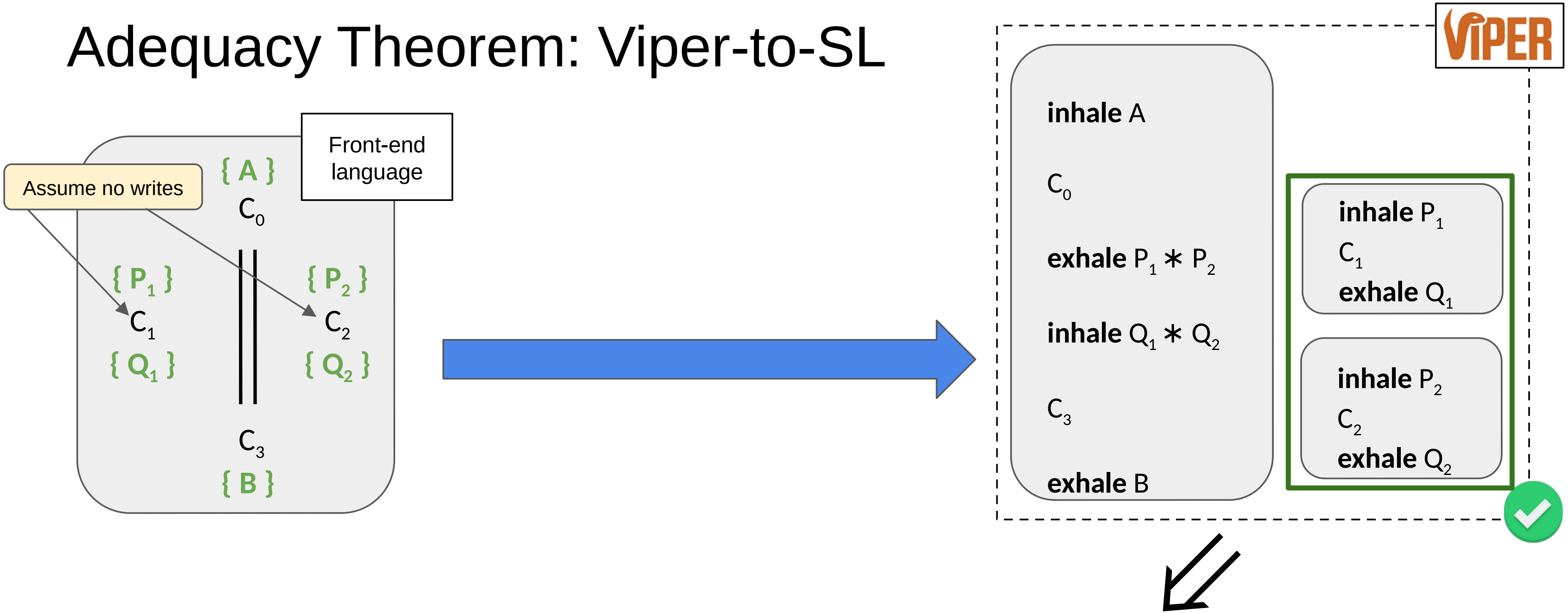
SL proof obligations proven by Viper

Adequacy Theorem: Viper-to-SL



SL proof obligations proven by Viper

Adequacy Theorem: Viper-to-SL

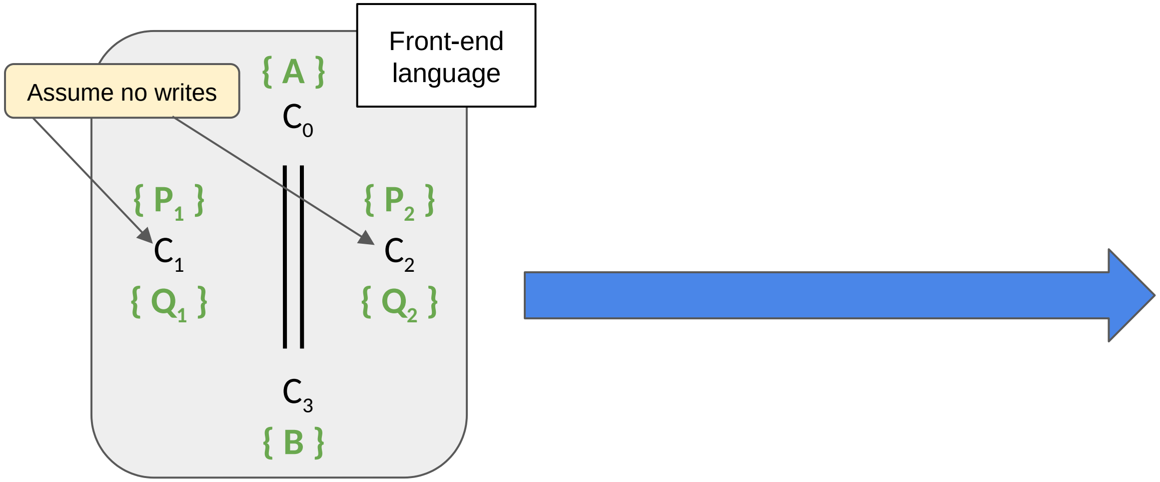


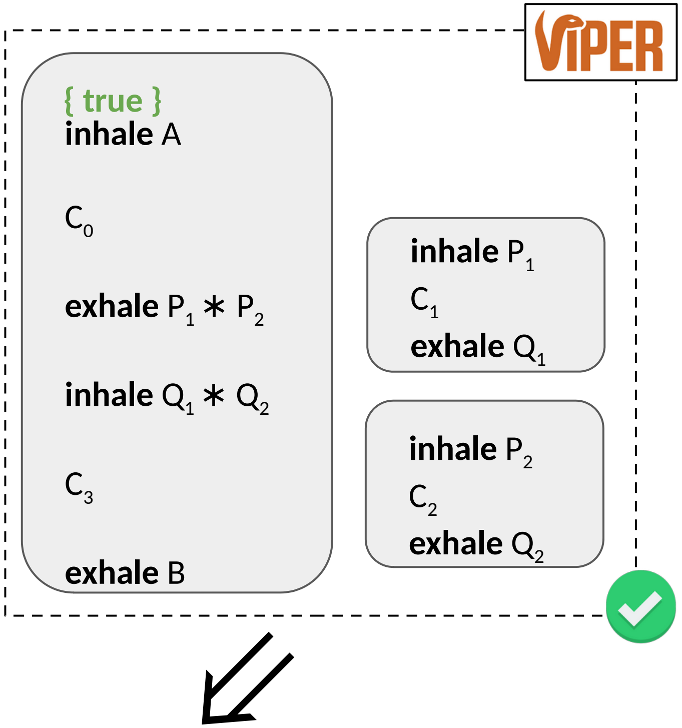
SL proof obligations proven by Viper

$\vdash \{ P_1 \}\ C_1\ \{ Q_1 \}$

$\vdash \{ P_2 \}\ C_2\ \{ Q_2 \}$

Adequacy Theorem: Viper-to-SL



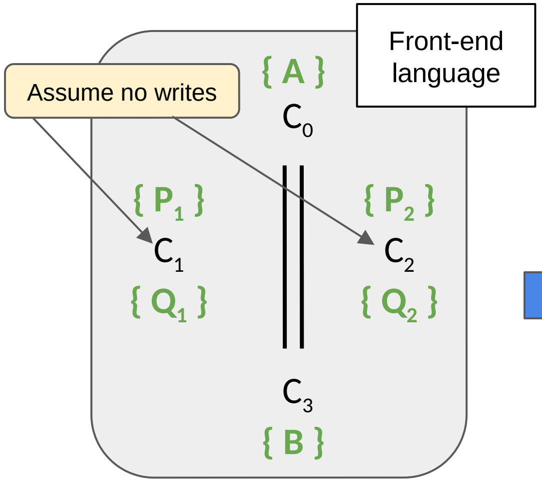


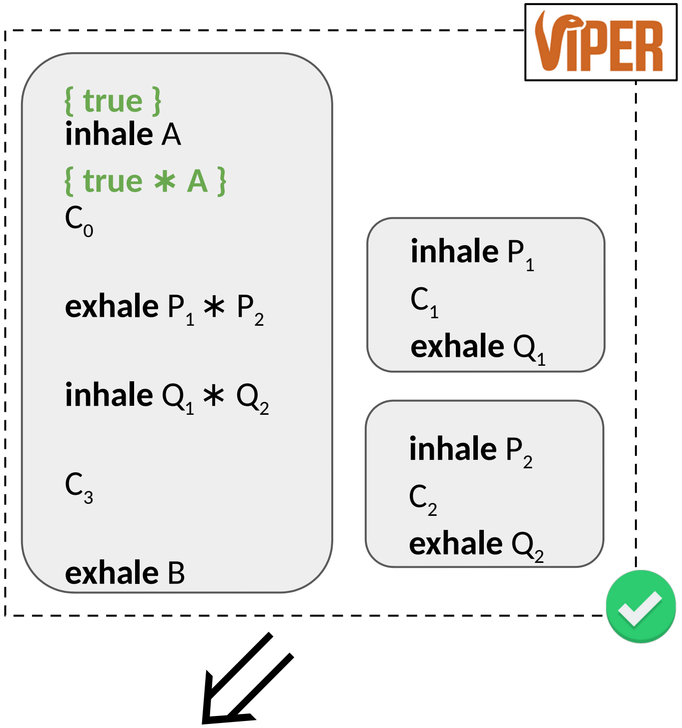
SL proof obligations proven by Viper

$\vdash \{ P_1 \}\ C_1\ \{ Q_1 \}$

$\vdash \{ P_2 \}\ C_2\ \{ Q_2 \}$

Adequacy Theorem: Viper-to-SL



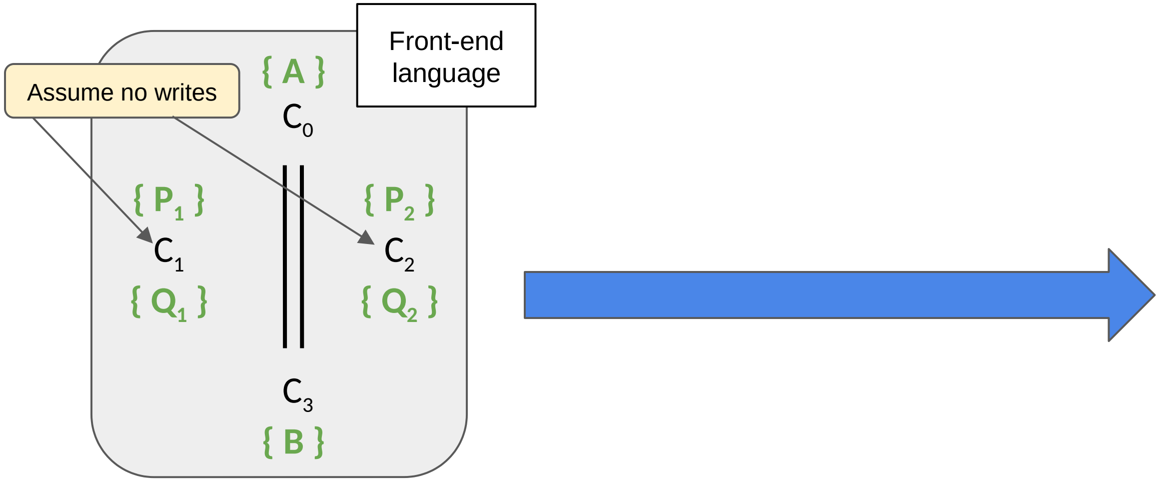


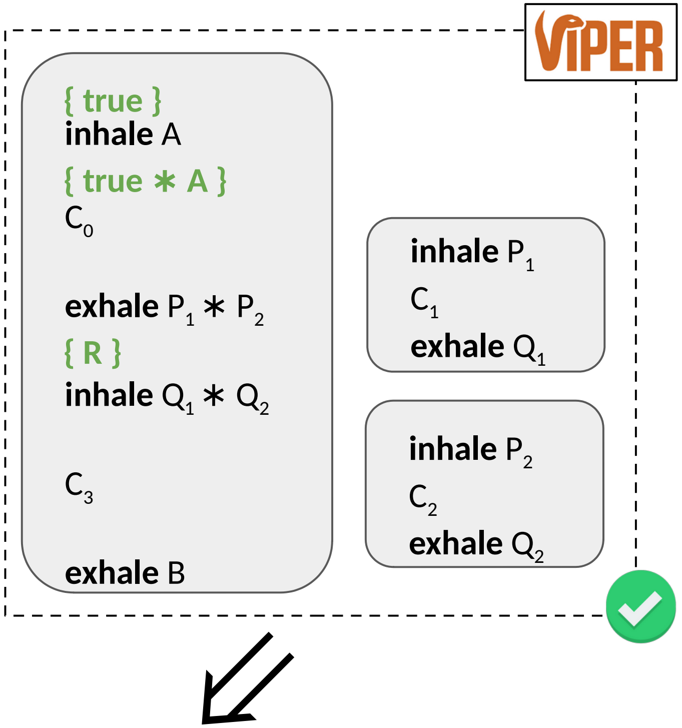
SL proof obligations proven by Viper

$\vdash \{ P_1 \}\ C_1\ \{ Q_1 \}$

$\vdash \{ P_2 \}\ C_2\ \{ Q_2 \}$

Adequacy Theorem: Viper-to-SL



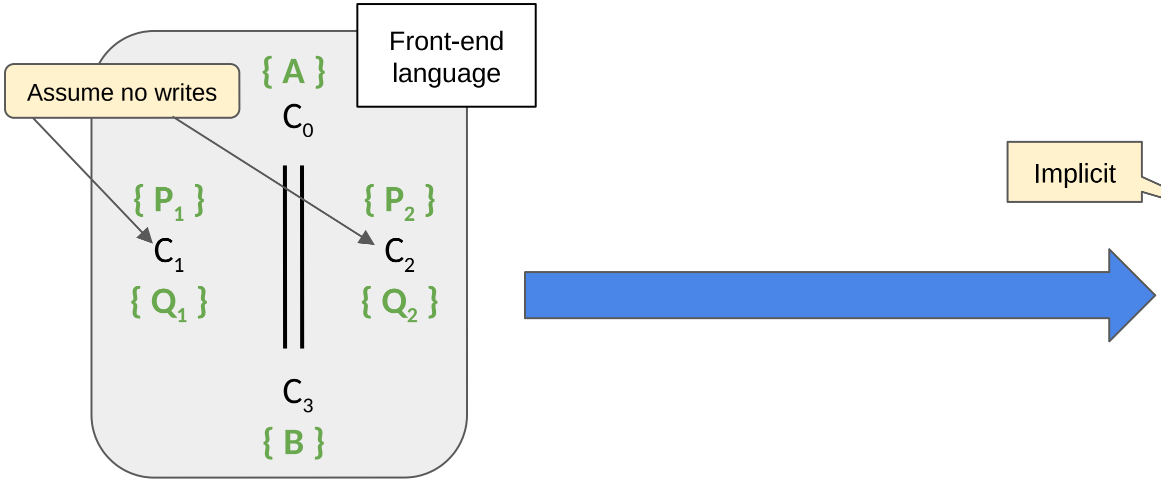


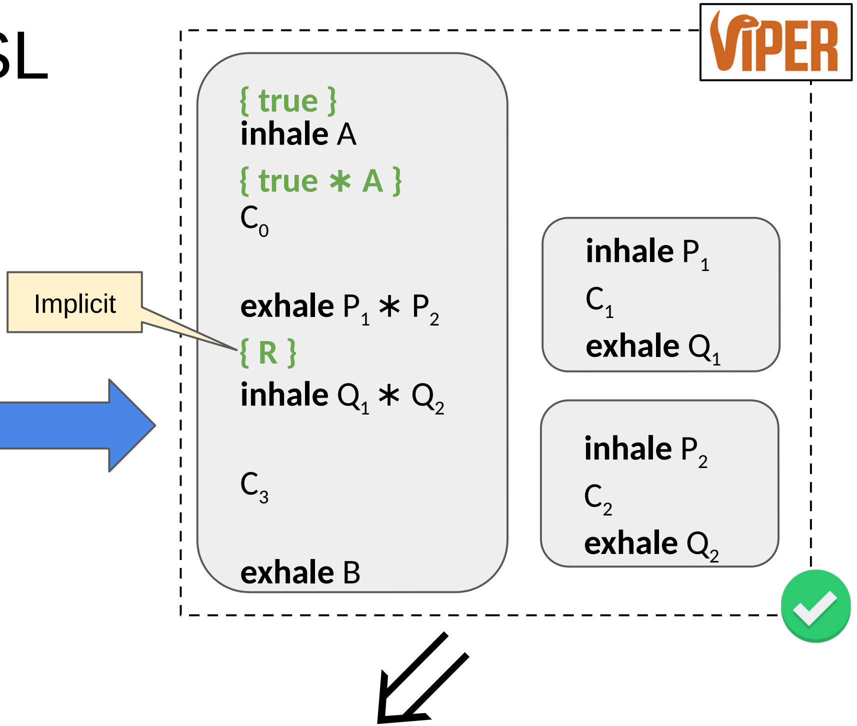
SL proof obligations proven by Viper

$\vdash \{ P_1 \}\ C_1\ \{ Q_1 \}$

$\vdash \{ P_2 \}\ C_2\ \{ Q_2 \}$

Adequacy Theorem: Viper-to-SL



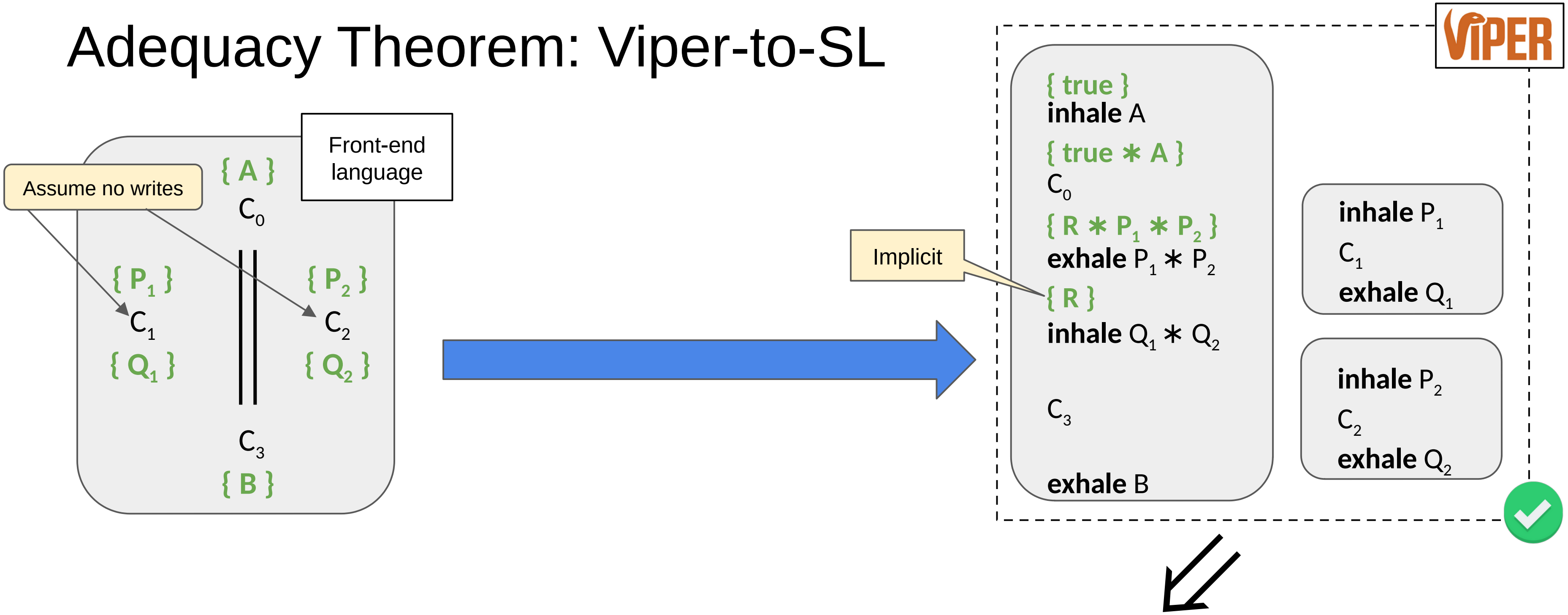


SL proof obligations proven by Viper

$\vdash \{ P_1 \}\ C_1\ \{ Q_1 \}$

$\vdash \{ P_2 \}\ C_2\ \{ Q_2 \}$

Adequacy Theorem: Viper-to-SL

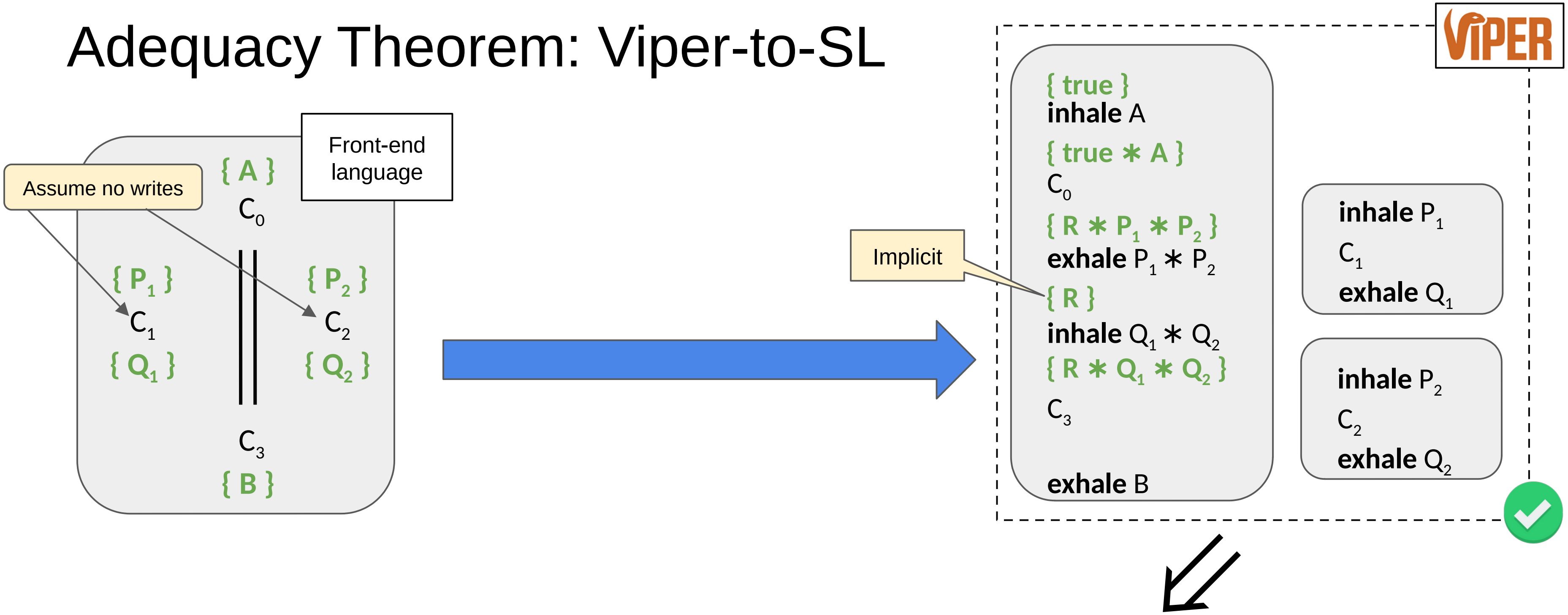


SL proof obligations proven by Viper

$\vdash \{ P_1 \}\ C_1\ \{ Q_1 \}$

$\vdash \{ P_2 \}\ C_2\ \{ Q_2 \}$

Adequacy Theorem: Viper-to-SL

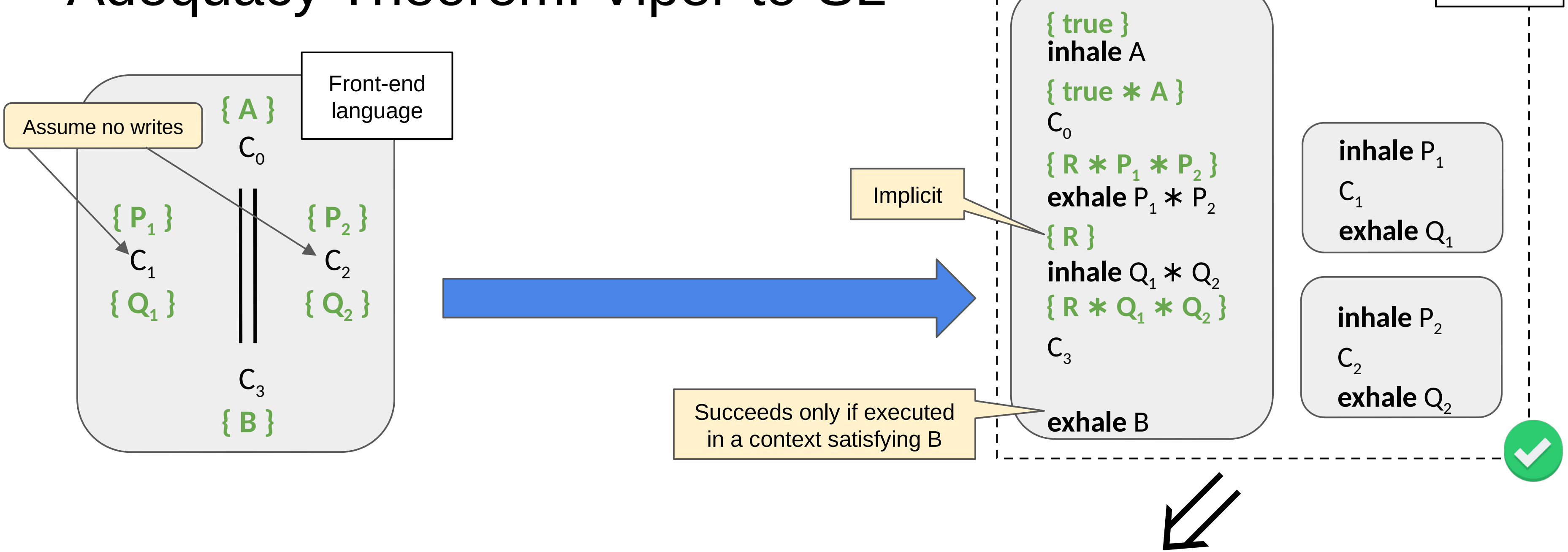


SL proof obligations proven by Viper

$\vdash \{ P_1 \}\ C_1\ \{ Q_1 \}$

$\vdash \{ P_2 \}\ C_2\ \{ Q_2 \}$

Adequacy Theorem: Viper-to-SL

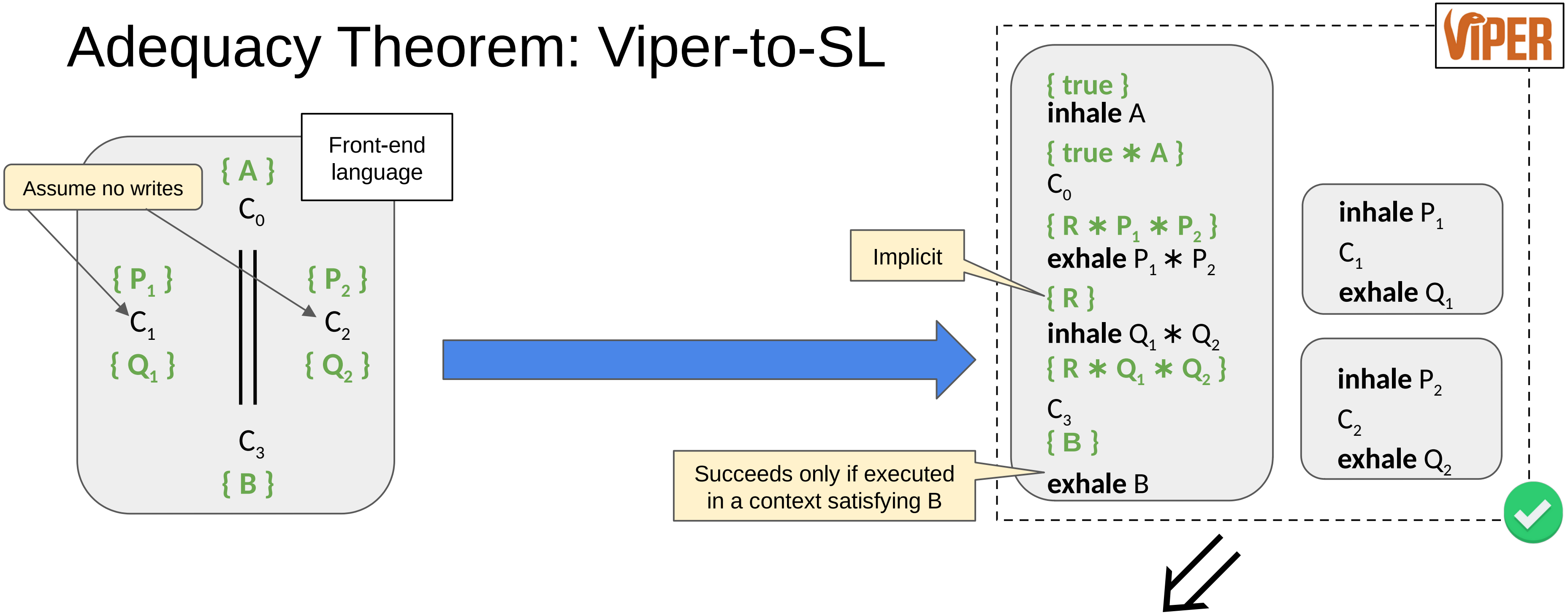


SL proof obligations proven by Viper

$\vdash \{ P_1 \}\ C_1\ \{ Q_1 \}$

$\vdash \{ P_2 \}\ C_2\ \{ Q_2 \}$

Adequacy Theorem: Viper-to-SL

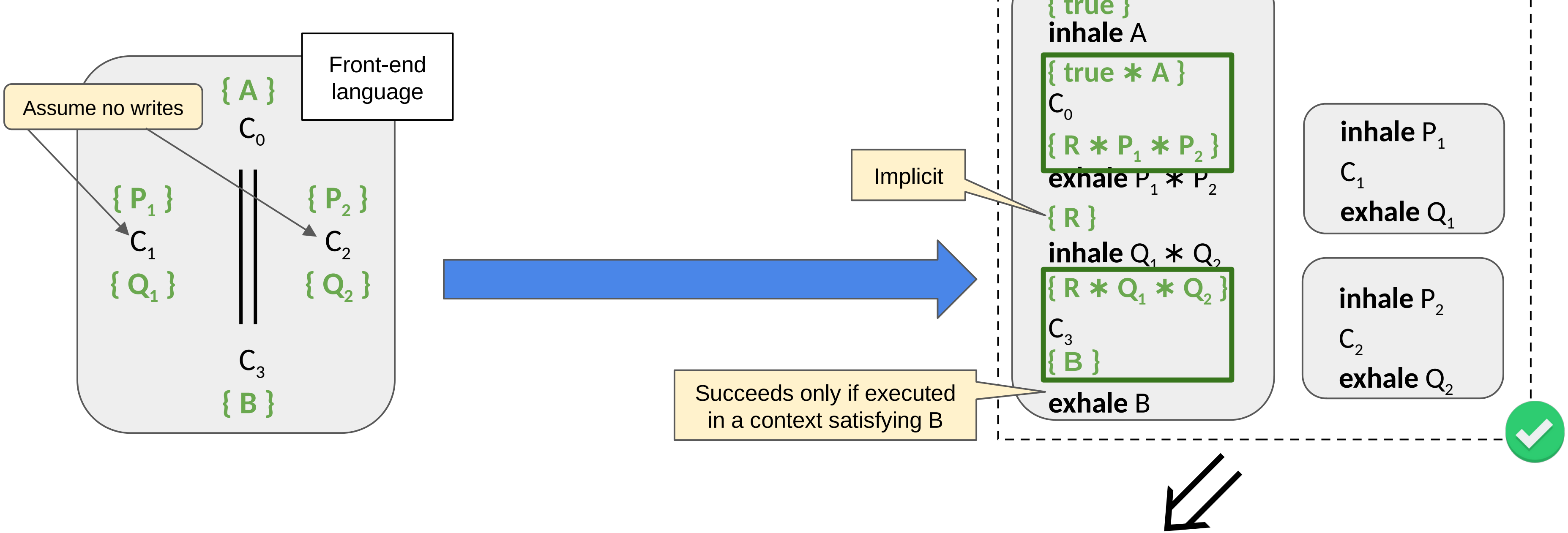


SL proof obligations proven by Viper

$\vdash \{ P_1 \}\ C_1\ \{ Q_1 \}$

$\vdash \{ P_2 \}\ C_2\ \{ Q_2 \}$

Adequacy Theorem: Viper-to-SL

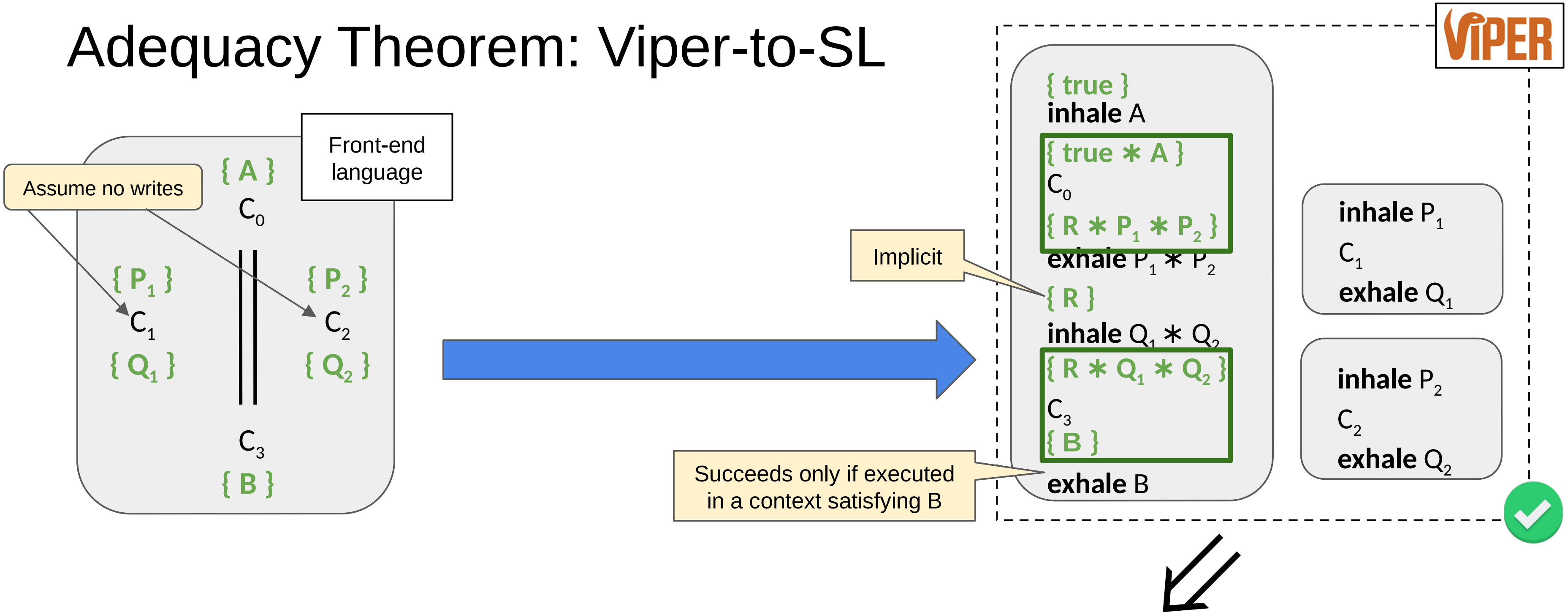


SL proof obligations proven by Viper

$\vdash \{ P_1 \}\ C_1\ \{ Q_1 \}$

$\vdash \{ P_2 \}\ C_2\ \{ Q_2 \}$

Adequacy Theorem: Viper-to-SL



SL proof obligations proven by Viper

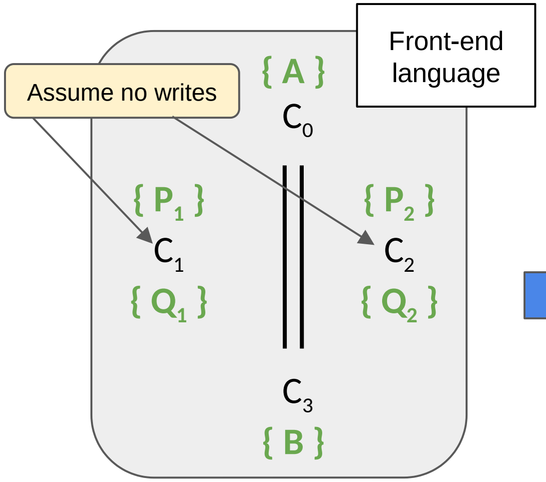
$\vdash \{ A \}\ C_0\ \{ R * P_1 * P_2 \}$

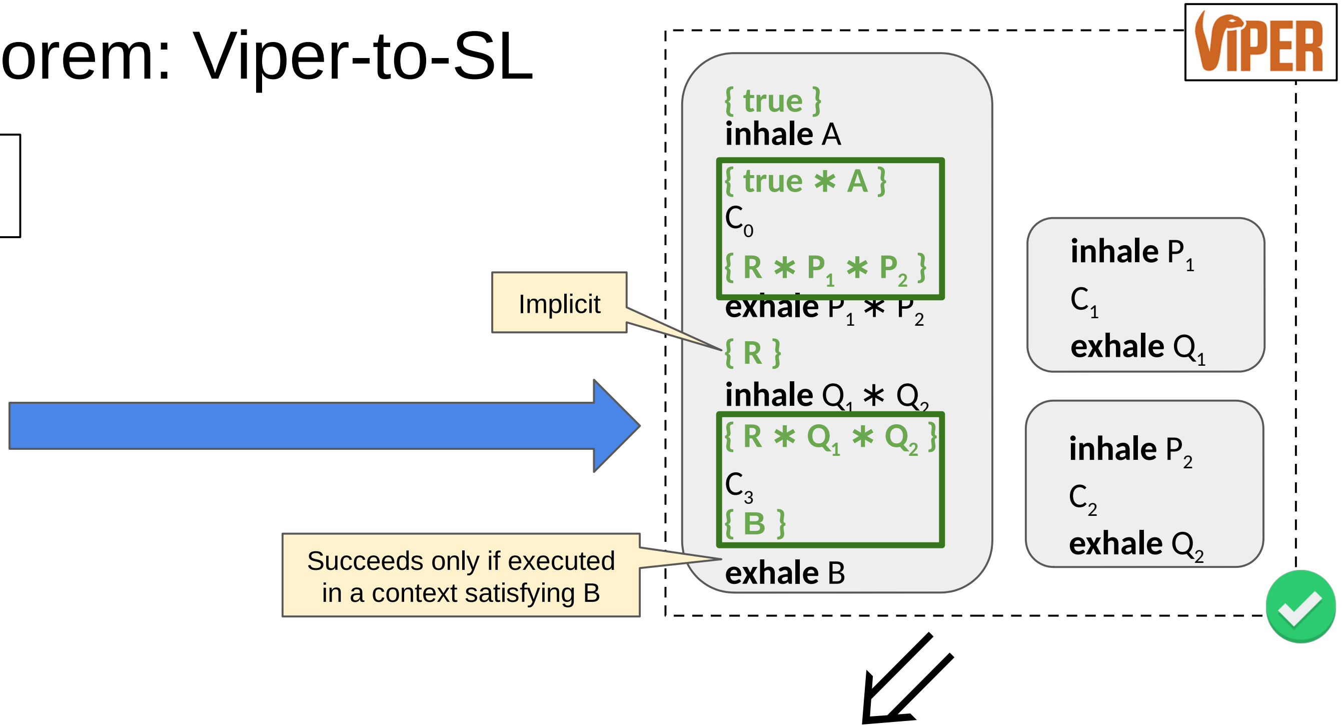
$\vdash \{ P_1 \}\ C_1\ \{ Q_1 \}$

$\vdash \{ R * Q_1 * Q_2 \}\ C_3\ \{ B \}$

$\vdash \{ P_2 \}\ C_2\ \{ Q_2 \}$

Adequacy Theorem: Viper-to-SL





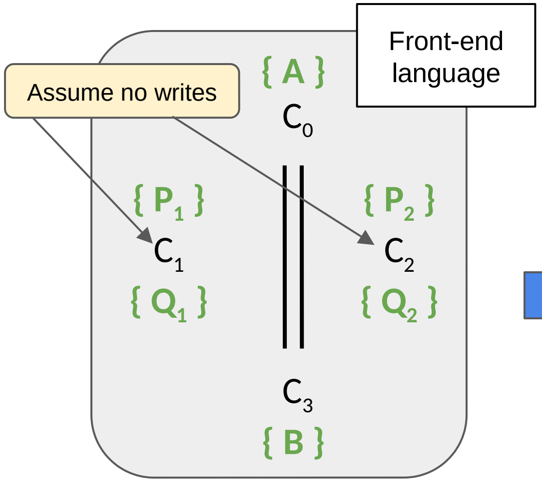
SL proof obligations proven by Viper

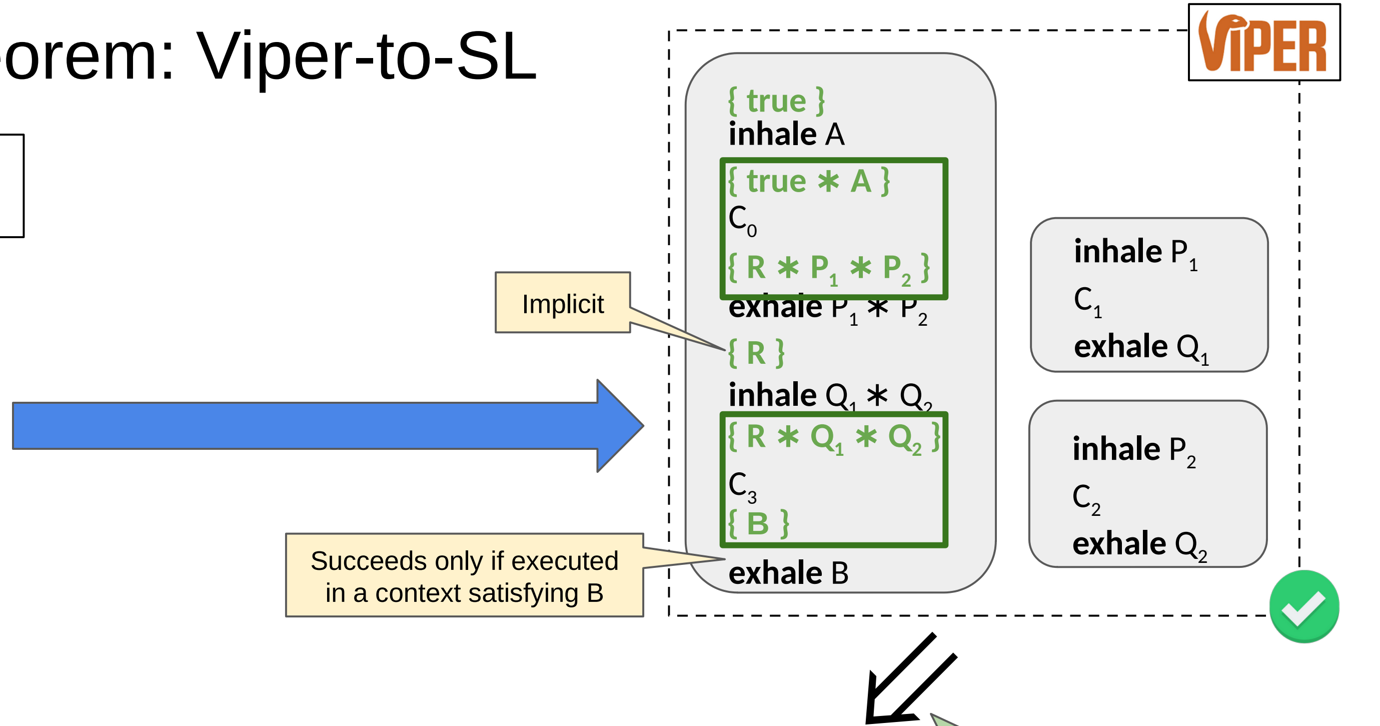
$\vdash \{ A \}\ C_0\ \{ R * P_1 * P_2 \}$ $\vdash \{ P_1 \}\ C_1\ \{ Q_1 \}$

$\vdash \{ R * Q_1 * Q_2 \}\ C_3\ \{ B \}$ $\vdash \{ P_2 \}\ C_2\ \{ Q_2 \}$

For *some* SL assertion **R**

Adequacy Theorem: Viper-to-SL





Viper-to-SL theorem
(independent from the front-end)

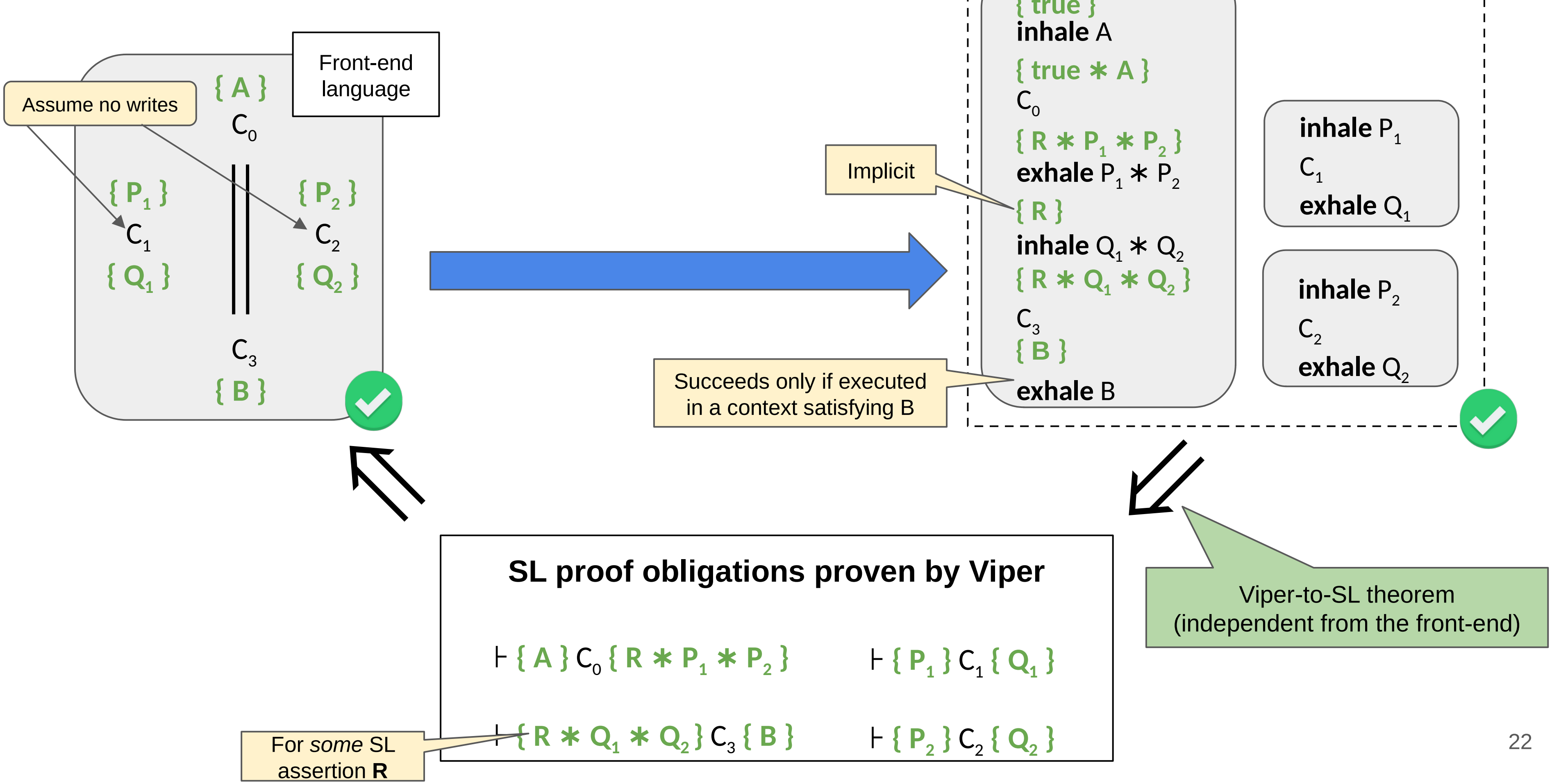
SL proof obligations proven by Viper

$\vdash \{ A \}\ C_0\ \{ R * P_1 * P_2 \}$ $\quad \vdash \{ P_1 \}\ C_1\ \{ Q_1 \}$

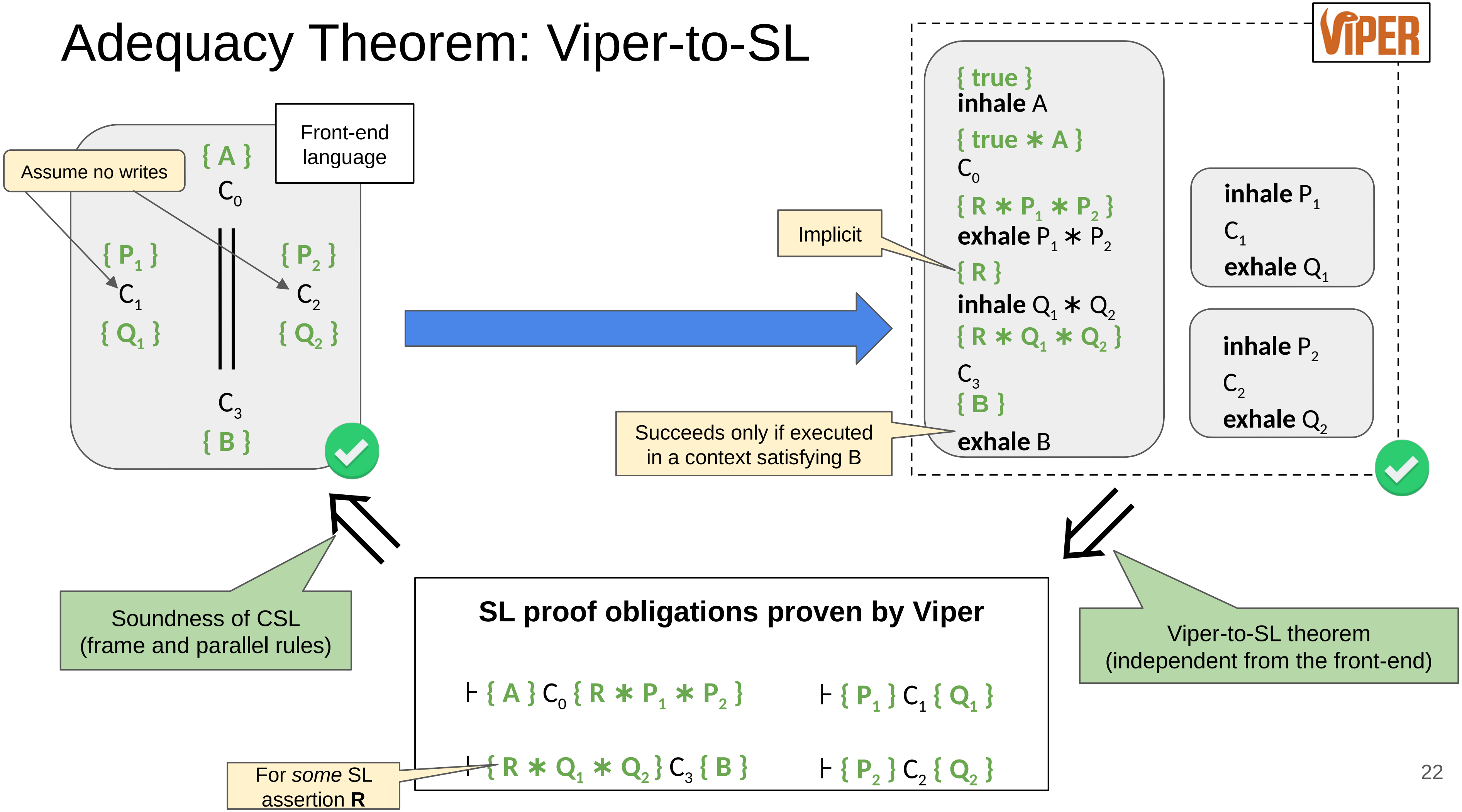
$\vdash \{ R * Q_1 * Q_2 \}\ C_3\ \{ B \}$ $\quad \vdash \{ P_2 \}\ C_2\ \{ Q_2 \}$

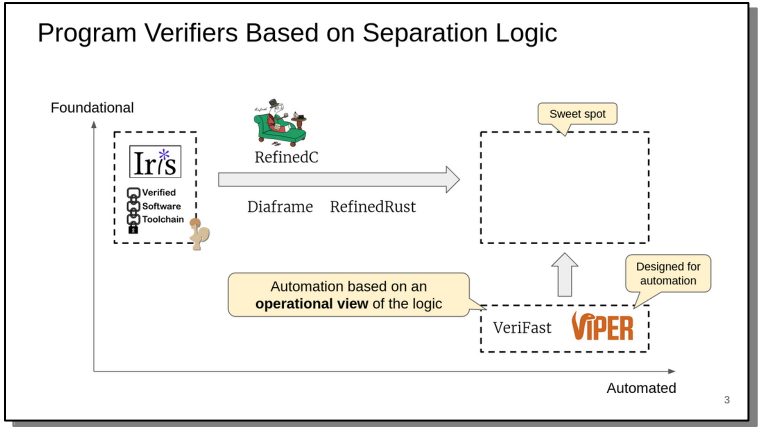
For *some* SL assertion **R**

Adequacy Theorem: Viper-to-SL

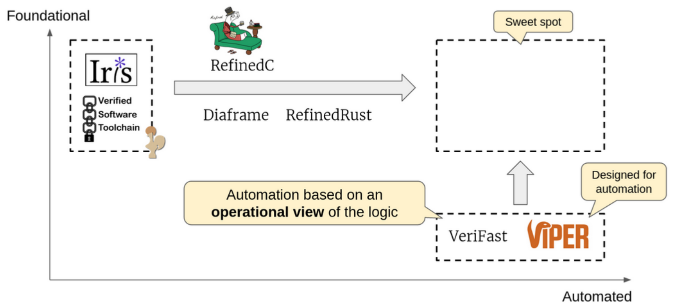


Adequacy Theorem: Viper-to-SL




Program Verifiers Based on Separation Logic
Foundational
Iris
Verified
Software
Toolchain
RefinedC
Diaframe
RefinedRust
Sweet spot
Designed for automation
Automation based on an **operational view** of the logic
VeriFast
VIPER
Automated
3

Program Verifiers Based on Separation Logic



3

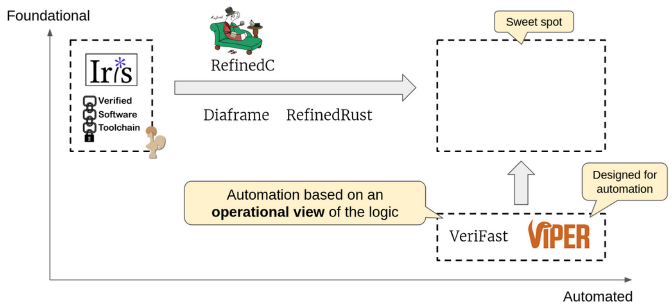
Verification Primitives: Inhale and Exhale

"A Basis for Verifying Multi-Threaded Programs" (Leino and Müller, ESOP 2009)

	inhale A	**exhale** A
Intuitive meaning	Adds resources specified by A to the current context	Removes resources specified by A from the current context
Logically	⊦ {P} **inhale** A {P ∗ A} wp (**inhale** A) {Q} = A −∗ Q	⊦ {P ∗ A} **exhale** A {P} wp (**exhale** A) {Q} = A ∗ Q
Operationally	• All resources required by A are obtained • All logical constraints are assumed	• All resources required by A are removed • All logical constraints are asserted
SL analogue of	**assume** A	**assert** A

11

Program Verifiers Based on Separation Logic



3

Verification Primitives: Inhale and Exhale

"A Basis for Verifying Multi-Threaded Programs" (Leino and Müller, ESOP 2009)

	inhale A	**exhale** A
Intuitive meaning	Adds resources specified by A to the current context	Removes resources specified by A from the current context
Logically	⊦ {P} **inhale** A {P ∗ A} wp (**inhale** A) {Q} = A −∗ Q	⊦ {P ∗ A} **exhale** A {P} wp (**exhale** A) {Q} = A ∗ Q
Operationally	• All resources required by A are obtained • All logical constraints are assumed	• All resources required by A are removed • All logical constraints are asserted
SL analogue of	**assume** A	**assert** A

11

Viper's Expression and Assertion Language

Design choice
- No impure existential
- No impure disjunction
- No impure implication
- No impure negation
- No impure logical conjunction

Field access; Heap-dependent functions

$e ::= e.x \mid f(e_1, \ldots, e_n) \mid$ **old**$[l](e) \mid e_1 + e_2 \mid e_1 / e_2 \mid n \mid v \mid b \mathbin{?} e_1 : e_2 \mid \ldots$

Pure expressions

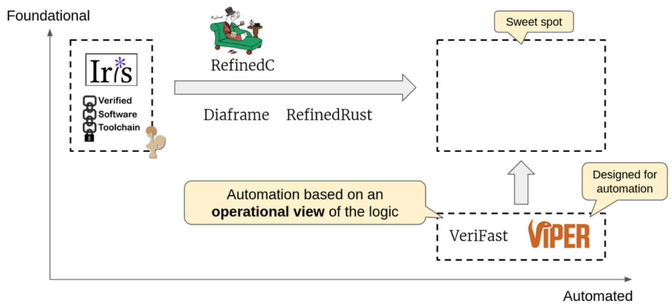
$b ::= e_1 = e_2 \mid \neg b \mid b_1 \Rightarrow b_2 \mid b_1 \lor b_2 \mid b_1 \land b_2 \mid \forall v.\, b \mid \ldots$

$A ::= b \mid$ **acc**$(e_1.x, e_2) \mid$ **acc**$(P(e_1, \ldots, e_n), e) \mid A_1 * A_2 \mid A_1 -\!\!* A_2 \mid \circledast v.\, A \mid b \Rightarrow A \mid \ldots$

Fractional permissions for heap locations; Recursive predicates with fractional permissions; Iterated separating conjunction

16

Program Verifiers Based on Separation Logic



3

Verification Primitives: Inhale and Exhale

"A Basis for Verifying Multi-Threaded Programs" (Leino and Müller, ESOP 2009)

	inhale A	**exhale** A
Intuitive meaning	Adds resources specified by A to the current context	Removes resources specified by A from the current context
Logically	$\vdash \{P\}$ **inhale** A $\{P * A\}$ wp (**inhale** A) $\{Q\} = A -\!\!* Q$	$\vdash \{P * A\}$ **exhale** A $\{P\}$ wp (**exhale** A) $\{Q\} = A * Q$
Operationally	• All resources required by A are obtained • All logical constraints are assumed	• All resources required by A are removed • All logical constraints are asserted
SL analogue of	**assume** A	**assert** A

11

Viper's Expression and Assertion Language

Design choice
- No impure existential
- No impure disjunction
- No impure implication
- No impure negation
- No impure logical conjunction

Field access — Heap-dependent functions

$e ::= e.x \mid f(e_1, \ldots, e_n) \mid \textbf{old}[l](e) \mid e_1 + e_2 \mid e_1 / e_2 \mid n \mid v \mid b\ ?\ e_1 : e_2 \mid \ldots$

Pure expressions

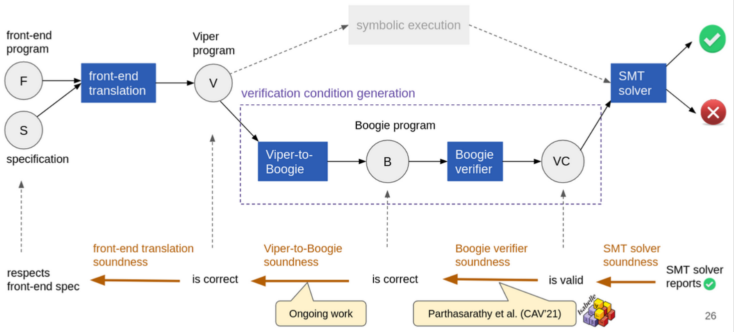
$b ::= e_1 = e_2 \mid \neg b \mid b_1 \Rightarrow b_2 \mid b_1 \lor b_2 \mid b_1 \land b_2 \mid \forall v.\ b \mid \ldots$

$A ::= b \mid \textbf{acc}(e_1.x, e_2) \mid \textbf{acc}(P(e_1, \ldots, e_n), e) \mid A_1 * A_2 \mid A_1 -\!\!* A_2 \mid \circledast v.\ A \mid b \Rightarrow A \mid \ldots$

Fractional permissions for heap locations — Recursive predicates with fractional permissions — Iterated separating conjunction

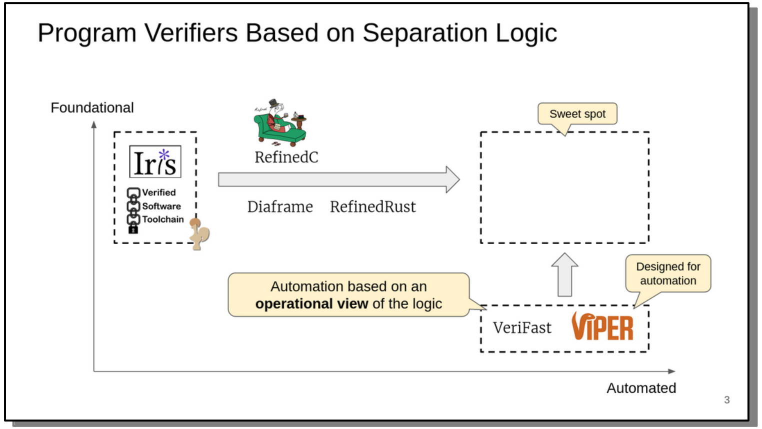
16

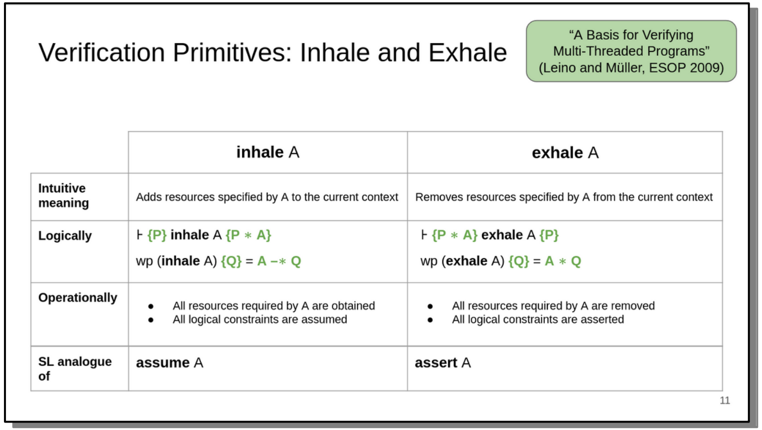
Soundness: Proof Strategy



26

Thank you for your attention!





	inhale A	**exhale** A
Intuitive meaning	Adds resources specified by A to the current context	Removes resources specified by A from the current context
Logically	⊦ {P} **inhale** A {P ∗ A} wp (**inhale** A) {Q} = A −∗ Q	⊦ {P ∗ A} **exhale** A {P} wp (**exhale** A) {Q} = A ∗ Q
Operationally	• All resources required by A are obtained • All logical constraints are assumed	• All resources required by A are removed • All logical constraints are asserted
SL analogue of	**assume** A	**assert** A