Raven

An Intermediate Verification Language (IVL) and Deductive Verifier for Concurrent Separation Logic

Ekanshdeep Gupta, Nisarg Patel, Thomas Wies

IARCS Verification Seminar Series

Tue June 9, 2026

Real-world consequences of concurrency bugs

Therac 25 radiation therapy fatalities (1985)

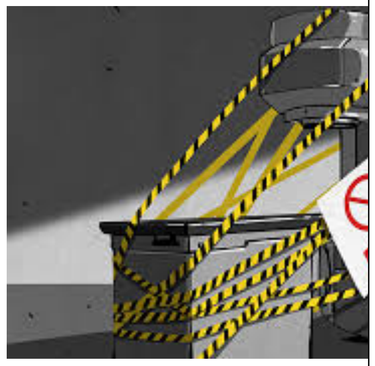
Northeastern Blackout (2003)

Toyota unintended acceleration problem (2010)

Linux security vulnerability (2016-2018)

Cloud service outage (2025)

Real-world consequences of concurrency bugs



Therac 25 ra
therapy fat
(1985)

Subject: Notice: T&E (Concur) Reimbursement Double Payment
From: gc2626@nyu.edu
Date: 3/5/24, 21:53
To: tw47@nyu.edu

Dear THOMAS WIES

It has come to our attention that on Friday, March 1st and Monday March 4th there was an issue within our system that distributes payments, which resulted in your T&E (Concur) reimbursement due to be paid this week to be doubled. This issue has since been corrected.

We have worked with our bank to reverse these erroneous payments and deposit the correct amount into your bank account. **This may take up to 3 to 5 days**

We apologize for any inconvenience this issue has caused. Should you have any questions or concerns, please do not hesitate to contact us at 212-998-1111 or email us at askfinancelink@nyu.edu

Kind Regards,

Gemma

ded acceleration
n (2010)

VS

noDB

Linux se
(2025)

Even concurrency experts make mistakes…

Correction of a Memory Management Method for Lock-Free Data Structures *

Maged M. Michael Michael L. Scott

Department of Computer Science
University of Rochester

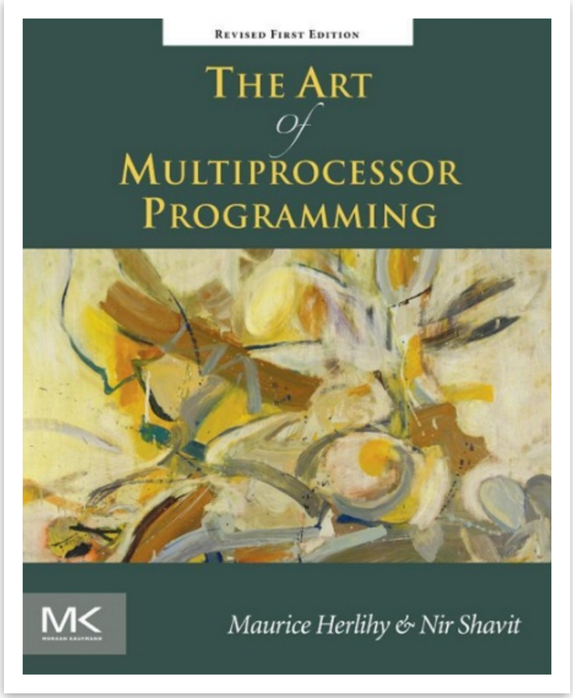
{mich

4.1 Bugs Found

We found several bugs that are not related to relaxations in the memory model. The snark algorithm has two known bugs [10, 26]. We found the first one quickly on test D0. The other one requires a fairly deep execution. We found it with the test Dq, which took about an hour.

We also found a not-previously-known bug in the lazy list-based set: the pseudocode fails to properly initialize the 'marked' field when a new node is added to the list. This simple bug went

[Burckhardt et al., PLDI '07]

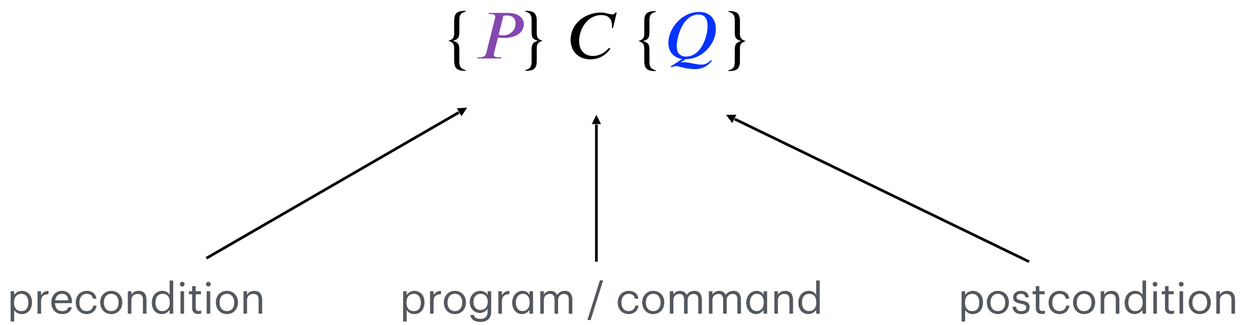


prior to this work (either automated or mechanized). In fact, our efforts identified one previously unreported bug in the original implementation of the data structure. Another bug was identified by Feldman et al. [2020], who presented an informal hindsight-based proof. While the fix proposed by

[Meyer, Wies, Wolff, PLDI '23]

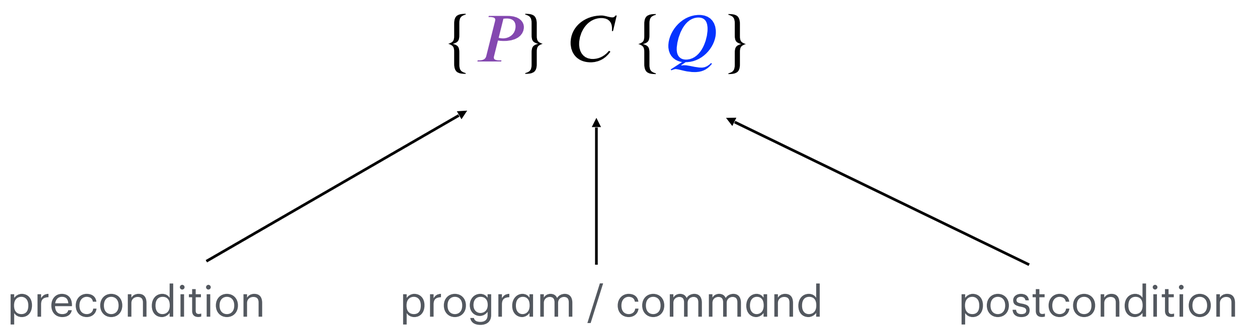
Separation Logic

[O'Hearn, Reynolds, Yang '01] [Reynolds '02]



Separation Logic

[O'Hearn, Reynolds, Yang '01] [Reynolds '02]

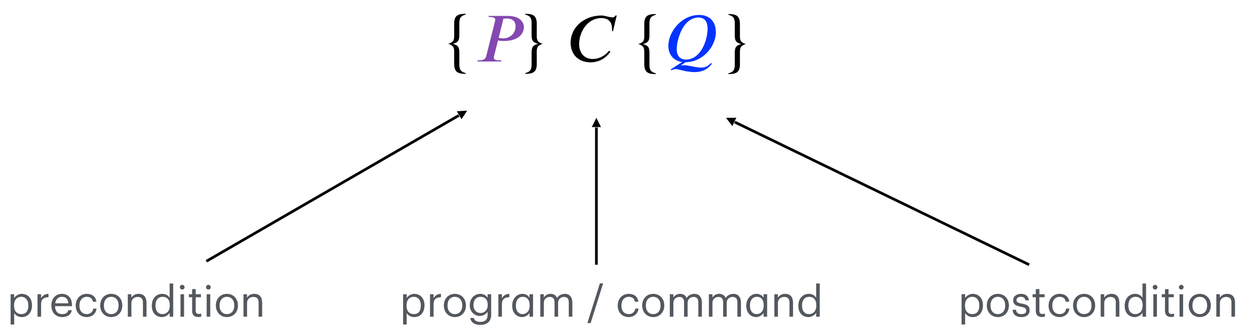


$\{P\}\ C\ \{Q\}$ being valid means:

- C executes without failure from any state that satisfies P.
- Moreover, if C terminates, then the final state satisfies Q.

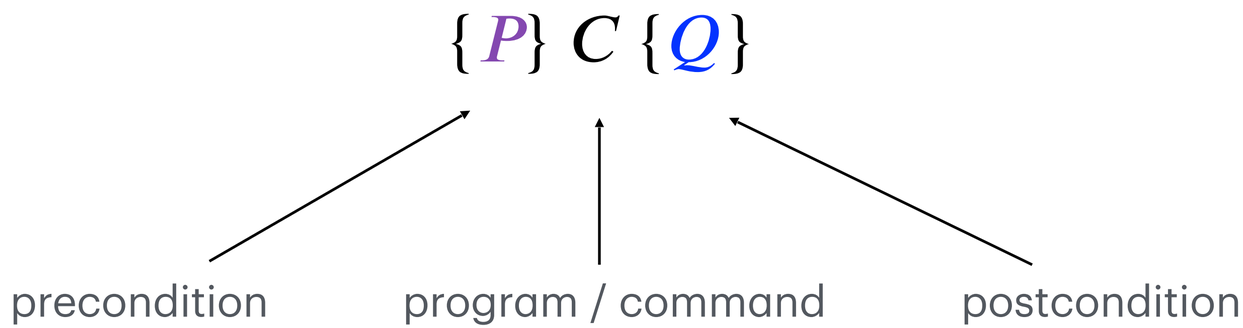
Separation Logic

[O'Hearn, Reynolds, Yang '01] [Reynolds '02]



Separation Logic

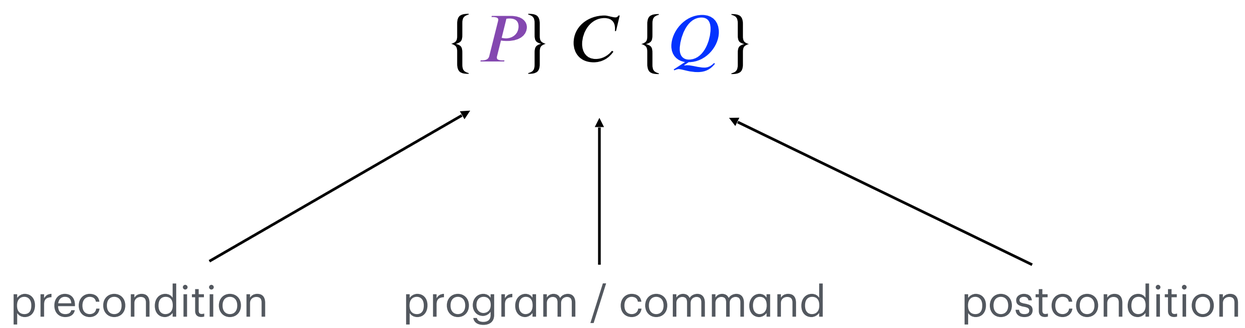
[O'Hearn, Reynolds, Yang '01] [Reynolds '02]



Things we need to understand:

Separation Logic

[O'Hearn, Reynolds, Yang '01] [Reynolds '02]

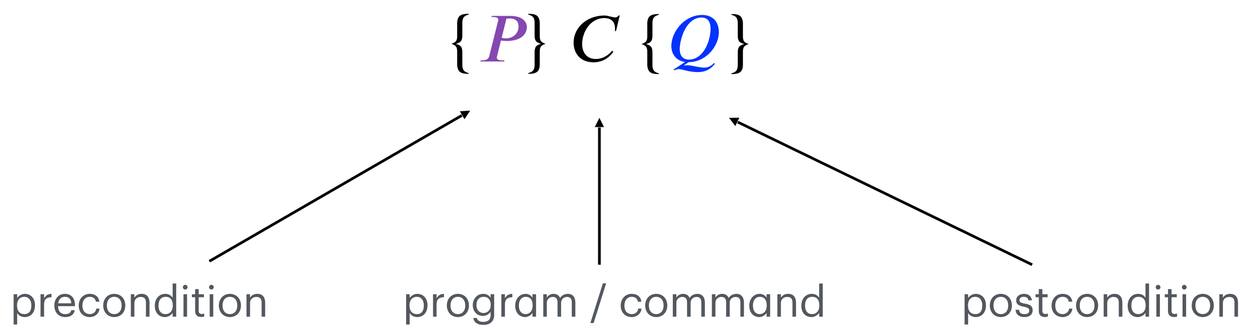


Things we need to understand:

- Assertions: P, Q

Separation Logic

[O'Hearn, Reynolds, Yang '01] [Reynolds '02]

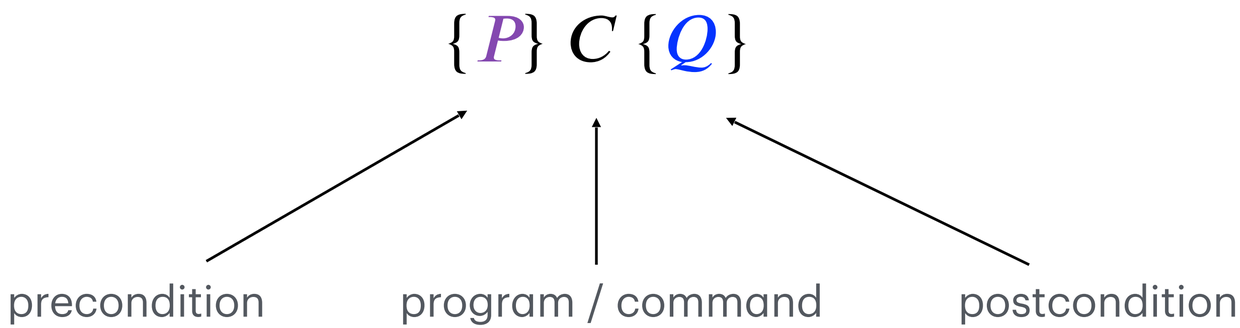


Things we need to understand:

- Assertions: P, Q
- Programs: C

Separation Logic

[O'Hearn, Reynolds, Yang '01] [Reynolds '02]



Things we need to understand:

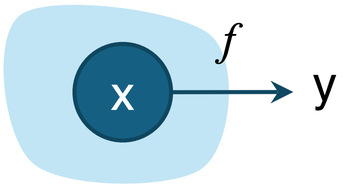
- Assertions: P, Q
- Programs: C

- Proof rules $\dfrac{\{P\}\ C_1\ \{R\} \qquad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1; C_2\ \{Q\}}$

Separation Logic Assertions

$$x \,.\, f \mapsto y$$

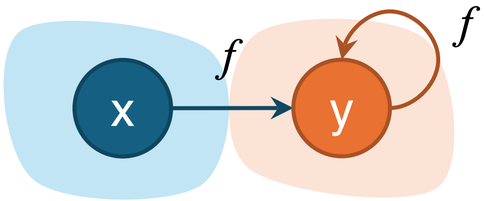
"Points-to" predicate



Separation Logic Assertions

$$x \,.\, f \mapsto y \;*\; y \,.\, f \mapsto y$$

Separating conjunction



Separation Logic Assertions

$$x\,.\,f \mapsto y \;*\; x\,.\,f \mapsto z$$

?

Separation Logic Assertions

$$x \,.\, f \mapsto y \;*\; x \,.\, f \mapsto z$$

unsatisfiable

Sub-heaps must be disjoint

($x \,.\, f$ can't be in two different places at once)

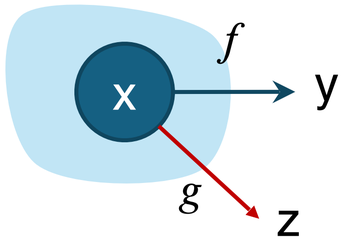
Separation Logic Assertions

$$x\,.\,f \mapsto y \;*\; x\,.\,g \mapsto z$$

?

Separation Logic Assertions

$$x \,.\, f \mapsto y \;*\; x \,.\, g \mapsto z$$



Separation Logic Assertion Semantics

- Domains
  - references: $\mathbf{Ref}$
  - values: $\mathbf{Val} = \mathbf{Ref} \cup \ldots$
  - fields: $\mathbf{Fld}$
  - (program) variables: $\mathbf{Var}$

Separation Logic Assertion Semantics

- Domains
  - references: Ref
  - values: $\mathrm{Val} = \mathrm{Ref} \cup \ldots$
  - fields: Fld
  - (program) variables: Var

- A state $\sigma \in \Sigma$ is a pair (h,s) of a stack s and a heap h
  - $s \in \mathrm{Var} \to \mathrm{Val}$
  - $h \in \mathrm{Ref} \times \mathrm{Fld} \rightharpoonup \mathrm{Val}$

Separation Logic Assertion Semantics

- Domains
  - references: Ref
  - values: $\mathrm{Val} = \mathrm{Ref} \cup \ldots$
  - fields: Fld
  - (program) variables: Var

- A state $\sigma \in \Sigma$ is a pair (h,s) of a stack s and a heap h
  - $\mathrm{s} \in \mathrm{Var} \to \mathrm{Val}$
  - $\mathrm{h} \in \mathrm{Ref} \times \mathrm{Fld} \rightharpoonup \mathrm{Val}$

- Composition of states

$$(h_1,\ s_1) * (h_2,\ s_2) = \begin{cases} (h_1 \uplus h_2, s_1) & \text{if } s_1 = s_2 \text{ and } h_1 \perp h_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Here, $h_1 \perp h_2$ means domains of h_1 and h_2 are disjoint

Separation Logic Assertion Semantics

- $(s, h) \vDash x \,.\, f \mapsto y \quad \Leftrightarrow \quad (s(x), f) \in dom(h) \text{ and } h(s(x), f) = s(y)$
- $\sigma \vDash P * Q \quad \Leftrightarrow \quad \exists \sigma_1, \sigma_2 \,.\; \sigma = \sigma_1 * \sigma_2 \text{ and } \sigma_1 \vDash P \text{ and } \sigma_2 \vDash Q$
- ... rest as in classical (first-order) logic

Programs: Syntax

- Basic commands c:
  - no-op: **`skip`**
  - guard: **`assume`**`(b)`
  - heap write: `x.f := y`
  - heap read: `x := y.f`
  - allocation: `x :=` **`new`**`(f:y,…)`
  - …

Programs: Syntax

- Basic commands c:
  - no-op: **`skip`**
  - guard: **`assume(b)`**
  - heap write: **`x.f := y`**
  - heap read: **`x := y.f`**
  - allocation: **`x := new(f:y,…)`**
  - …

- Commands $C \in Com$:
  - basic commands: c
  - sequencing: $C_1;C_2$
  - nondet. choice: C_1+C_2
  - looping: $C*$
  - …

Programs: Operational Semantics

Programs: Operational Semantics

- Reduction relation

$$\rightarrow \quad \subseteq (\mathrm{Com} \times \Sigma) \times \left(\mathrm{Com} \times \Sigma \cup \{\mathrm{abort}\}\right)$$

  - Notation: $(C, \sigma) \rightarrow (C', \sigma')$
  - Meaning: C takes a step in state σ, yielding continuation C' and new state σ'

Programs: Operational Semantics

- Reduction relation

  $\rightarrow \quad \subseteq (\mathrm{Com} \times \Sigma) \times \left(\mathrm{Com} \times \Sigma \cup \{\mathrm{abort}\}\right)$

  - Notation: $(C, \sigma) \rightarrow (C', \sigma')$
  - Meaning: C takes a step in state σ, yielding continuation C' and new state σ'

- Example: semantics of heap writes
  - $(\mathrm{x.f{:=}y}, \sigma) \rightarrow (\mathrm{skip}, \sigma')$ if $\sigma = \left(h, s\right)$ and $\left(s(\mathrm{x}), \mathrm{f}\right) \in dom\left(h\right)$
    and $\sigma' = (h\left[(\mathrm{x}, \mathrm{f}) \mapsto s(\mathrm{y})\right], s)$
  - $(\mathrm{x.f{:=}y}, \sigma) \rightarrow$ abort if $\sigma = \left(h, s\right)$ and $\left(s(\mathrm{x}), \mathrm{f}\right) \notin dom\left(h\right)$

Hoare Triples Formally

Hoare Triples Formally

"$\{P\}\ C\ \{Q\}$ valid" means

Hoare Triples Formally

"$\{P\}\ C\ \{Q\}$ valid" means

- C executes without failure from any state that satisfies P.

Hoare Triples Formally

"$\{P\}\ C\ \{Q\}$ valid" means

- C executes without failure from any state that satisfies P.
- Moreover, if C terminates, then the final state satisfies Q.

Hoare Triples Formally

"$\{P\}\ C\ \{Q\}$ valid" means

- C executes without failure from any state that satisfies P.
- Moreover, if C terminates, then the final state satisfies Q.

Formally: for all $\sigma,\ \sigma'$

Hoare Triples Formally

"$\{P\}\ C\ \{Q\}$ valid" means

- C executes without failure from any state that satisfies P.
- Moreover, if C terminates, then the final state satisfies Q.

Formally: for all $\sigma,\ \sigma'$

- if $\sigma \vDash P$ then $(C,\ \sigma) \nrightarrow^{*}$ abort

Hoare Triples Formally

"$\{P\}\ C\ \{Q\}$ valid" means

- C executes without failure from any state that satisfies P.
- Moreover, if C terminates, then the final state satisfies Q.

Formally: for all $\sigma,\ \sigma'$

- if $\sigma \vDash P$ then $(C,\ \sigma) \nrightarrow^{*}$ abort
- if $\sigma \vDash P$ and $(C,\ \sigma) \rightarrow^{*} (\text{skip},\ \sigma')$ then $\sigma' \vDash Q$

Separation Logic Proof Rules

[Alloc] $\{\text{true}\}\ \text{x} := \text{new(f:y)}\ \{\text{x.f} \mapsto\ \text{y}\}$

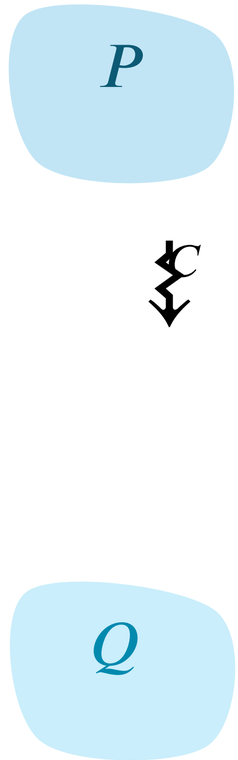
[Write] $\{\text{x.f} \mapsto\ \text{v}\}\ \text{x.f} := \text{y}\ \{\text{x.f} \mapsto\ \text{y}\}$

[Read] $\{\text{x.f} \mapsto\ \text{v}\}\ \text{y} := \text{x.f}\ \{\text{x.f} \mapsto\ \text{v}\ *\ \text{y} = \text{v}\}$

[Seq] $$\frac{\{P\}\ C_1\ \{R\} \qquad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1; C_2\ \{Q\}}$$

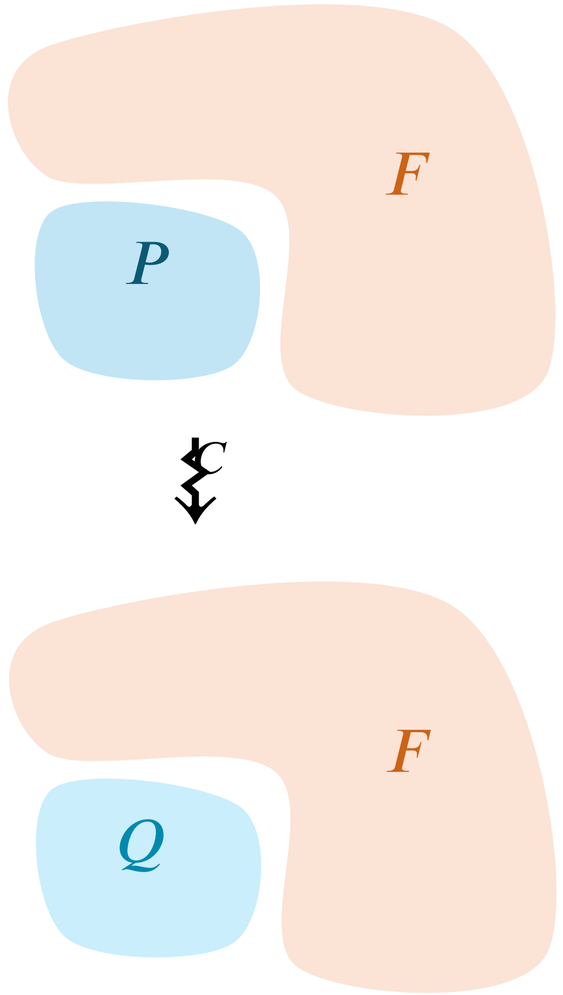
Frame Rule

$$\{P\} \;\; C \;\; \{Q\}$$



Frame Rule

$$\text{[Frame]}\quad \frac{\{P\}\ \ C\ \ \{Q\}}{\{P * F\}\ \ C\ \ \{Q * F\}}$$



Example Proof

$\{\text{true}\}$

```
x := new (f: 0);
y := new (f: 0);
x.f := 1
```

$\{y.f \mapsto 0 \ * \ x.f \mapsto 1\}$

Example Proof

$$[\text{Seq}]\ \frac{\{P\}\ C_1\ \{R\} \qquad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1; C_2\ \{Q\}}$$

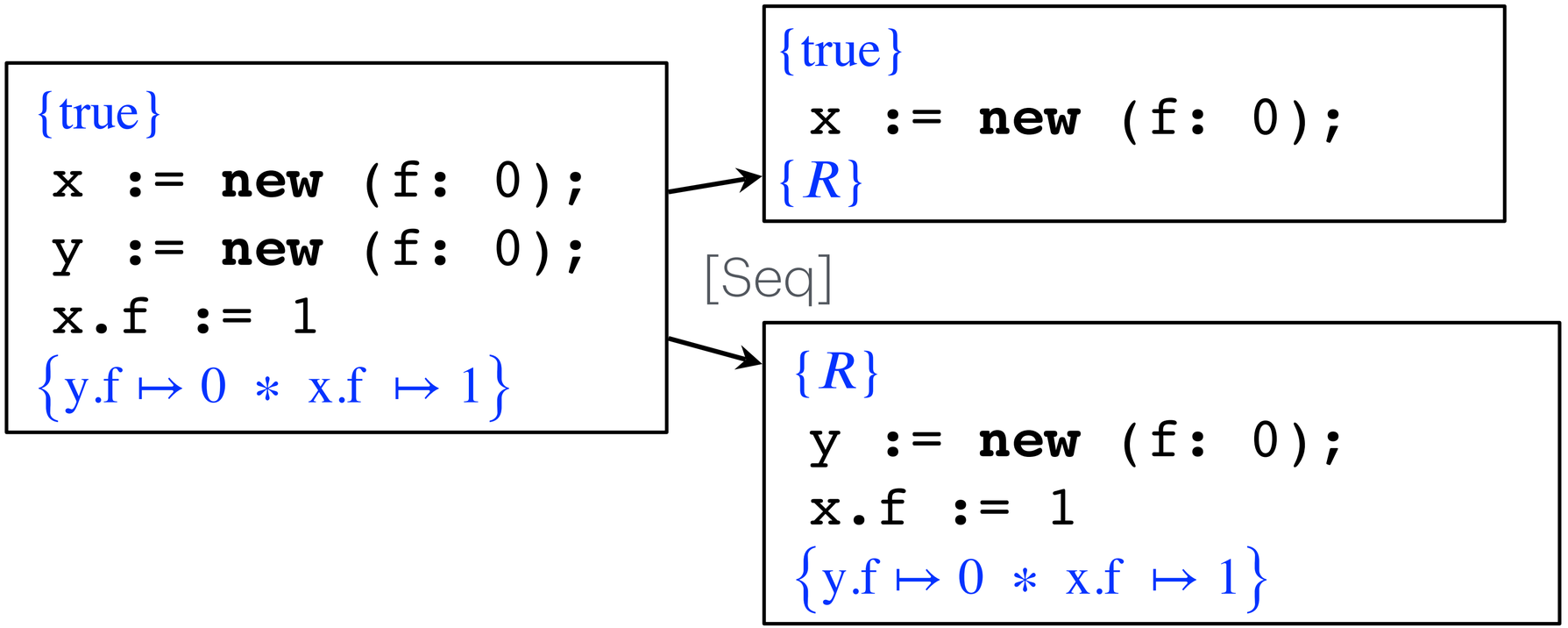
{true}

```
x := new (f: 0);
y := new (f: 0);
x.f := 1
```

$\{y.f \mapsto 0 \ * \ x.f \mapsto 1\}$

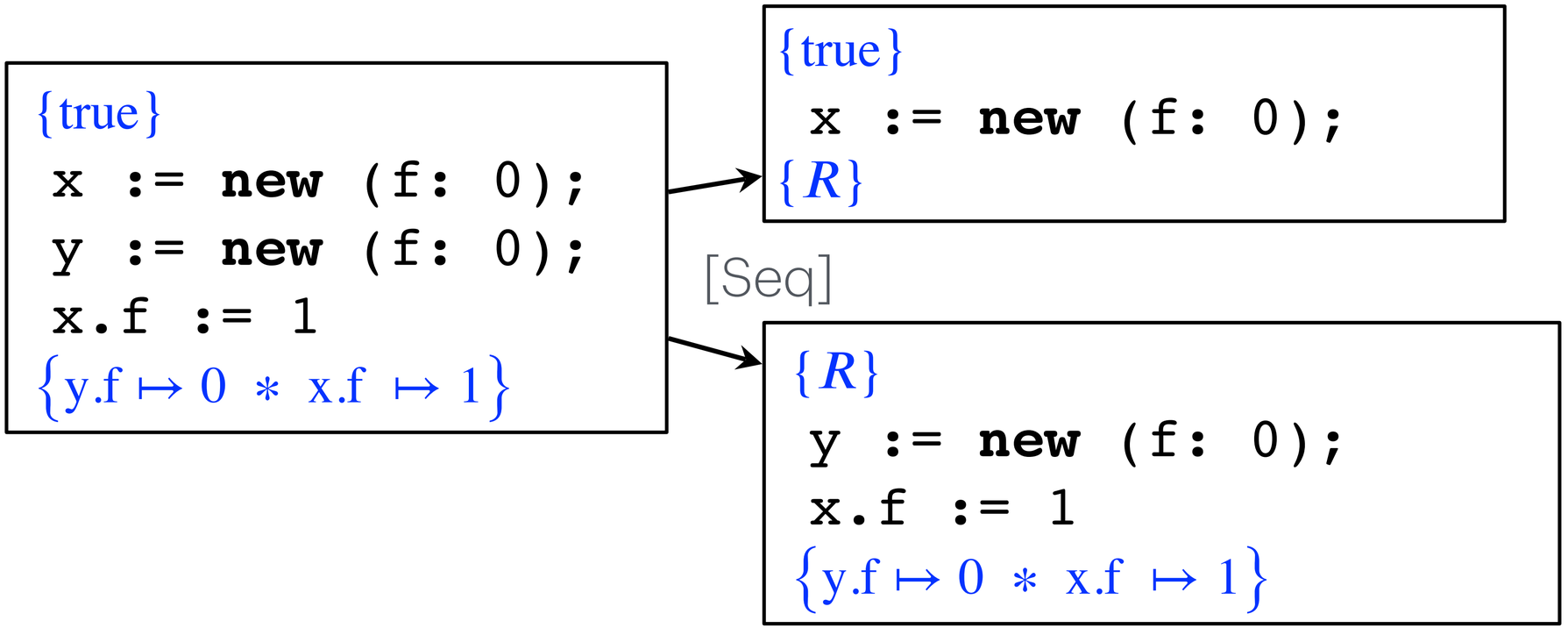
Example Proof

$$[\text{Seq}]\ \frac{\{P\}\ C_1\ \{R\} \qquad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1; C_2\ \{Q\}}$$



Example Proof

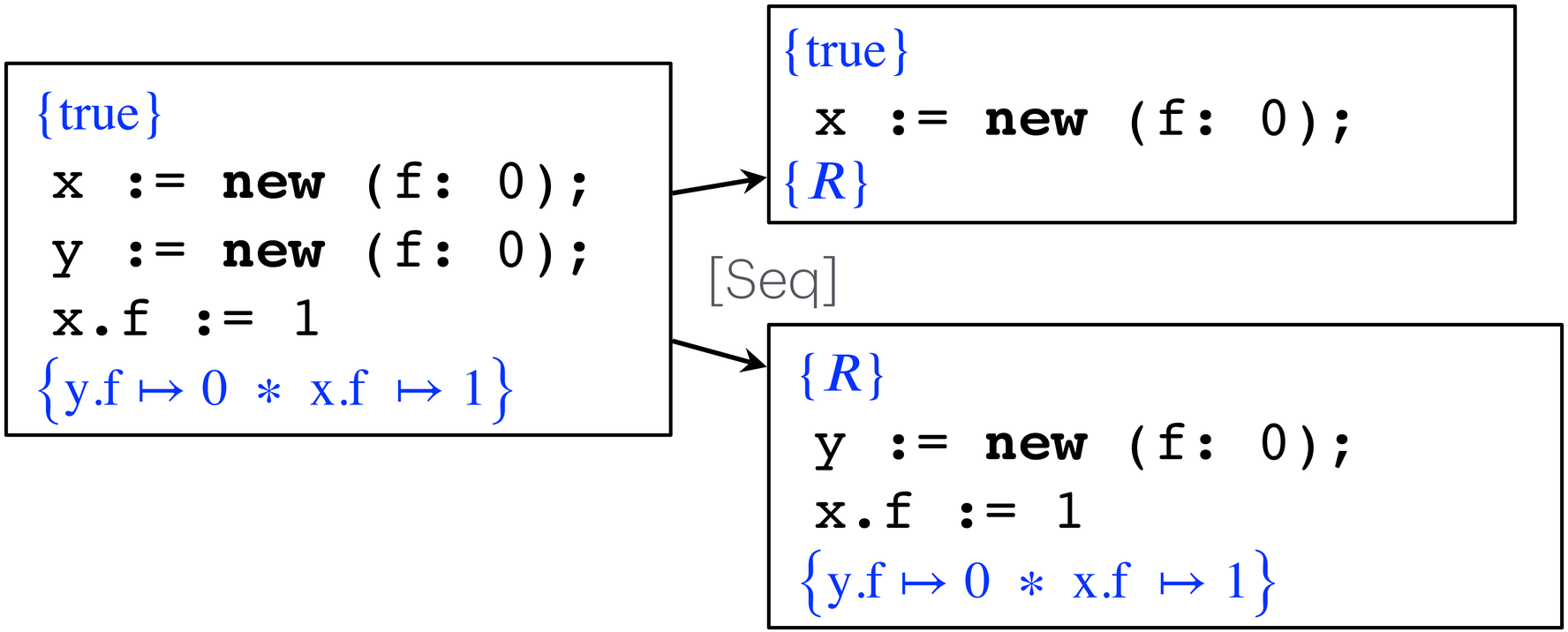
$$\text{[Seq]}\ \frac{\{P\}\ C_1\ \{R\} \qquad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1; C_2\ \{Q\}}$$



R?

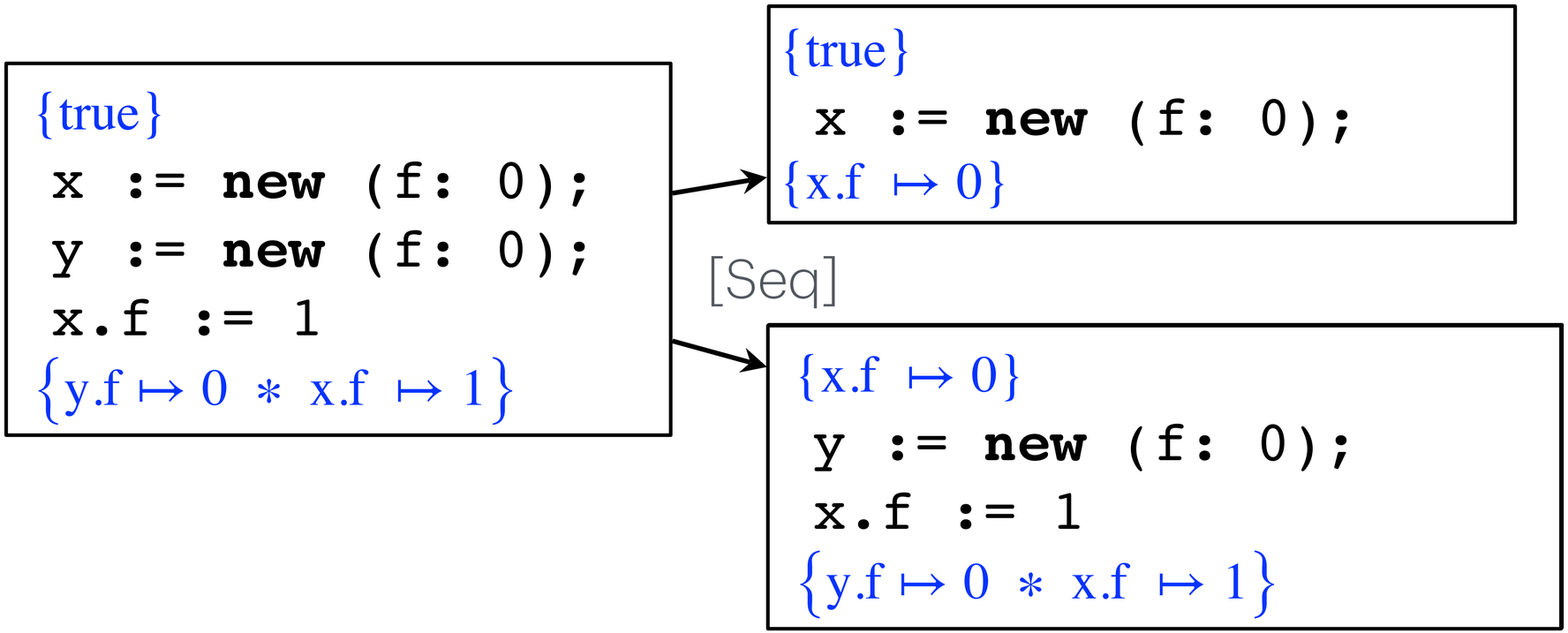
Example Proof

[Alloc] $\{true\}\ x := new(f{:}y)\ \{x.f \mapsto y\}$



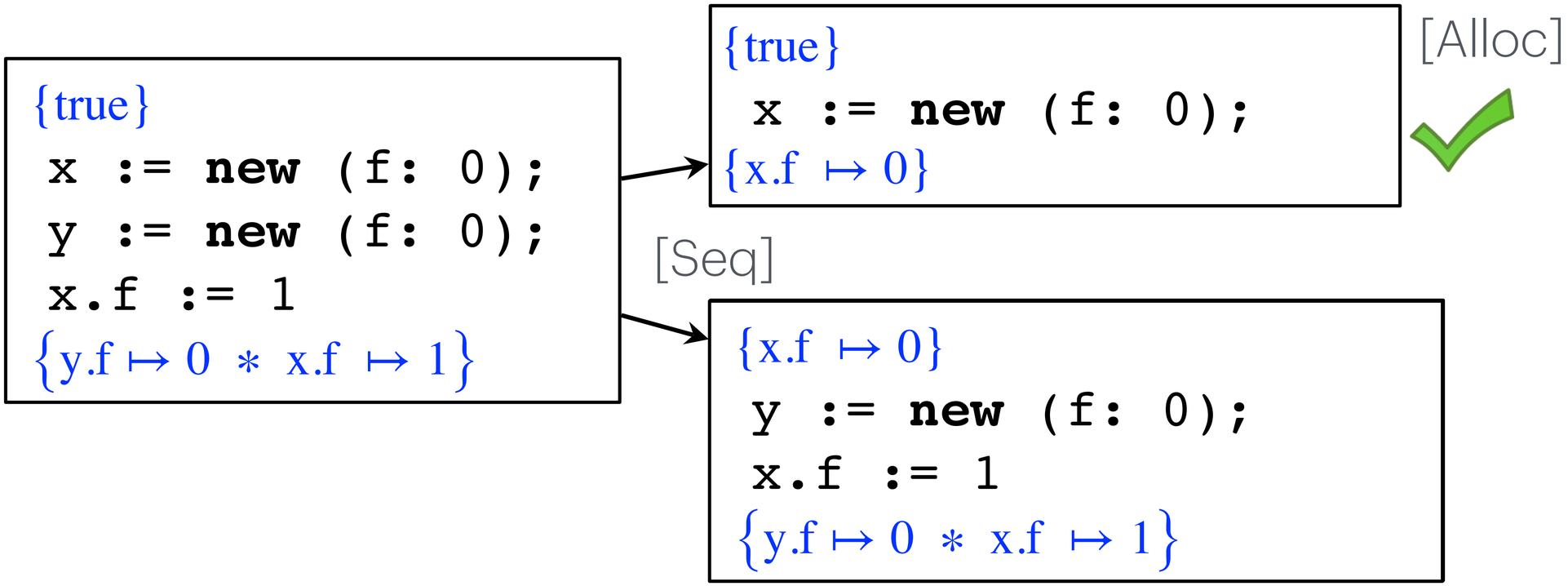
Example Proof

[Alloc] $\{true\}\ x := new(f{:}y)\ \{x.f \mapsto y\}$



Example Proof

[Alloc] $\{true\}\ x := new(f{:}y)\ \{x.f \mapsto y\}$



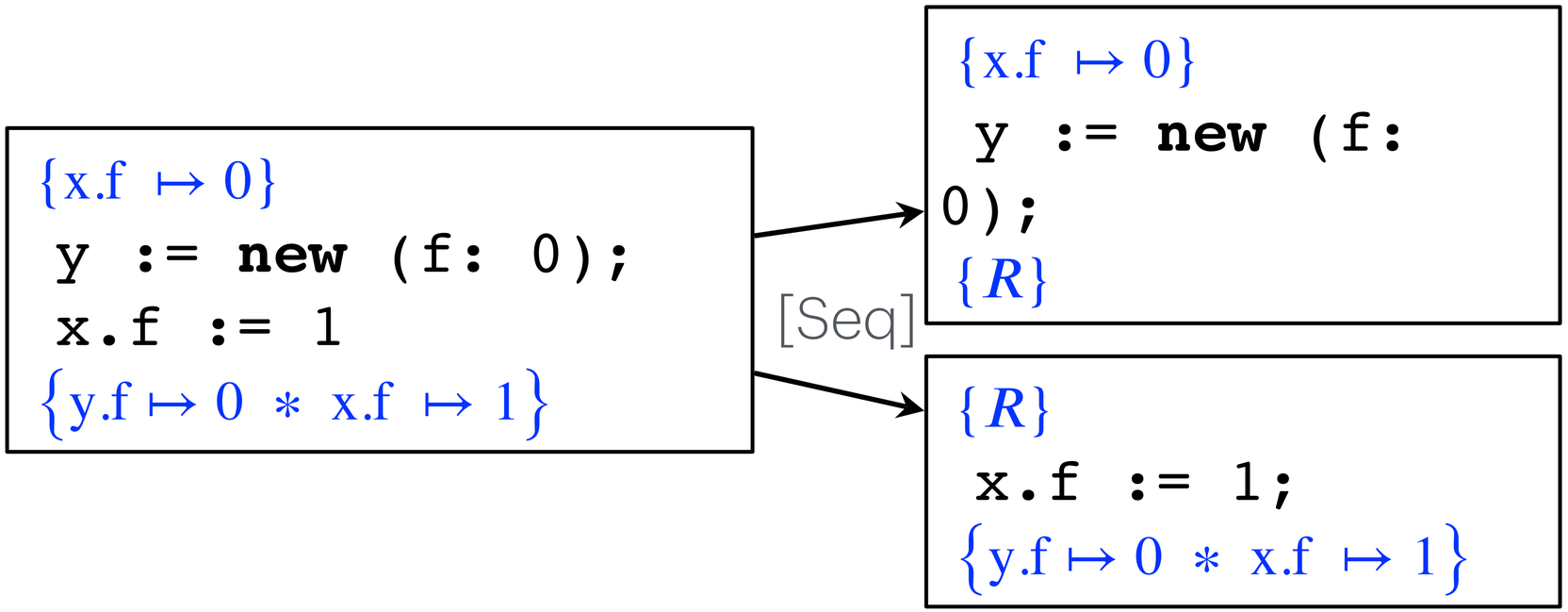
Example Proof

$\{x.f \mapsto 0\}$

```
y := new (f: 0);
x.f := 1
```

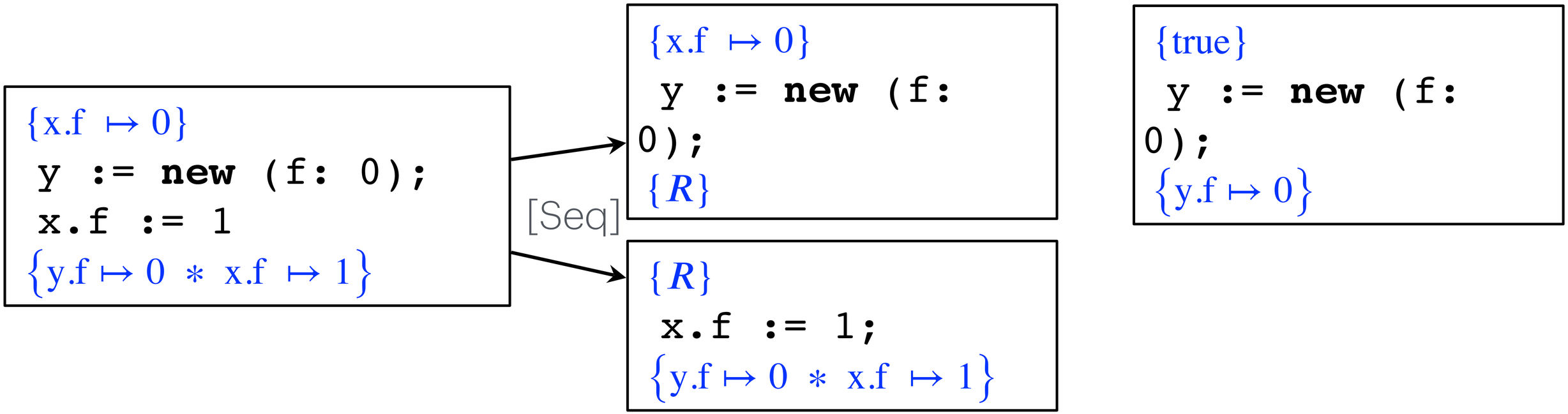
$\{y.f \mapsto 0 * x.f \mapsto 1\}$

Example Proof



Example Proof

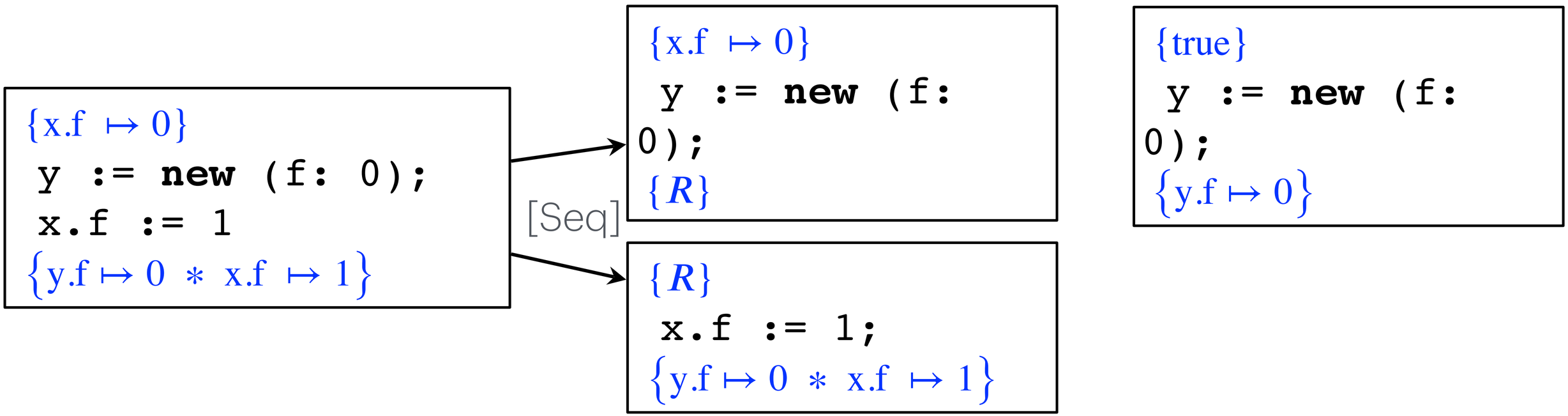
[Alloc] $\{\text{true}\}\ \text{x} := \text{new(f:y)}\ \{\text{x.f} \mapsto \text{y}\}$



Example Proof

[Alloc] $\{\text{true}\}\ \text{x} := \text{new(f:y)}\ \{\text{x.f} \mapsto \text{y}\}$

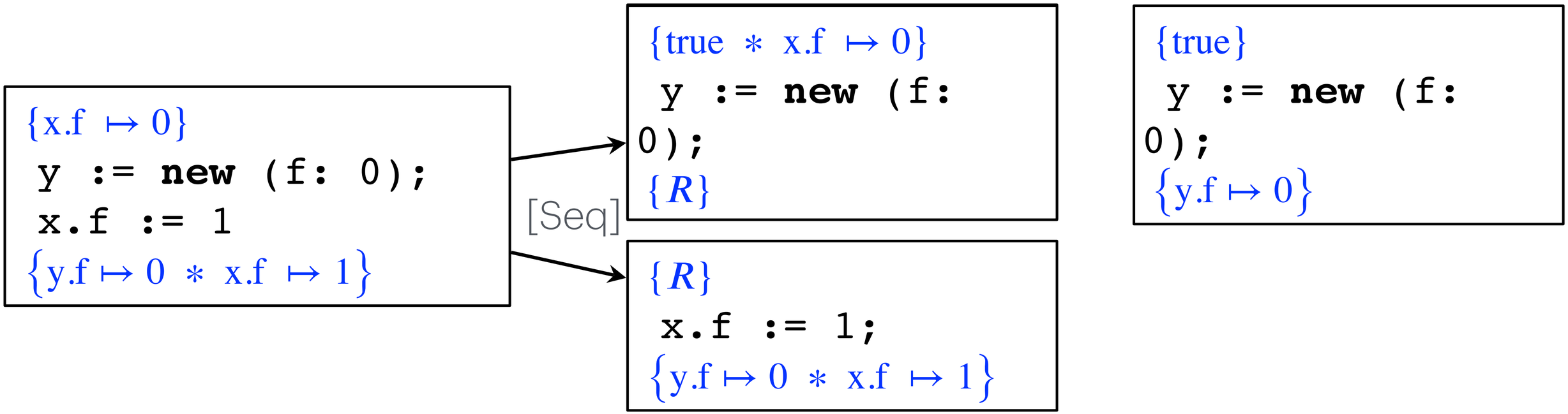
[Frame] $$\frac{\{P\}\ \ C\ \ \{Q\}}{\{P * F\}\ \ C\ \ \{Q * F\}}$$



Example Proof

[Alloc] $\{\text{true}\}\ \text{x} := \text{new(f:y)}\ \{\text{x.f} \mapsto \text{y}\}$

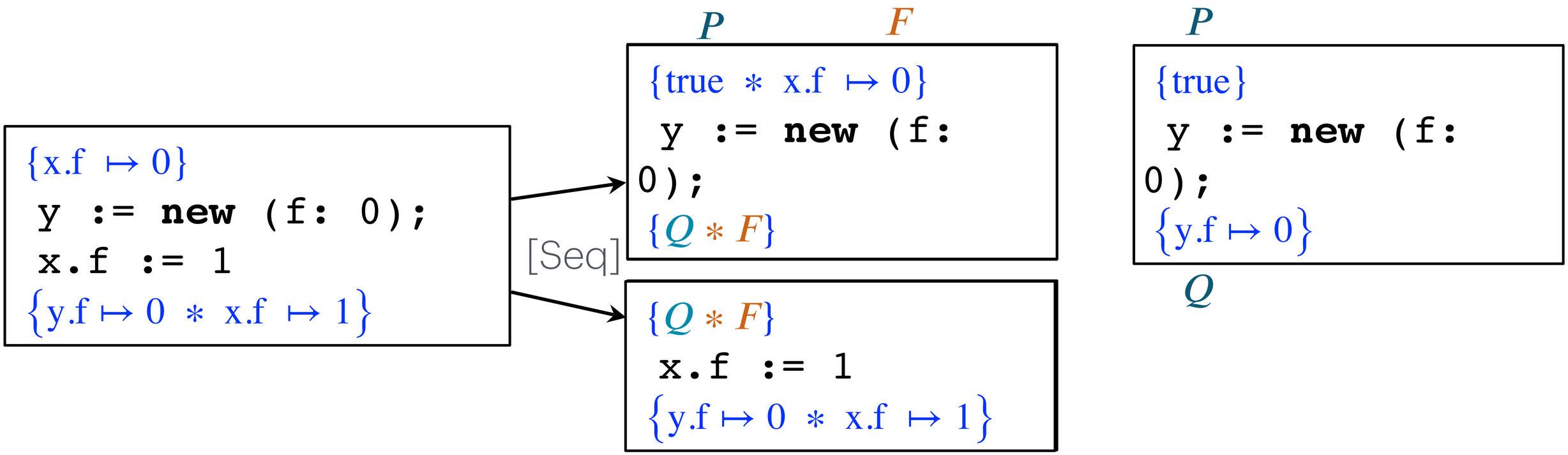
[Frame] $$\frac{\{P\}\ \ C\ \ \{Q\}}{\{P * F\}\ \ C\ \ \{Q * F\}}$$



Example Proof

[Alloc] $\{\text{true}\}\ \text{x} := \text{new(f:y)}\ \{\text{x.f} \mapsto \text{y}\}$

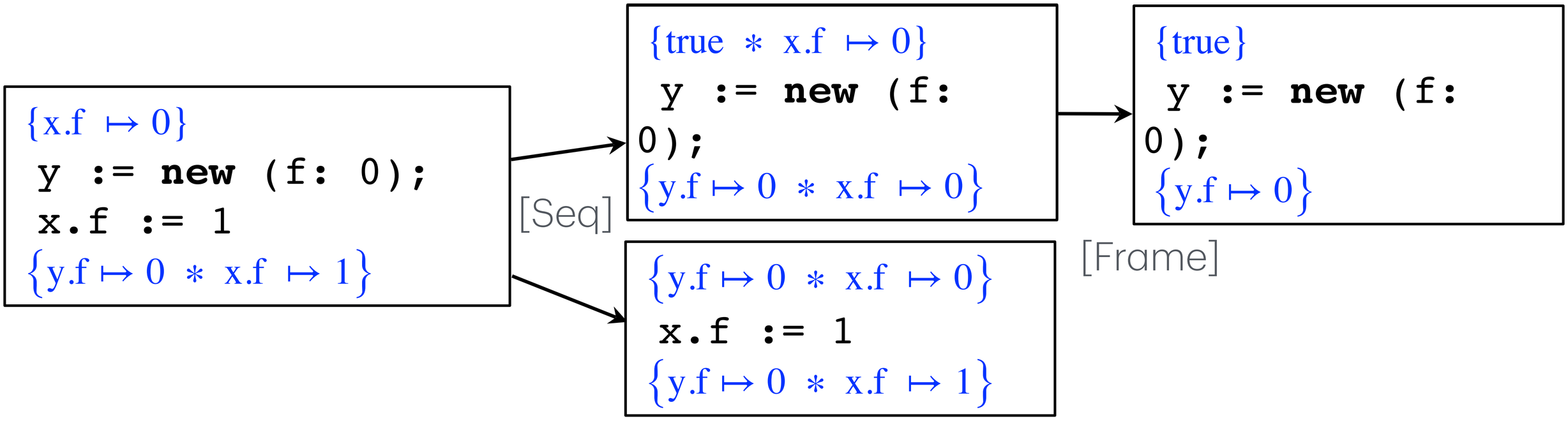
[Frame] $$\frac{\{P\}\ \ C\ \ \{Q\}}{\{P * F\}\ \ C\ \ \{Q * F\}}$$



Example Proof

[Alloc] $\{\text{true}\}\ \text{x} := \text{new(f:y)}\ \{\text{x.f} \mapsto \text{y}\}$

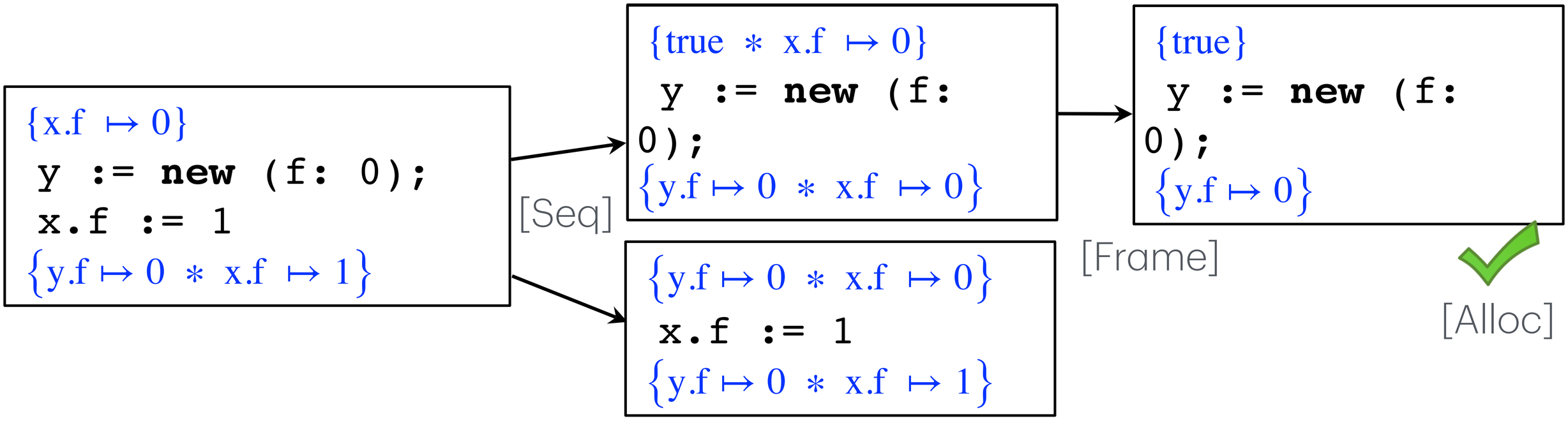
[Frame] $$\frac{\{P\}\ C\ \{Q\}}{\{P * F\}\ C\ \{Q * F\}}$$



Example Proof

[Alloc] $\{\text{true}\}\ \text{x} := \text{new(f:y)}\ \{\text{x.f} \mapsto \text{y}\}$

[Frame] $$\frac{\{P\}\ C\ \{Q\}}{\{P * F\}\ C\ \{Q * F\}}$$



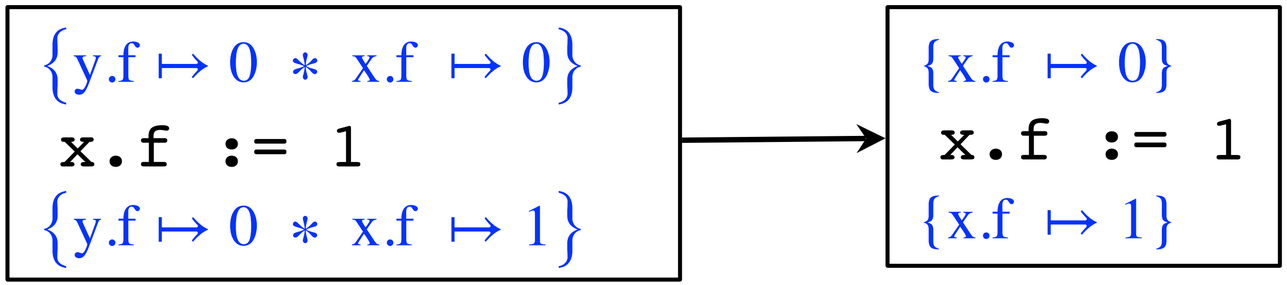
Example Proof

$\{y.f \mapsto 0 \ * \ x.f \ \mapsto 0\}$

```
x.f := 1
```

$\{y.f \mapsto 0 \ * \ x.f \ \mapsto 1\}$

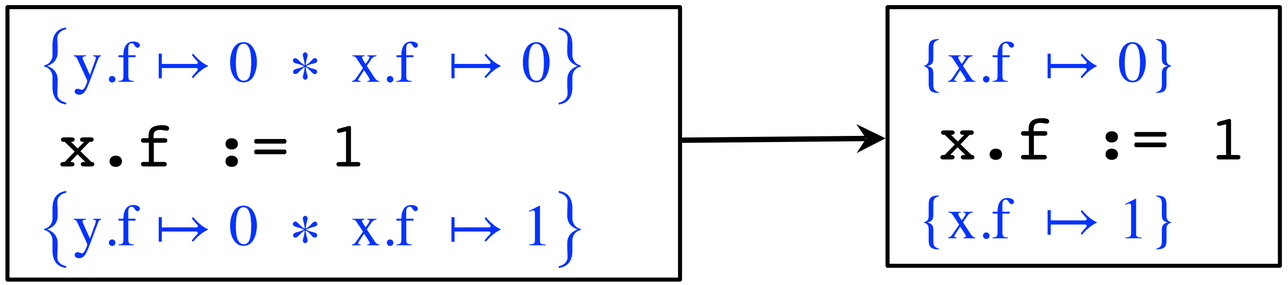
Example Proof



[Frame]

Example Proof

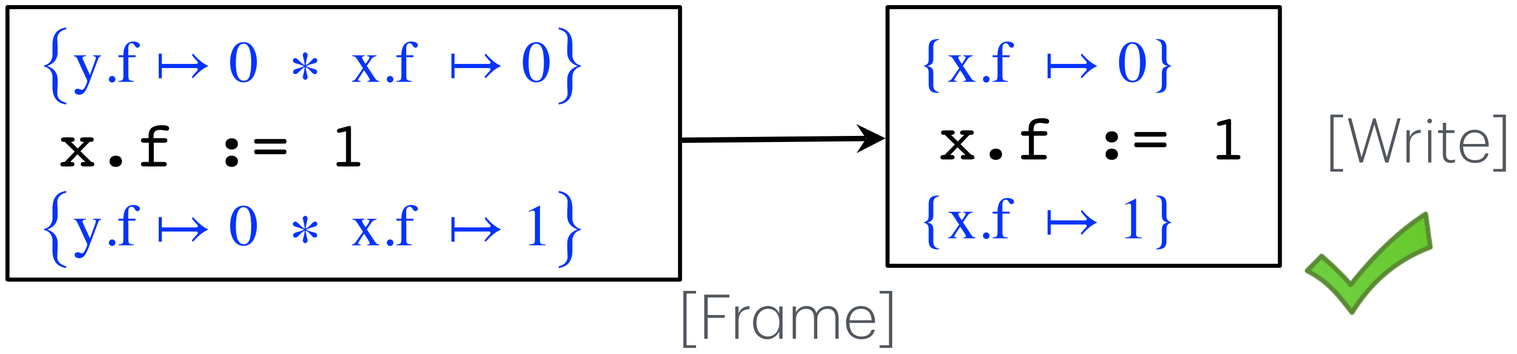
[Write] $\{x.f \mapsto v\}\ x.f := y\ \{x.f \mapsto y\}$



[Frame]

Example Proof

[Write] $\{x.f \mapsto v\}\ x.f := y\ \{x.f \mapsto y\}$



Sequential Stack

```
proc push(s: Ref, x: Int)
{
  var hd: Ref := s.head;

  var hd0: Ref;
  hd0 := new(value: x, next: hd);

  s.head := hd0;
}
```

Fig 1a: push() in a sequential stack

```
proc pop(s: Ref)
  returns (result: IntOption)
{
  var hd: Ref := s.head;

  if (hd == null) {
    return none();
  }

  var hd_next: Ref := hd.next;
  var hd_value: Int := hd.value;
  s.head := hd_next;
}
```

Fig 1b: pop() in a sequential stack

Treiber Stack

```
proc push(s: Ref, x: Int)
{
  var hd: Ref := s.head;

  var hd0: Ref;
  hd0 := new(value: x, next: hd);
  var res: Bool := cas(s.head, hd, hd0);

  if (res) {
    return;
  } else {
    push(s, x);
  }
}
```

Fig 2a: push() in a Treiber Stack

```
proc pop(s: Ref)
  returns (result: IntOption)
{
  var hd: Ref := s.head;

  if (hd == null) {
    return none();
  }

  var hd_next: Ref := hd.next;
  var hd_value: Int := hd.value;
  var res : Bool := cas(s.head, hd, hd_next);

  if (res) {
    return some(hd_value);
  } else {
    var ret: IntOption := pop(s);
    return ret;
  }
}
```

Fig 2b: pop() in a Treiber Stack

Guaranteeing Correctness

- Hand-written test suites
- Randomized test generation
- Memory-safe programming languages
- *Functional correctness*
  - *Mathematical proof of correctness*#
  - Formal Verification!

```
proc pop(s: Ref)
  returns (result: IntOption)
{
  var hd: Ref := s.head;

  if (hd == null) {
    return none();
  }

  var hd_next: Ref := hd.next;
  var hd_value: Int := hd.value;
  var res : Bool := cas(s.head, hd, hd_next);

  if (res) {
    return some(hd_value);
  } else {
    var ret: IntOption := pop(s);
    return ret;
  }
}
```

Fig 2b: pop() in a Treiber Stack

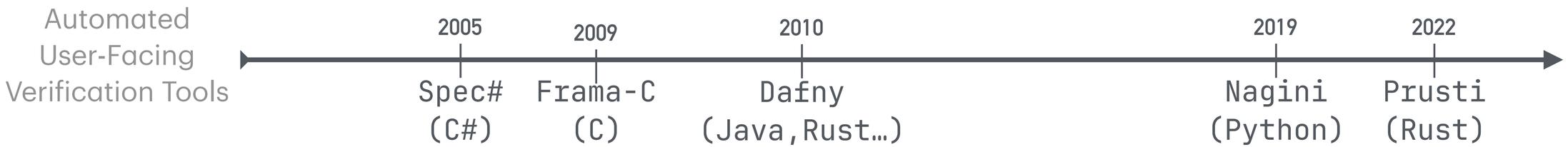
: under certain assumptions

Intermediate Verification Languages

- SMT Solvers: powerful & versatile reasoning paradigm

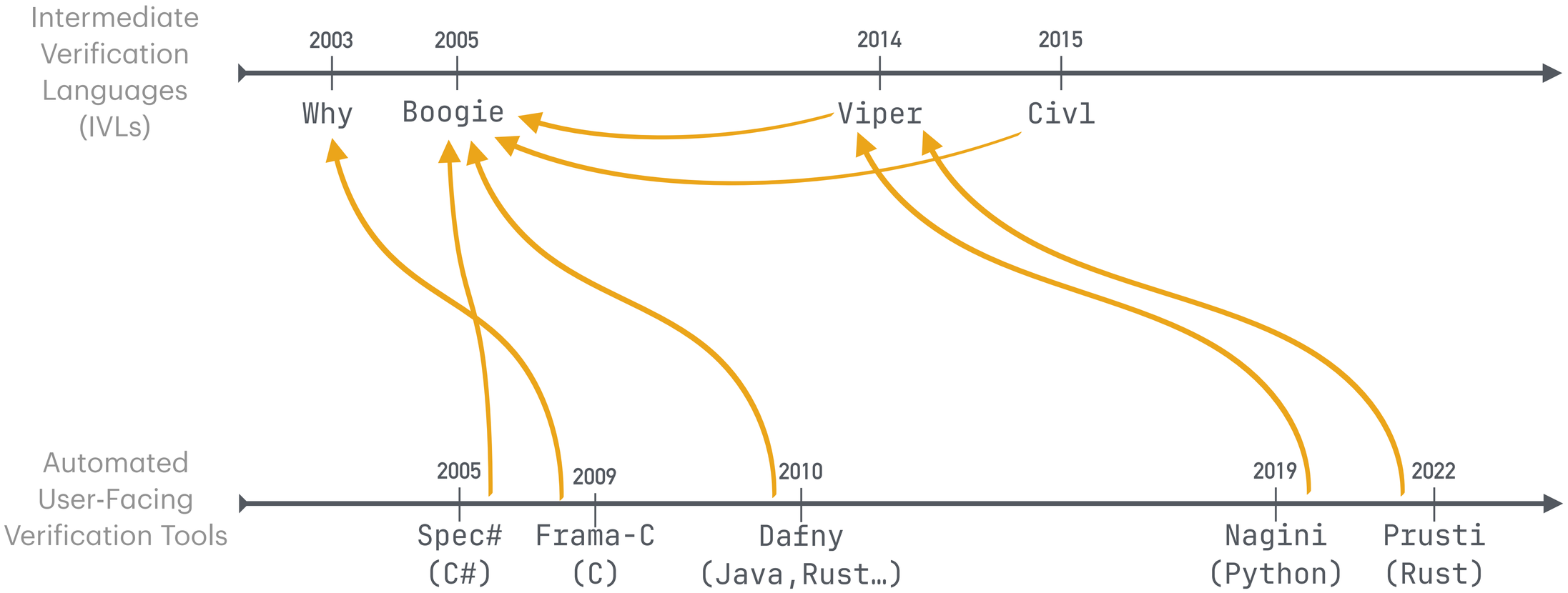
Intermediate Verification Languages

- SMT Solvers: powerful & versatile reasoning paradigm



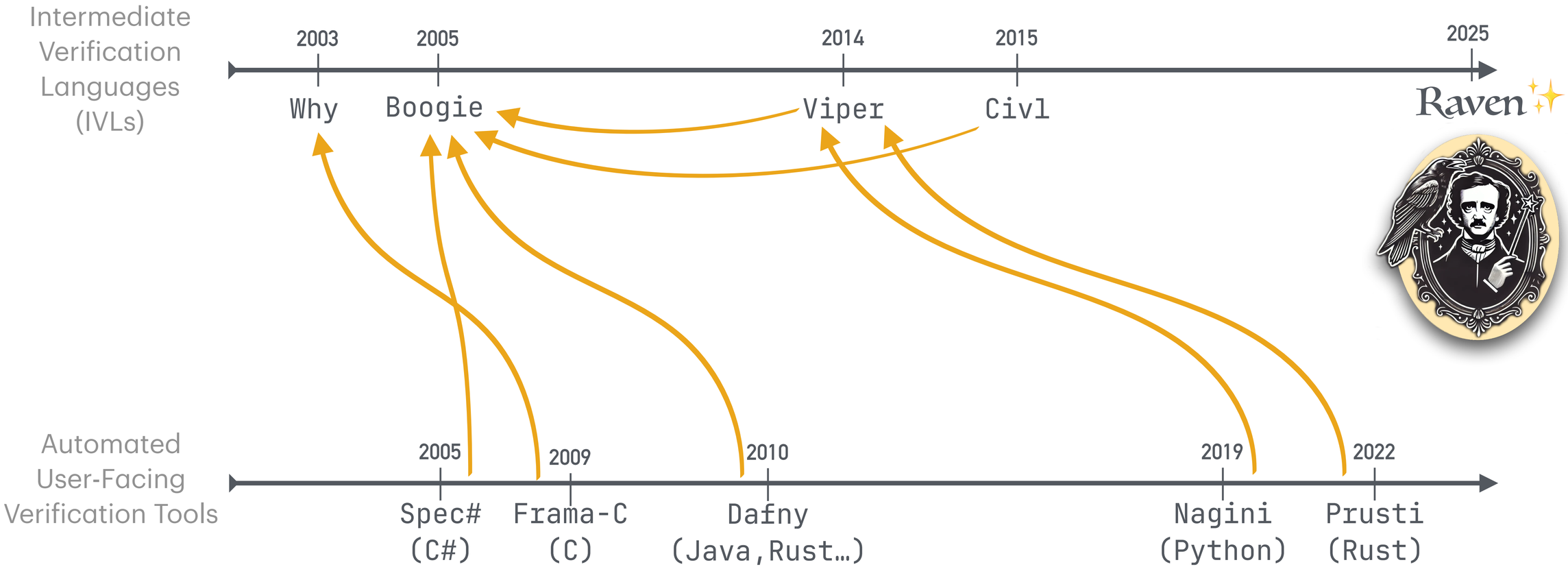
Intermediate Verification Languages

- SMT Solvers: powerful & versatile reasoning paradigm



Intermediate Verification Languages

- SMT Solvers: powerful & versatile reasoning paradigm

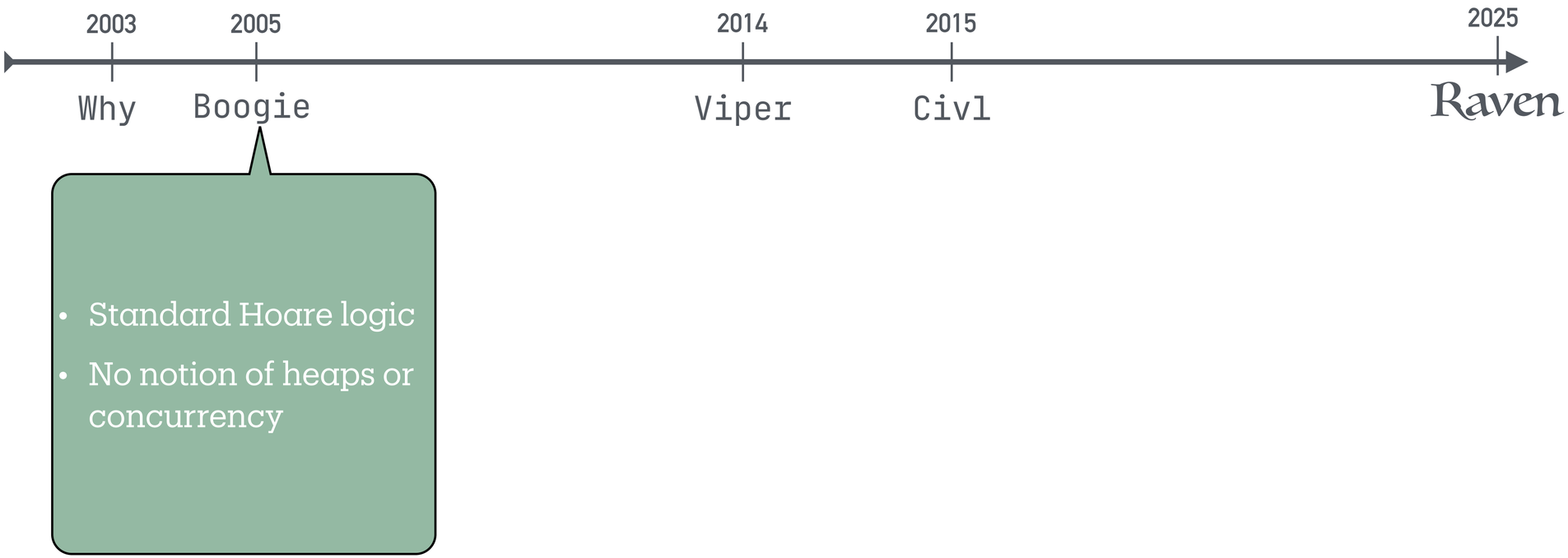


Intermediate Verification Languages

Intermediate Verification Languages (IVLs)

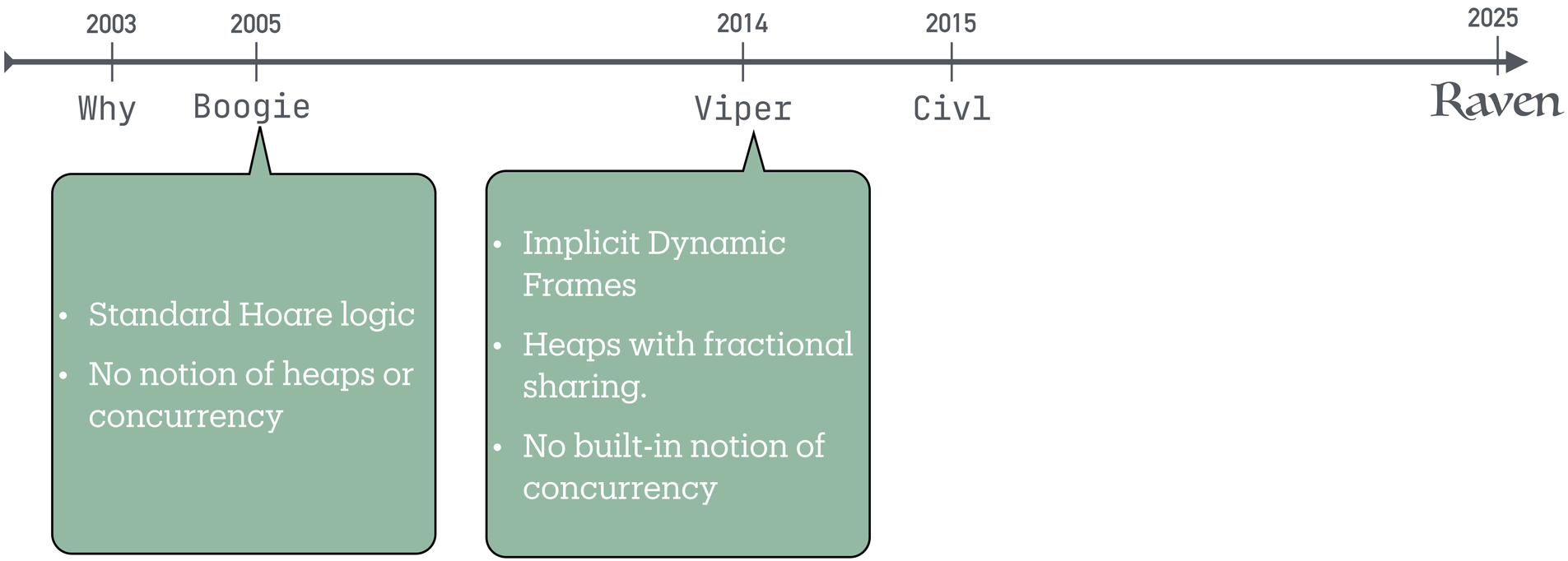
Intermediate Verification Languages

Intermediate Verification Languages (IVLs)



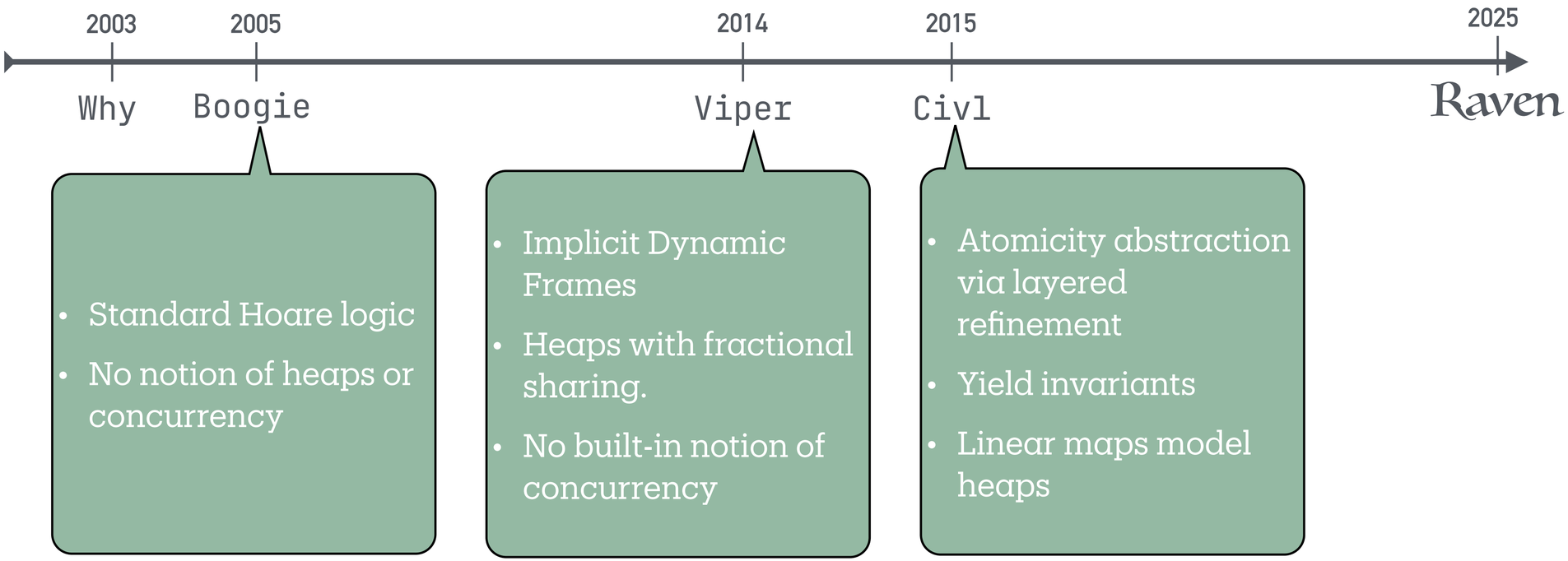
Intermediate Verification Languages

Intermediate Verification Languages (IVLs)



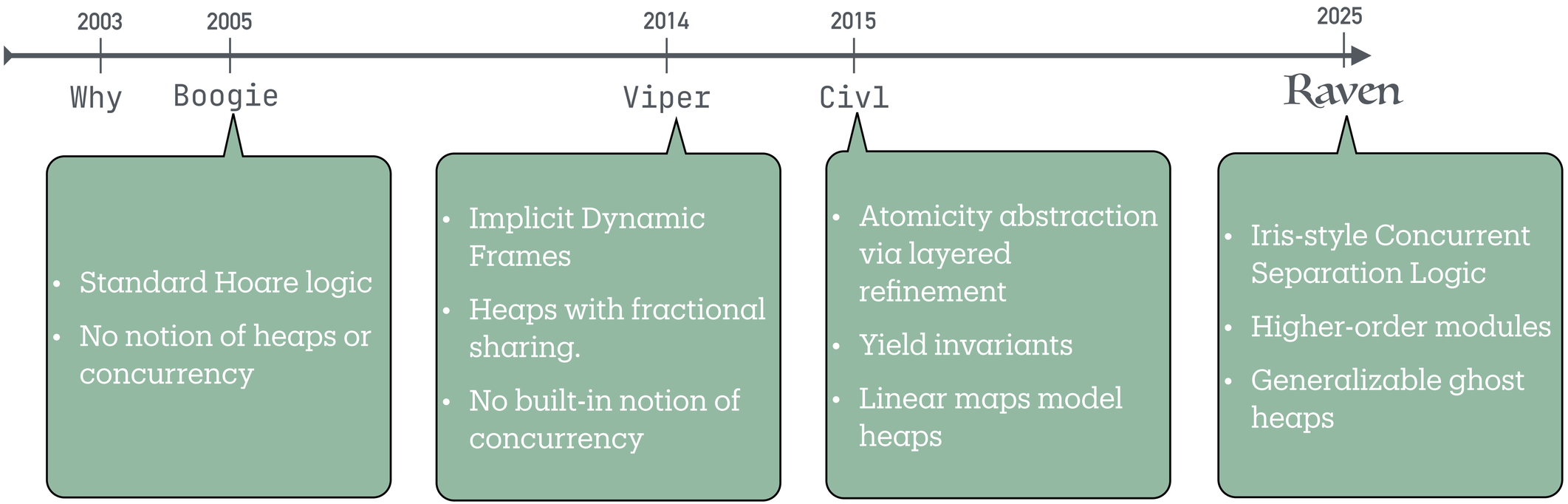
Intermediate Verification Languages

Intermediate Verification Languages (IVLs)



Intermediate Verification Languages

Intermediate Verification Languages (IVLs)



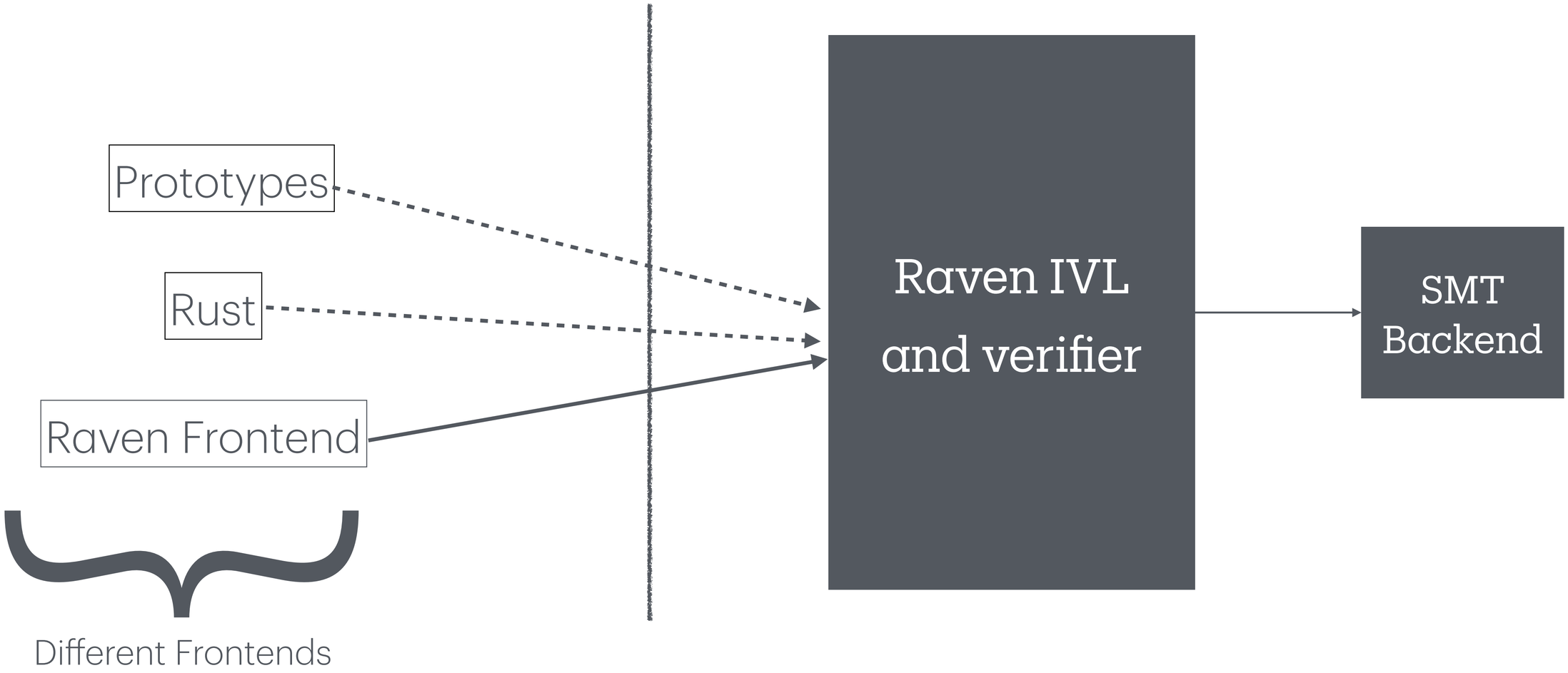
Raven Demo!

[*Gupta*, Patel, Wies; CAV'25]

Try it out!

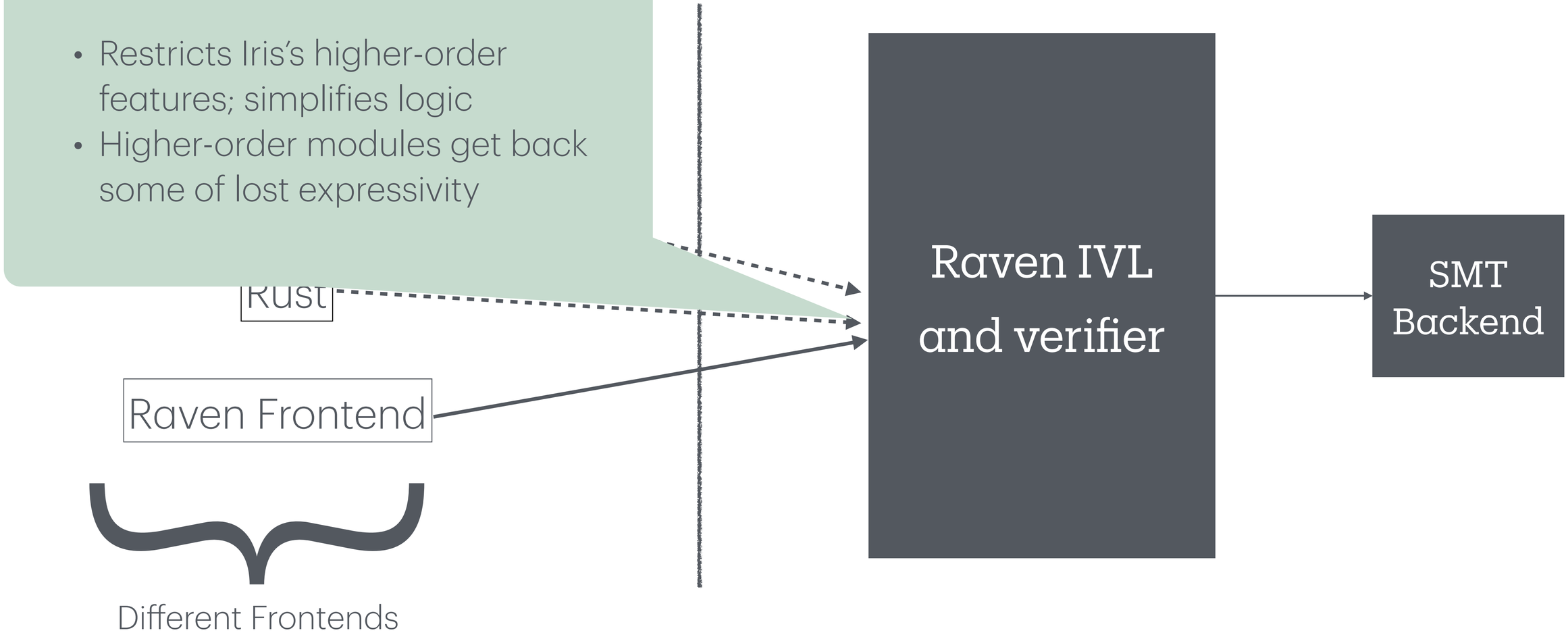
https://github.com/nyu-acsys/raven

Raven Pipeline

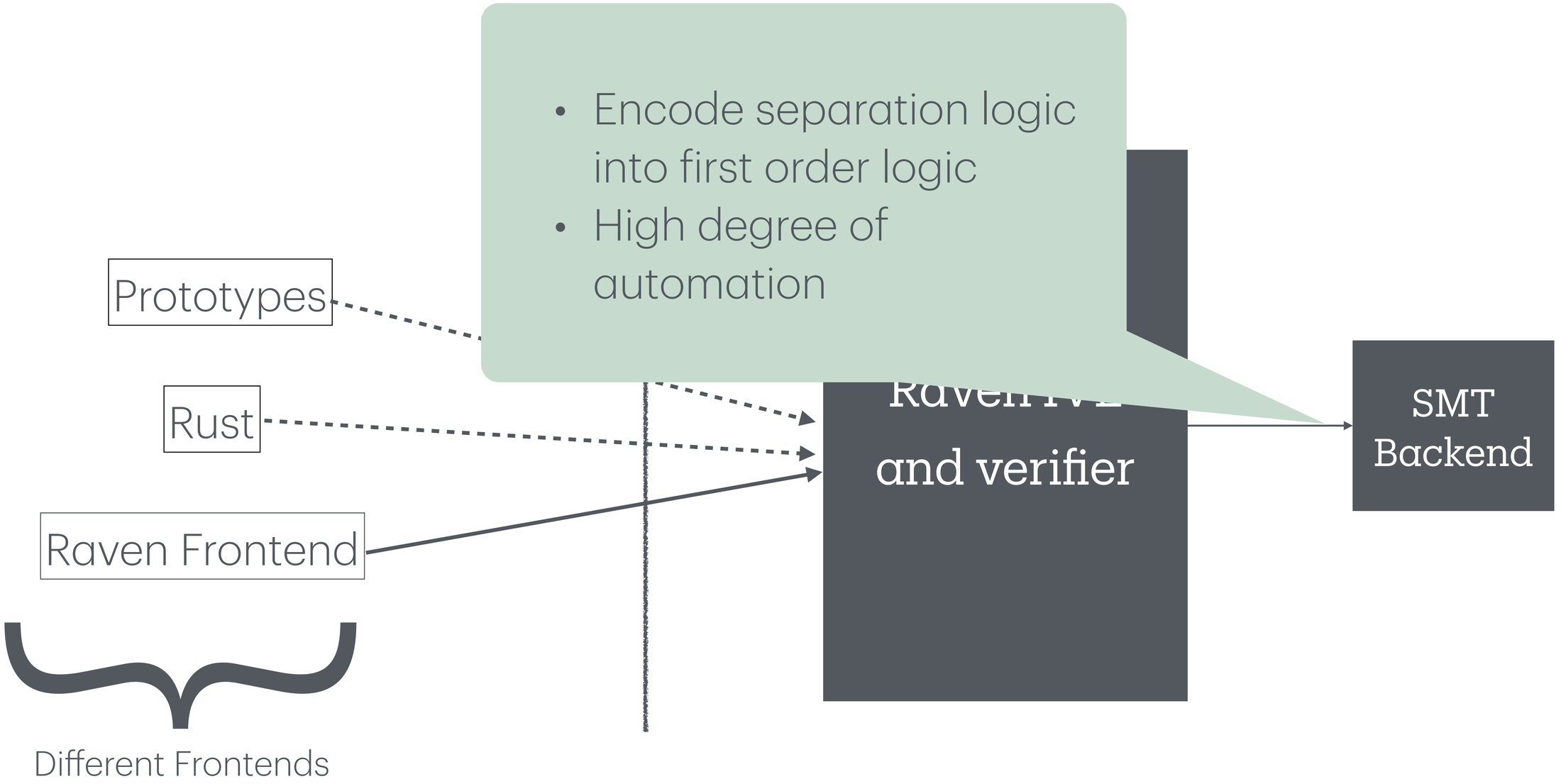


Raven Pipeline

- Restricts Iris's higher-order features; simplifies logic
- Higher-order modules get back some of lost expressivity

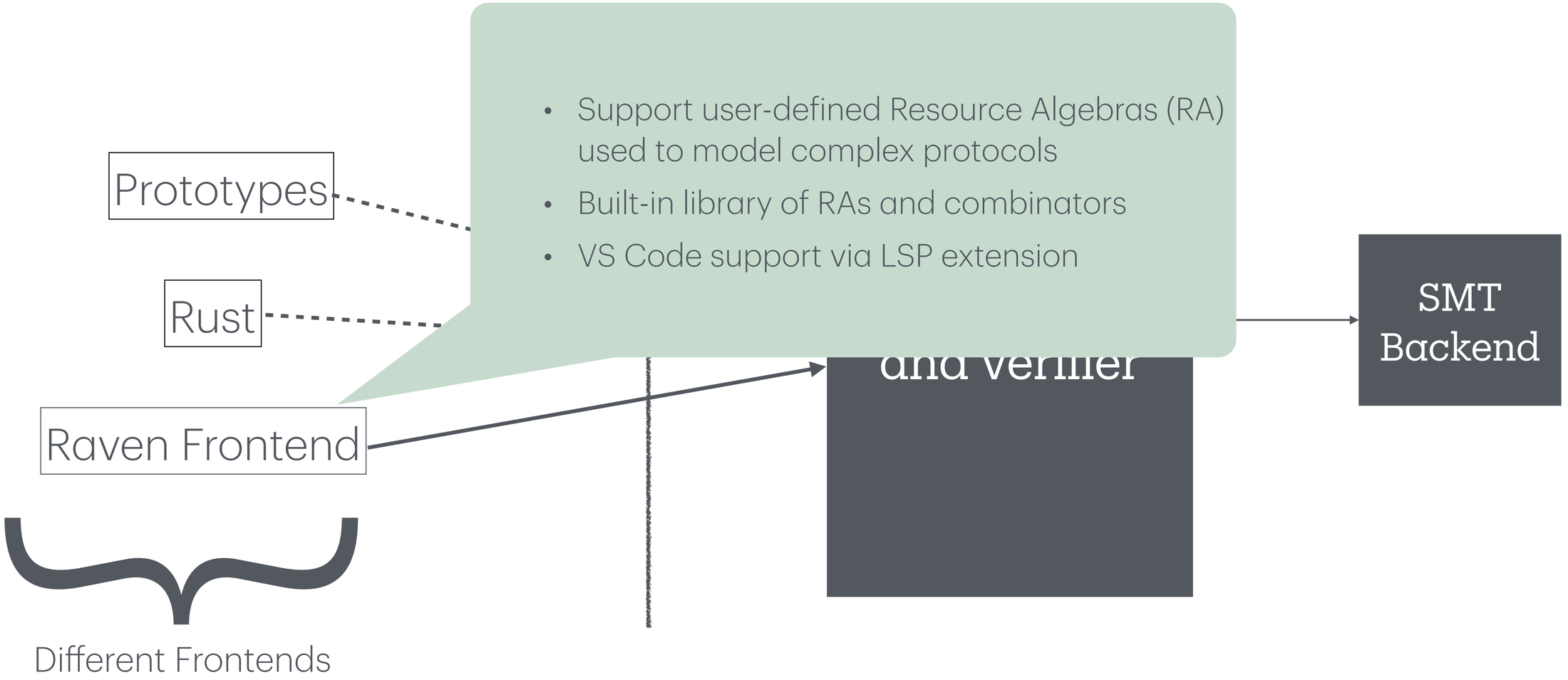


Raven Pipeline

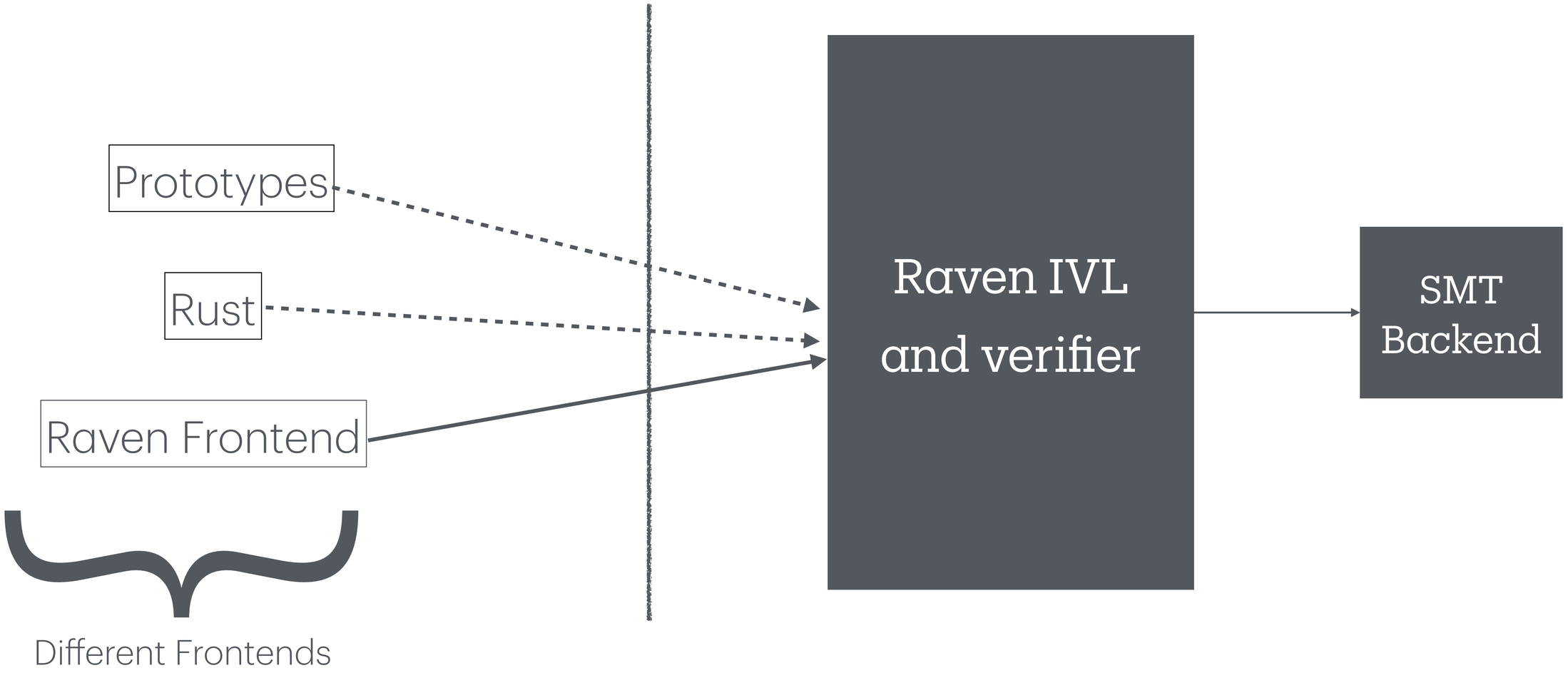


- Encode separation logic into first order logic
- High degree of automation

Raven Pipeline



Raven Pipeline



Implementation & Evaluation

- Implemented in 16K lines of OCaml
- Sizable benchmark set of concurrent data structures:
  - various locks, counters, Treiber stack, Michael Scott queue, concurrent B-tree, ...

Total	**Raven**	**Iris**	**Grasshopper**
Proof Overhead (# of instructions)	498	3687	423
Runtime	***13.97s***	*71.71s*	*26.93s*

Raven vs Iris + GRASSHopper — components of a larger case study, the GiveUp Template

Total	**Raven**	**Diaframe**
Proof Overhead (# of instructions)	1077	1067
Runtime	***14.16s***	*945.65s*

Raven vs Diaframe — set of 23 benchmarks

ExtensionAPI

[Under Submission]

ExtensionAPI

[Under Submission]

- Front-ends often implement Raven encodings for their custom primitives:
  - implement a custom parser and encoding scheme
  - non-modular, error-prone, redundant effort
  - Re-duplicates efforts across the pipeline

- Add custom ***commands/expressions/types*** by directly integrating with existing codebase pipeline
- Rapid prototyping of custom front-ends
- Reuse of existing codebase as much as possible

ExtensionAPI

[Under Submission]

ExtensionAPI

[Under Submission]

- Model Probablistic Programming Logic **Eris** in Raven

Additional Work

ExtensionAPI

[Under Submission]

- Model Probablistic Programming Logic **Eris** in Raven

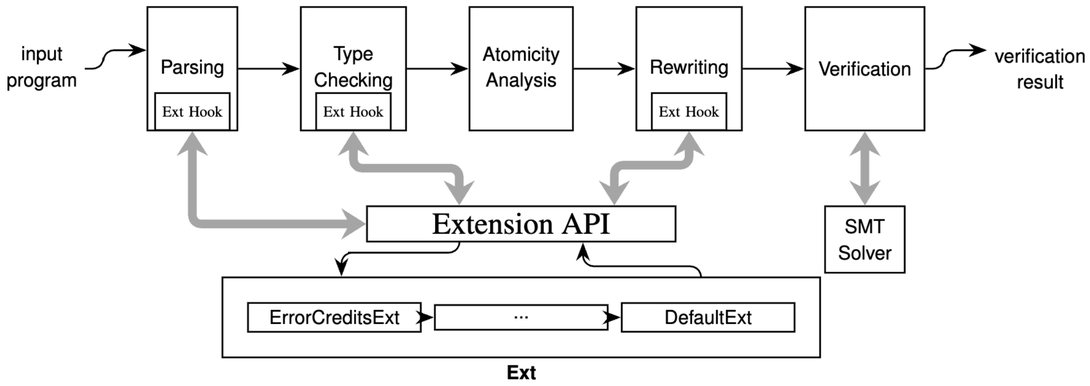


Figure 3: Raven's verification pipeline and extension API

Additional Work

ExtensionAPI

[Under Submission]

- Model Probablistic Programming Logic **Eris** in Raven

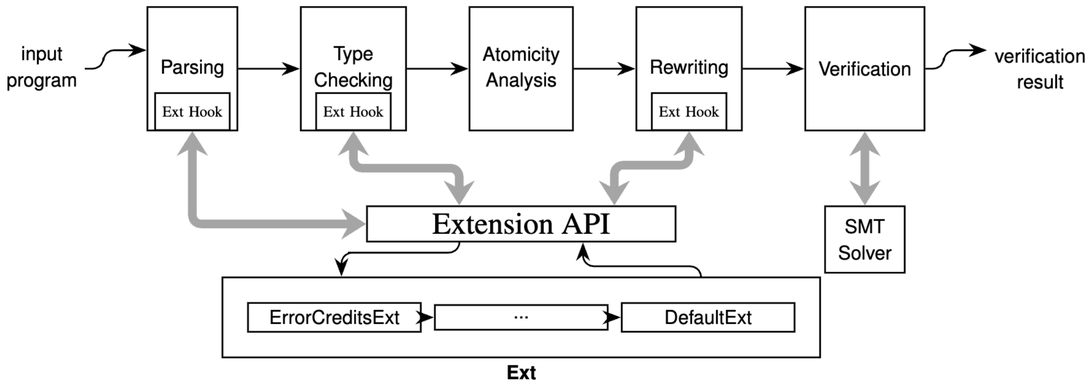


Figure 3: Raven's verification pipeline and extension API

```
proc doubleRand() returns (ret: Bool)
  requires EC.error(0.1)
  ensures ret == true
{
  var q1: Int := EC.rand(10);
  var q2: Int := EC.rand(10;
    ECFn: EC.error(0.1), (\ x :: (x == q1 ? 1.0 : 0.0))
  );

  if (q1 == q2) {
    EC.contra();
    return false;
  }

  return true;
}
```

Meta-Theory Soundness

- Mechanizing Raven's **meta-theory** in the Rocq implementation of *Iris*:
- F*oundational correctness* guarantees for Raven's formal semantics
- Formalize compatibility with the widely used *Iris* separation logic

Summary

Summary

- Raven encodes Iris-style *Concurrent Separation Logic* reasoning into *First-Order Logic*.

Summary

- Raven encodes Iris-style *Concurrent Separation Logic* reasoning into *First-Order Logic*.
- Robust automation, useful heuristics, user-friendly experience.

Summary

- Raven encodes Iris-style *Concurrent Separation Logic* reasoning into *First-Order Logic*.
- Robust automation, useful heuristics, user-friendly experience.
- Simplifies Iris logic; trades off expressivity for better automation.

Summary

- Raven encodes Iris-style *Concurrent Separation Logic* reasoning into *First-Order Logic*.
- Robust automation, useful heuristics, user-friendly experience.
- Simplifies Iris logic; trades off expressivity for better automation.
- Considerable benchmark set of verified data structures.

Summary

- Raven encodes Iris-style *Concurrent Separation Logic* reasoning into *First-Order Logic*.
- Robust automation, useful heuristics, user-friendly experience.
- Simplifies Iris logic; trades off expressivity for better automation.
- Considerable benchmark set of verified data structures.
- Working on additional front-ends targeting Raven IVL.

Summary

- Raven encodes Iris-style *Concurrent Separation Logic* reasoning into *First-Order Logic*.
- Robust automation, useful heuristics, user-friendly experience.
- Simplifies Iris logic; trades off expressivity for better automation.
- Considerable benchmark set of verified data structures.
- Working on additional front-ends targeting Raven IVL.

Try Raven Out!

Summary

- Raven encodes Iris-style *Concurrent Separation Logic* reasoning into *First-Order Logic*.
- Robust automation, useful heuristics, user-friendly experience.
- Simplifies Iris logic; trades off expressivity for better automation.
- Considerable benchmark set of verified data structures.
- Working on additional front-ends targeting Raven IVL.

Try Raven Out!

Reach out: `ekansh@nyu.edu`

Questions?