Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security

Marco Eilers
Severin Meier
Peter Müller

ETH zürich

Noninterference

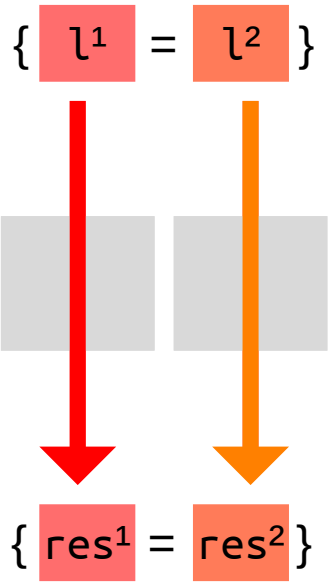
```
def compute(l: Int, h: Int) -> Int:
  var res;
  if (h > 0):
    res := 1
  else:
    res := 2
  return res;
}
```

Noninterference

```
def compute(l: Int, h: Int) -> Int:
  var res;
  if (h > 0):
    res := 1
  else:
    res := 2
  return res;
}
```

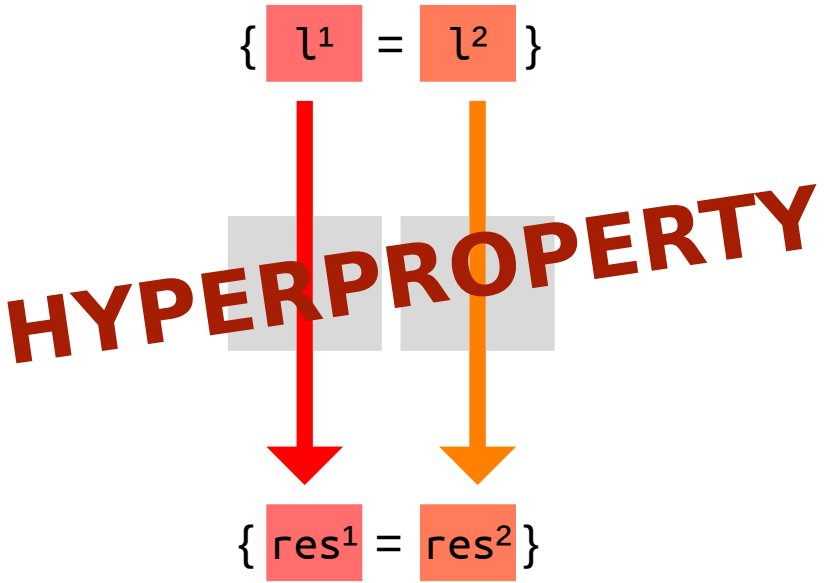
Noninterference

```
def compute(l: Int, h: Int) -> Int:
  var res;
  if (h > 0):
    res := 1
  else:
    res := 2
  return res;
}
```



Noninterference

```
def compute(l: Int, h: Int) -> Int:
  var res;
  if (h > 0):
    res := 1
  else:
    res := 2
  return res;
}
```



Self-composition and Product Programs

[Barthe et al., Math. Struc. Comp. Sci. 21(6), 2011]
[Barthe et al., FM 2011]

Self-composition and Product Programs

[Barthe et al., Math. Struc. Comp. Sci. 21(6), 2011]
[Barthe et al., FM 2011]

```
var res1;
res1 := foo(l1)
...
```

```
var res2;
res2 := foo(l2)
...
```

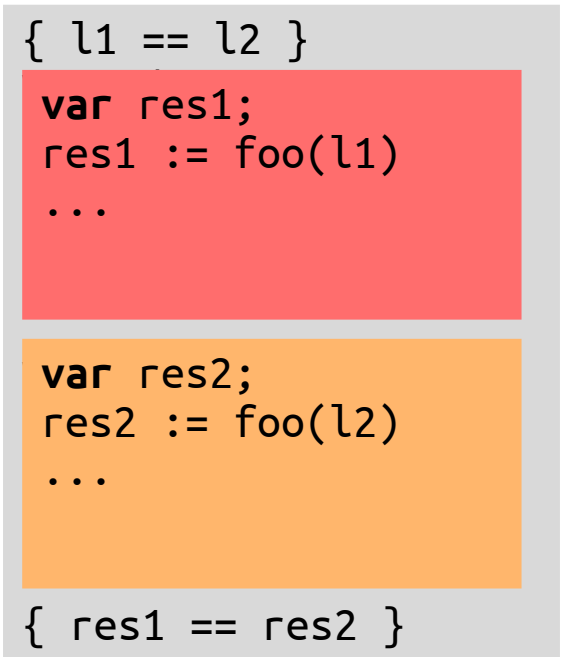
Self-composition and Product Programs

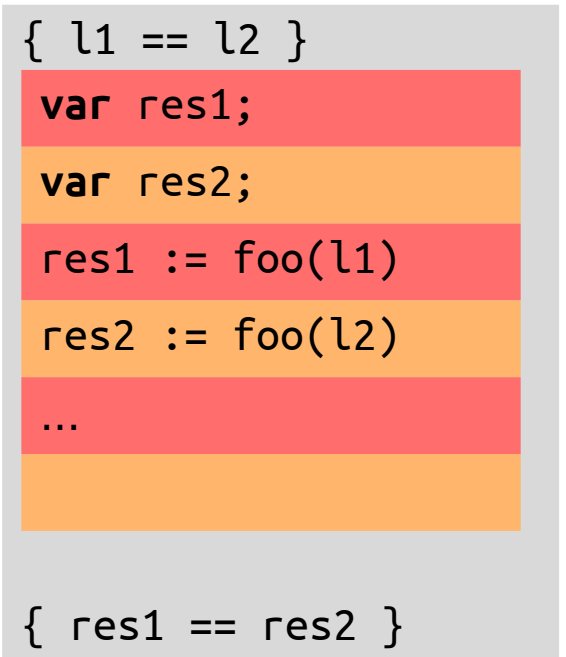
[Barthe et al., Math. Struc. Comp. Sci. 21(6), 2011]
[Barthe et al., FM 2011]

```
{ l1 == l2 }
var res1;
res1 := foo(l1)
...

var res2;
res2 := foo(l2)
...

{ res1 == res2 }
```

Self-composition and Product Programs

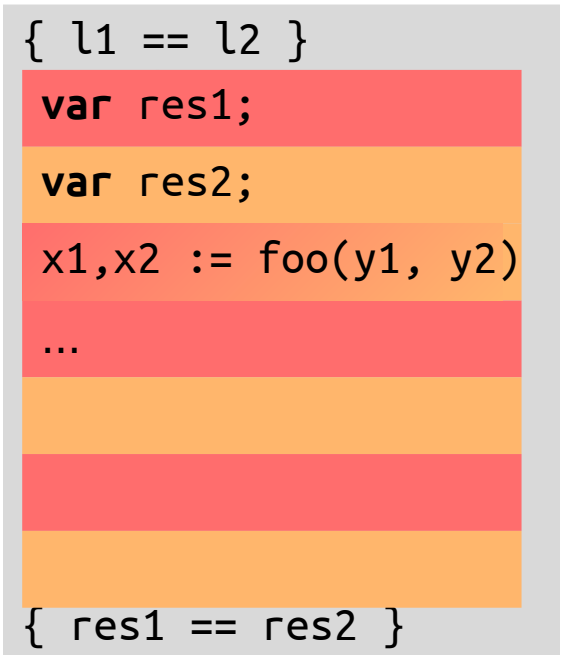
[Barthe et al., Math. Struc. Comp. Sci. 21(6), 2011]
[Barthe et al., FM 2011]





Modular Product Programs

[Eilers, Müller, Hitz, ESOP 2018]



Modular Product Programs

[Eilers, Müller, Hitz, ESOP 2018]

```
{ l1 == l2 }
var res1;
var res2;
x1,x2 := foo(y1, y2)
...
{ res1 == res2 }
```

- Calls *not* duplicated

Modular Product Programs

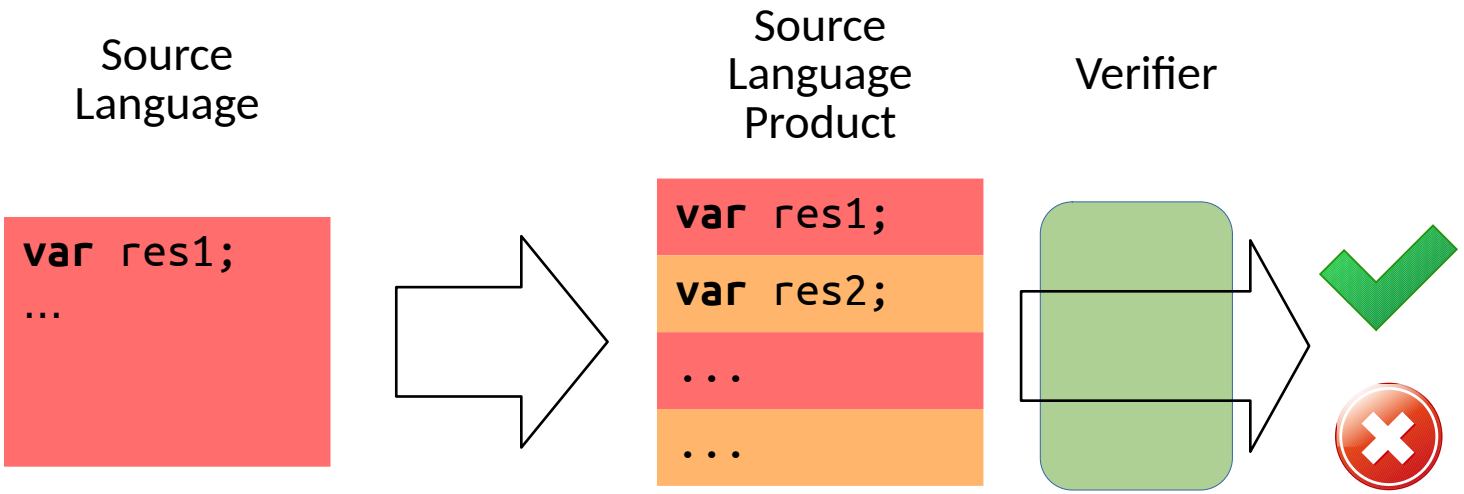
[Eilers, Müller, Hitz, ESOP 2018]

```
{ l1 == l2 }
var res1;
var res2;
x1,x2 := foo(y1, y2)
...
{ res1 == res2 }
```

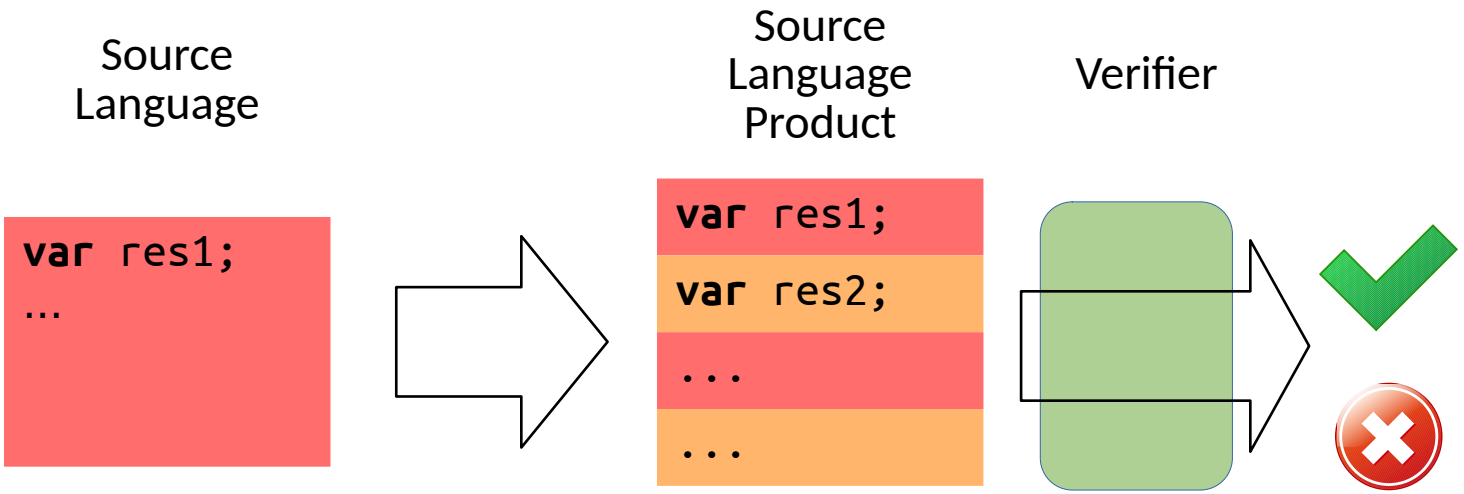
```
def foo(y1, y2)
  # ensures:
    result1 == result2
  ...
```

- Calls *not* duplicated
- Enables use of *relational* specifications

Problem: Real Languages

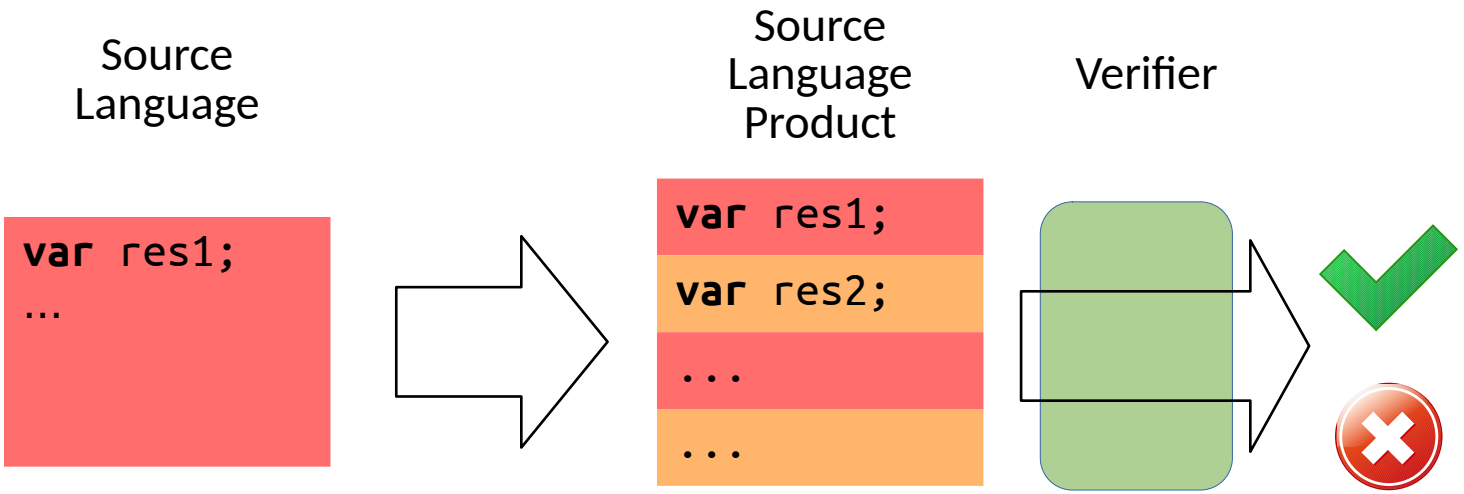


Problem: Real Languages



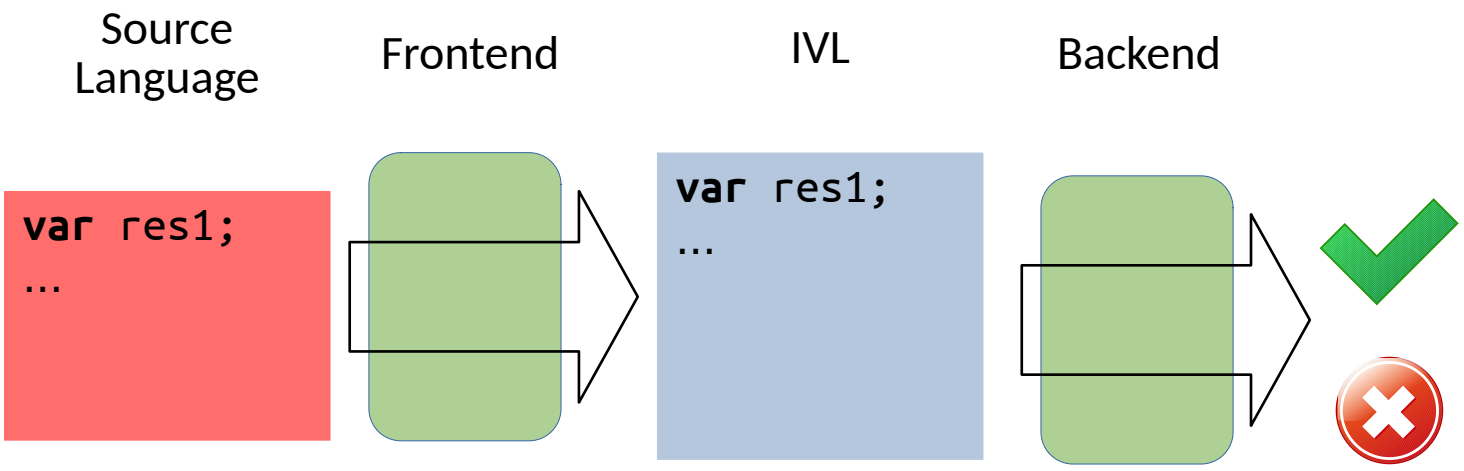
- Complex definition
- Specific to source language

Problem: Real Languages

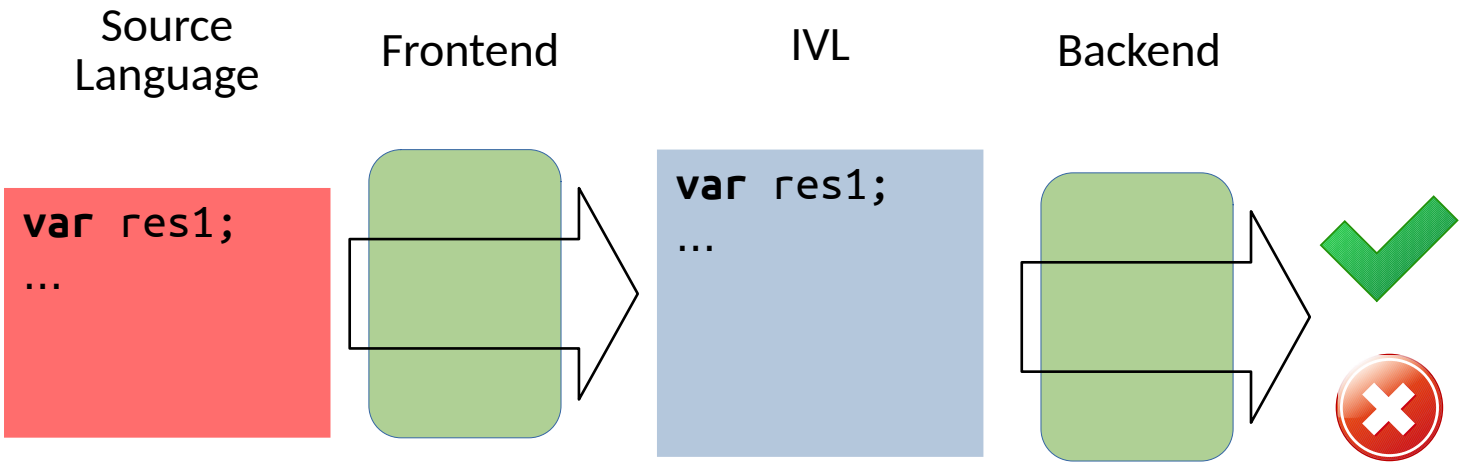


- Complex definition
- Specific to source language
- Only sequential

Typical Verifier Architecture

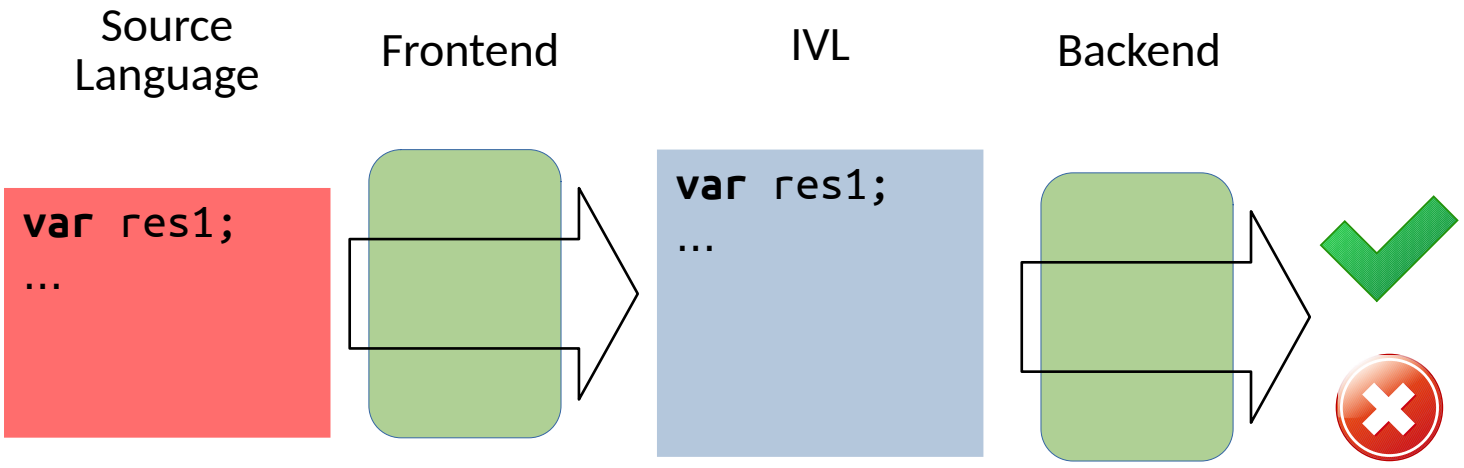


Typical Verifier Architecture



- Simple, sequential
- Examples: Boogie, Viper, Why3

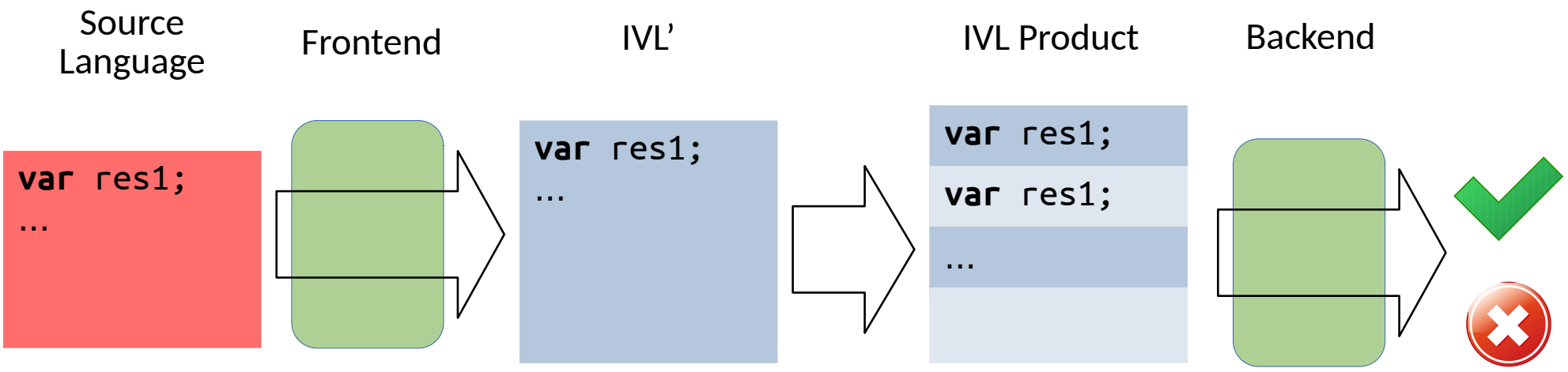
Typical Verifier Architecture



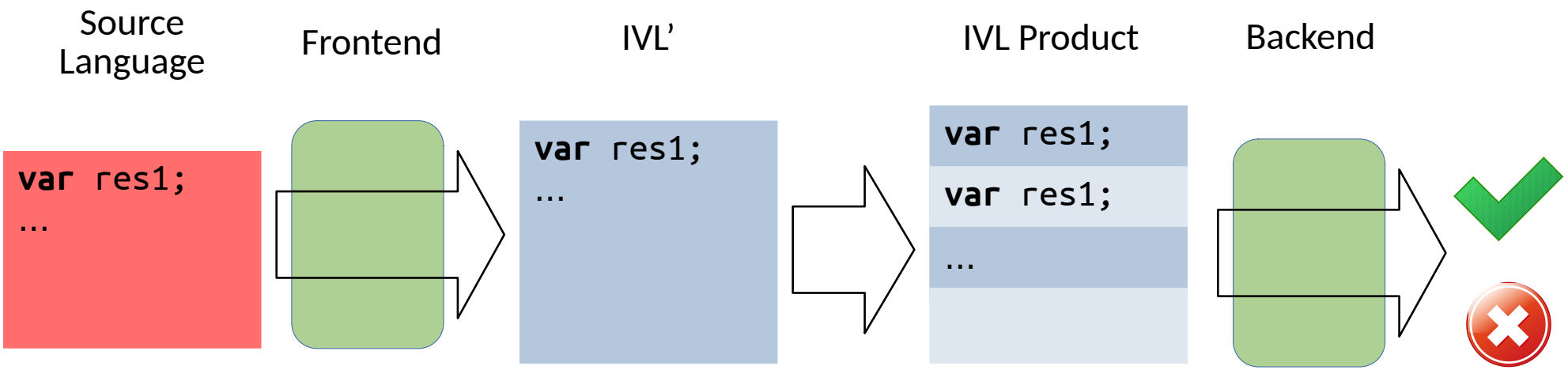
- Boogie: Dafny, VCC, GPUVerify
- Viper: Vercors, Prusti, Nagini
- Why3: Frama-C

- Simple, sequential
- Examples: Boogie, Viper, Why3

Proposed Architecture

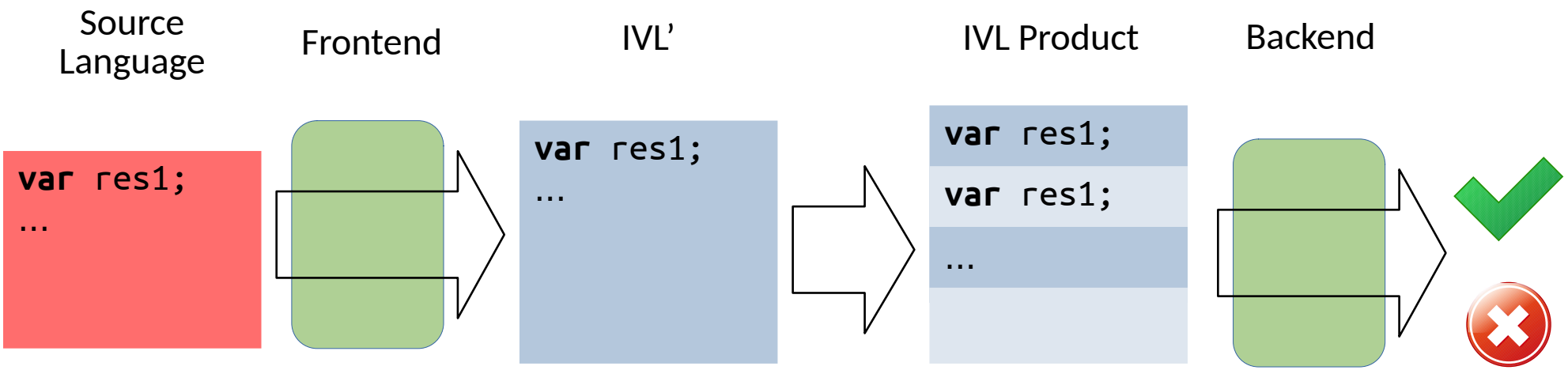


Proposed Architecture



- Simple
- Sequential

Proposed Architecture

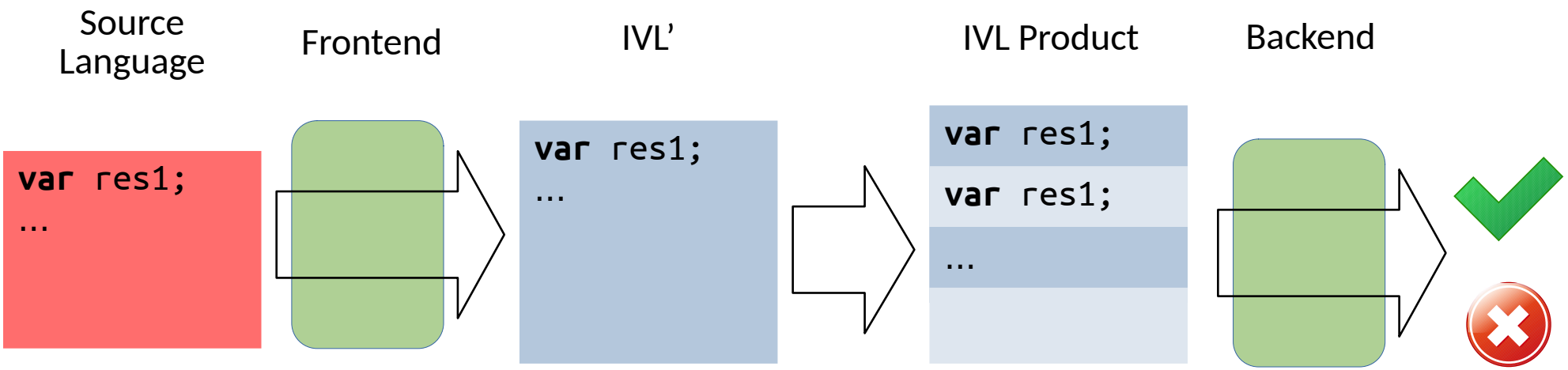


- Reused

- Simple
- Sequential
- Reusable

- Reused

Proposed Architecture



- Reused

- Simple
- Sequential
- Reusable

- Reused

Already used by SymDiff [Lahiri et al., CAV 2012]

Questions

Questions

What are the requirements to ensure soundness?

$\models$

Questions

What are the requirements to ensure soundness?

How can this approach be extended to concurrent programs?

$$\models$$

$$c \parallel c$$

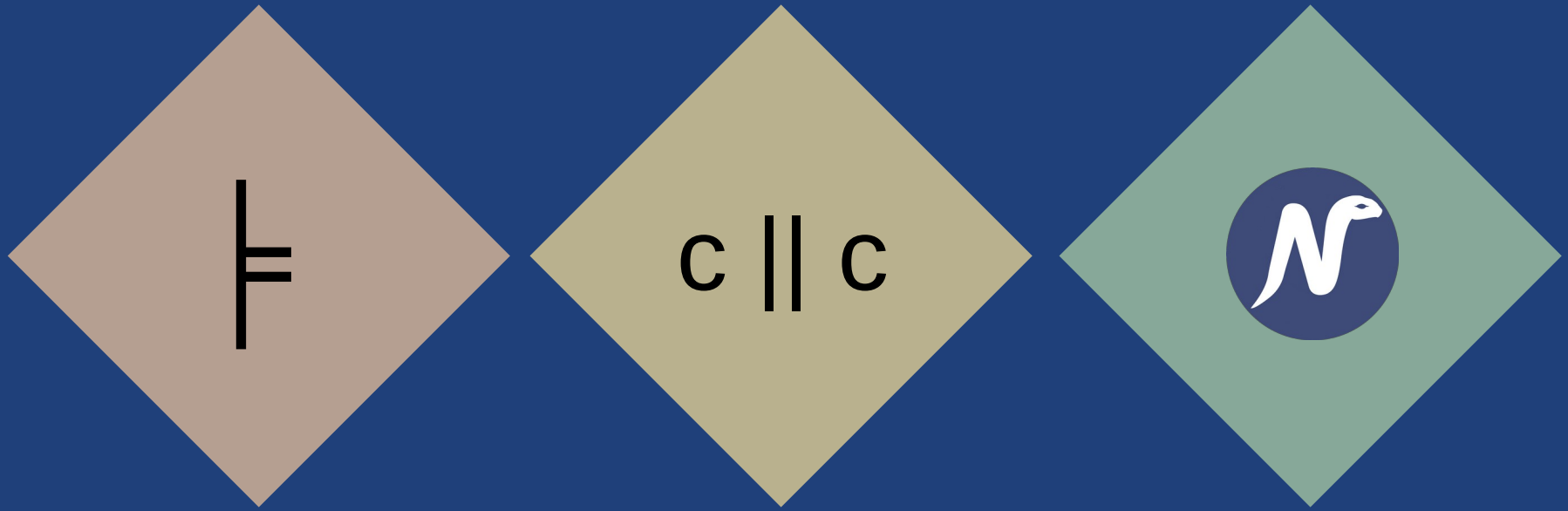
Questions

- What are the requirements to ensure soundness?
- How can this approach be extended to concurrent programs?
- Does this approach allow for expressive and performant verification in practice?

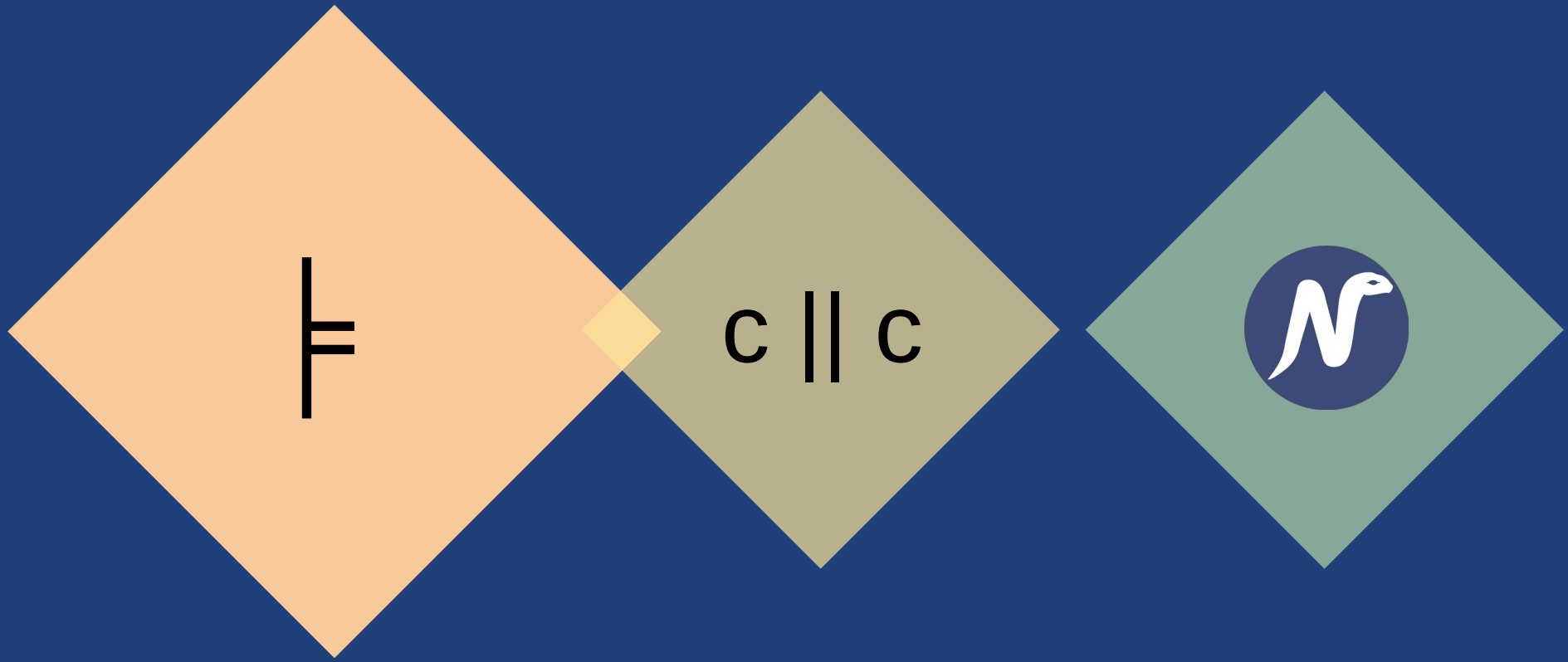
$\models$

$c \parallel c$

Overview



Soundness



Soundness Problem (1): Typical Encoding

```
def bar(a: A):
  # ensures result != 0
  return a.foo()
```

Soundness Problem (1): Typical Encoding

```
def bar(a: A):
  # ensures result != 0
  return a.foo()
```

```
class A:
  def foo(self):
    # ensures result >= 1
    return 1

class B(A):
  def foo(self):
    # ensures result >= 2
    return 2
```

Soundness Problem (1): Typical Encoding

```
def bar(a: A):
  # ensures result != 0
  return a.foo()
```

```
assert
  (result >= 2) ==>
  (result >= 1)
```

```
class A:
  def foo(self):
    # ensures result >= 1
    return 1

class B(A):
  def foo(self):
    # ensures result >= 2
    return 2
```

Soundness Problem (1): Typical Encoding

```
def bar(a: A):
  # ensures result != 0
  return a.foo()
```

```
class A:
  def foo(self):
    # ensures result >= 1
    return 1

class B(A):
  def foo(self):
    # ensures result >= 2
    return 2
```

```
assert
  (result >= 2) ==>
  (result >= 1)


def bar(a: A):
  # ensures result != 0
  assume result >= 1
```

Soundness Problem (1): Typical Encoding

```
def bar(a: A):
  # ensures result != 0
  return a.foo()
```

```
assert
  (result >= 2) ==>
  (result >= 1)


def bar(a: A):
  # ensures result != 0
  assume result >= 1
```

```
class A:
  def foo(self):
    # ensures result >= 1
    return 1

class B(A):
  def foo(self):
    # ensures result >= 2
    return 2
```

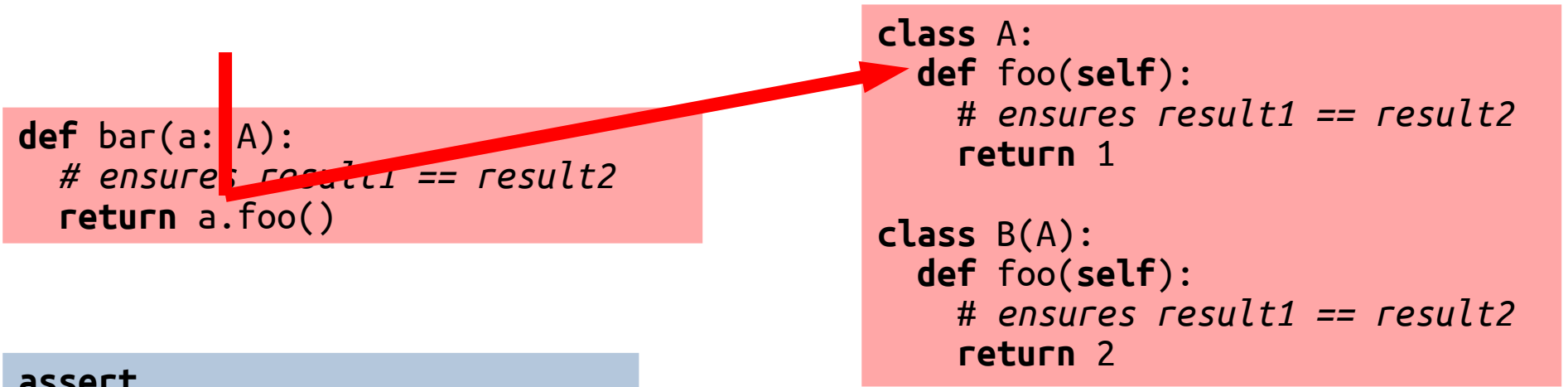
Soundness Problem (2): Unsound Case

```
def bar(a: A):
  # ensures result1 == result2
  return a.foo()
```

```
class A:
  def foo(self):
    # ensures result1 == result2
    return 1

class B(A):
  def foo(self):
    # ensures result1 == result2
    return 2
```

```
assert
  (result1 == result2) ==>
  (result1 == result2)


def bar(a: A):
  # ensures result1 == result2
  assume result1 == result2
```

Soundness Problem (2): Unsound Case

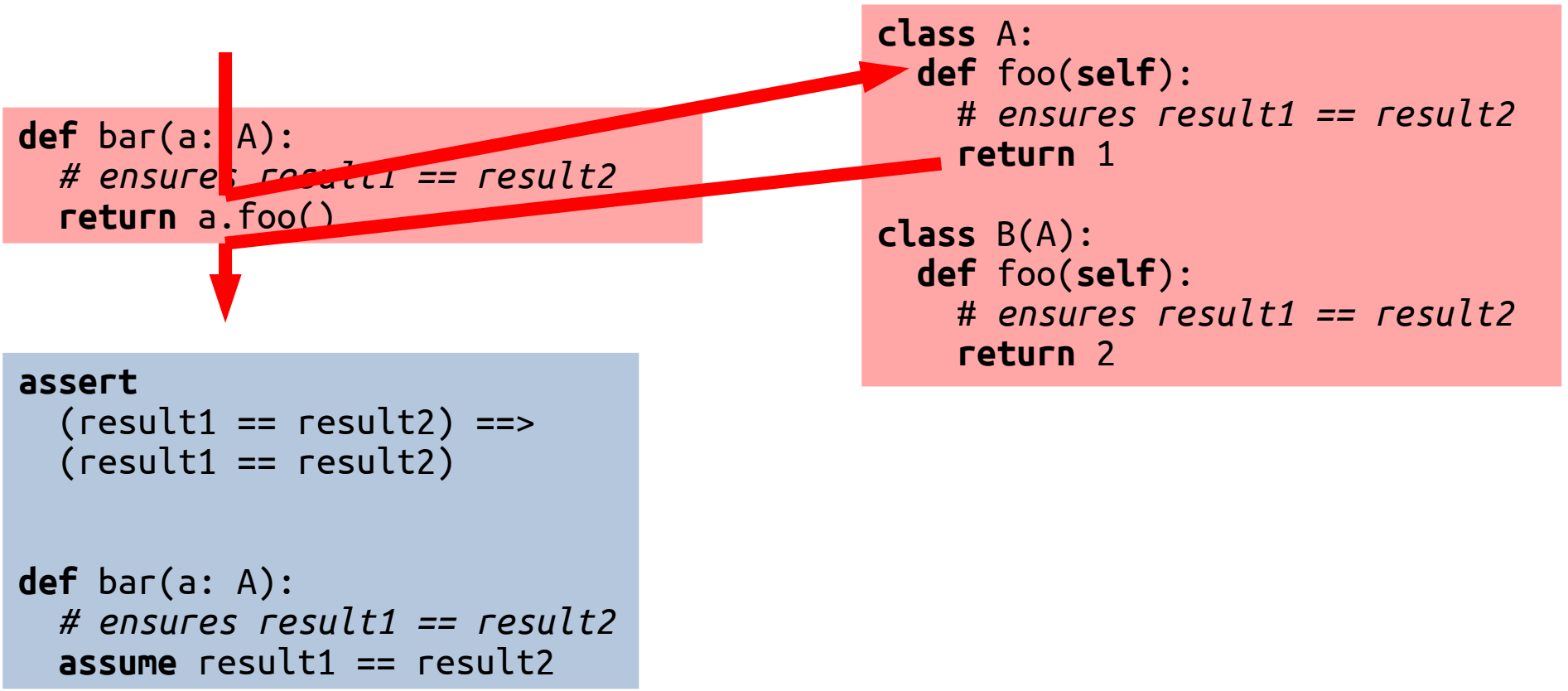


```
def bar(a: A):
  # ensures result1 == result2
  return a.foo()
```

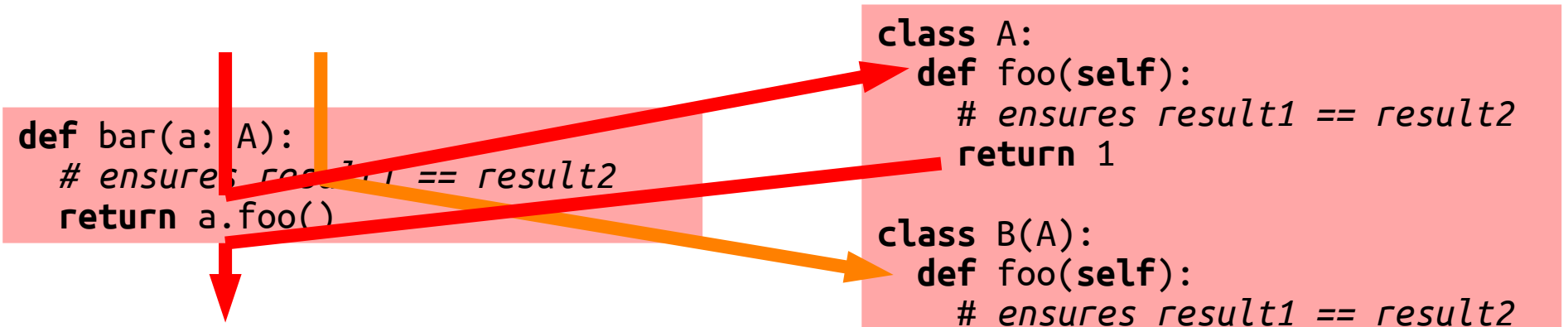
```
class A:
  def foo(self):
    # ensures result1 == result2
    return 1

class B(A):
  def foo(self):
    # ensures result1 == result2
    return 2
```

```
assert
  (result1 == result2) ==>
  (result1 == result2)


def bar(a: A):
  # ensures result1 == result2
  assume result1 == result2
```

Soundness Problem (2): Unsound Case



Soundness Problem (2): Unsound Case

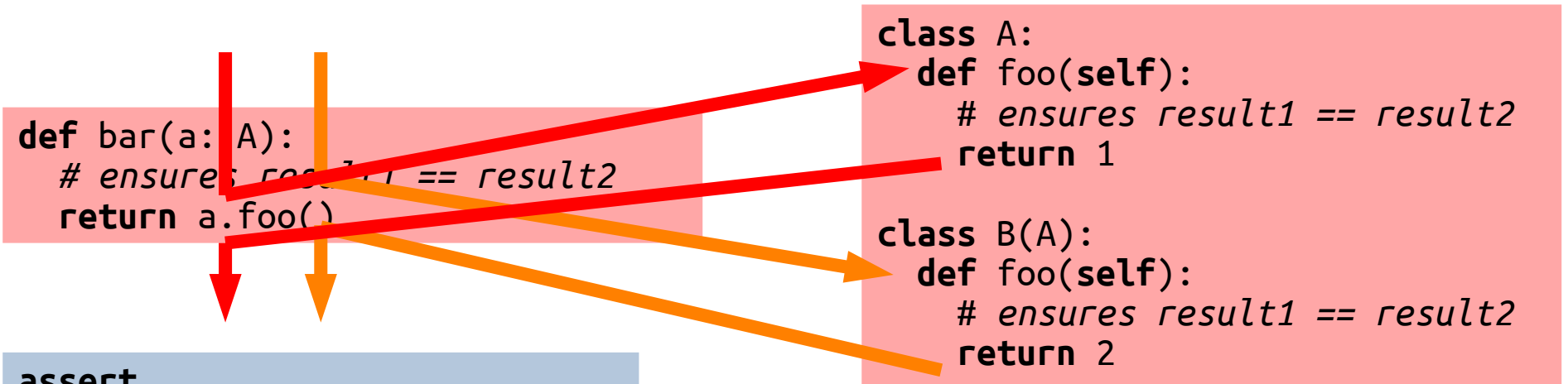


```
def bar(a: A):
  # ensures result1 == result2
  return a.foo()
```

```
class A:
  def foo(self):
    # ensures result1 == result2
    return 1

class B(A):
  def foo(self):
    # ensures result1 == result2
    return 2
```

```
assert
  (result1 == result2) ==>
  (result1 == result2)


def bar(a: A):
  # ensures result1 == result2
  assume result1 == result2
```

Soundness Problem (2): Unsound Case

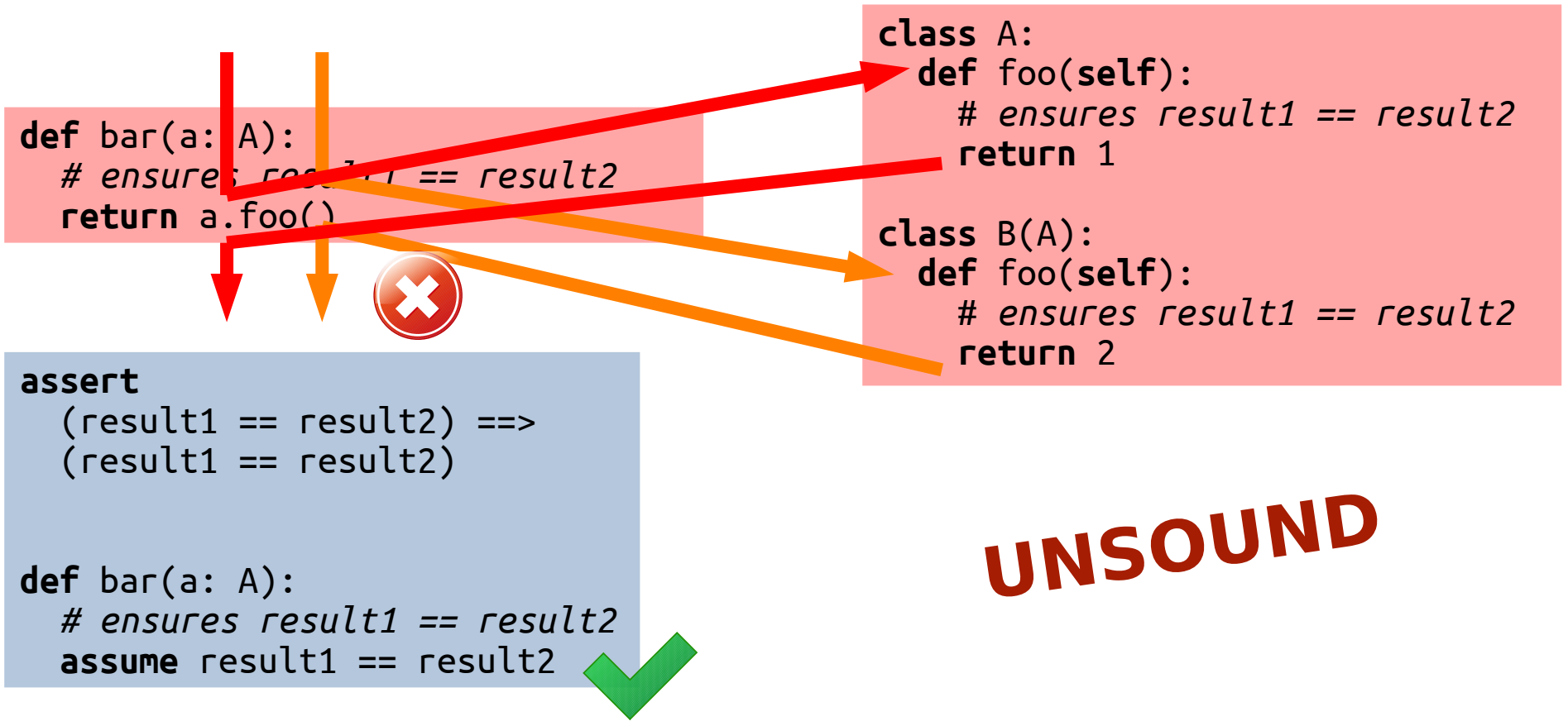


```
def bar(a: A):
  # ensures result1 == result2
  return a.foo()
```

```
class A:
  def foo(self):
    # ensures result1 == result2
    return 1

class B(A):
  def foo(self):
    # ensures result1 == result2
    return 2
```

```
assert
  (result1 == result2) ==>
  (result1 == result2)


def bar(a: A):
  # ensures result1 == result2
  assume result1 == result2
```

Soundness Problem (2): Unsound Case



Criterion: Operationality

Encodings that encode *operational* behavior are always sound

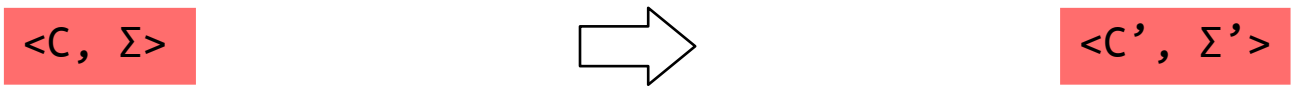
Criterion: Operationality

Encodings that encode *operational* behavior are always sound

<C, Σ>

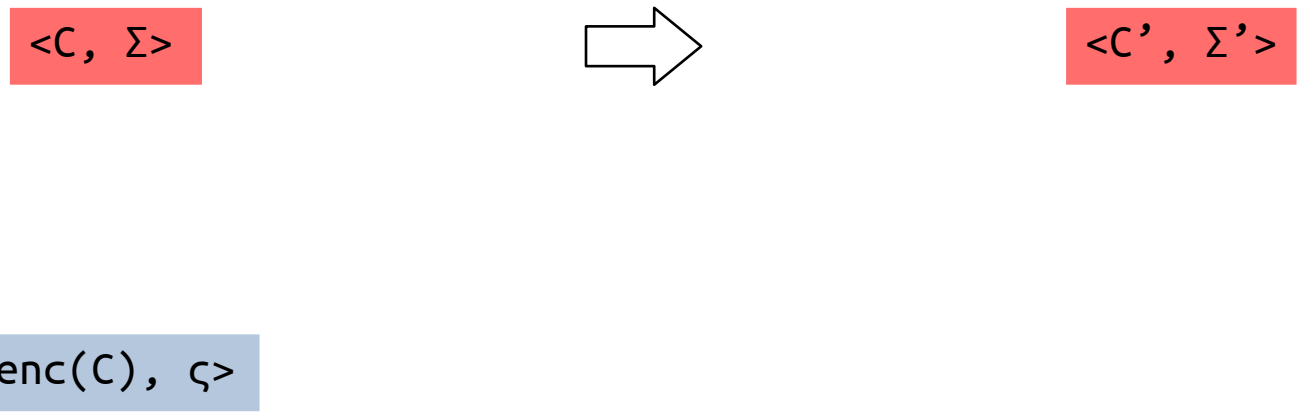
Criterion: Operationality

Encodings that encode *operational* behavior are always sound



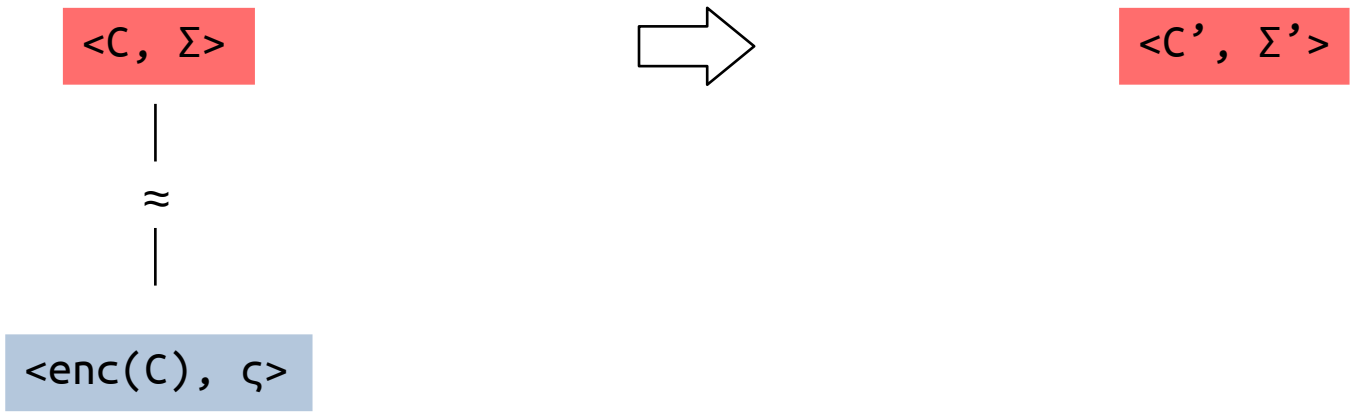
Criterion: Operationality

Encodings that encode *operational* behavior are always sound



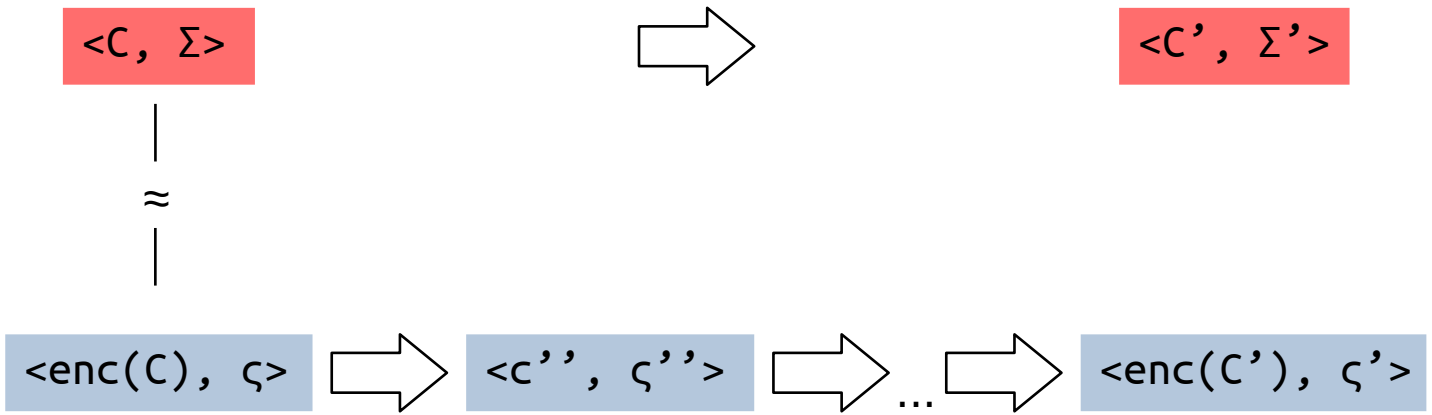
Criterion: Operationality

Encodings that encode *operational* behavior are always sound



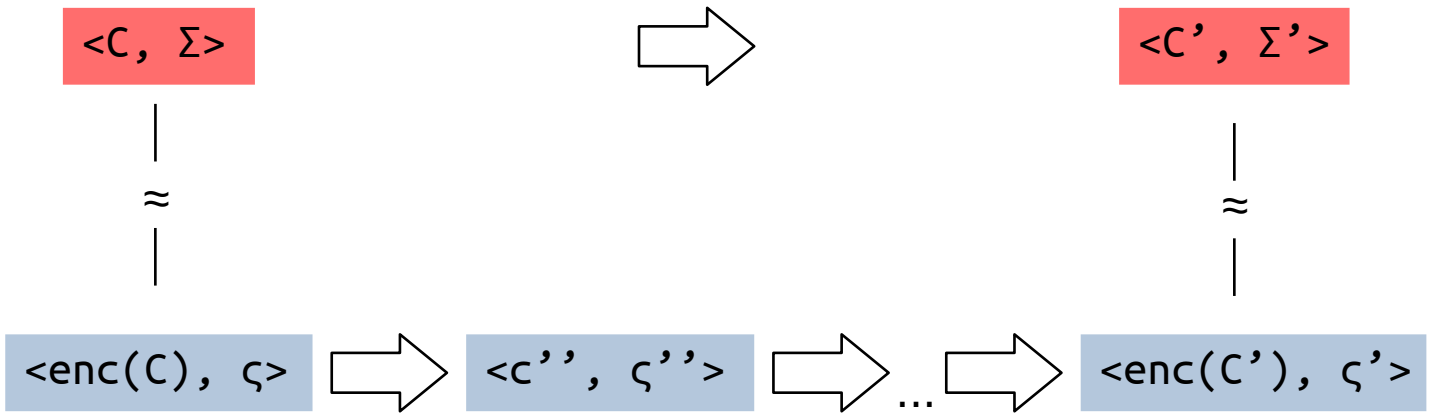
Criterion: Operationality

Encodings that encode *operational* behavior are always sound



Criterion: Operationality

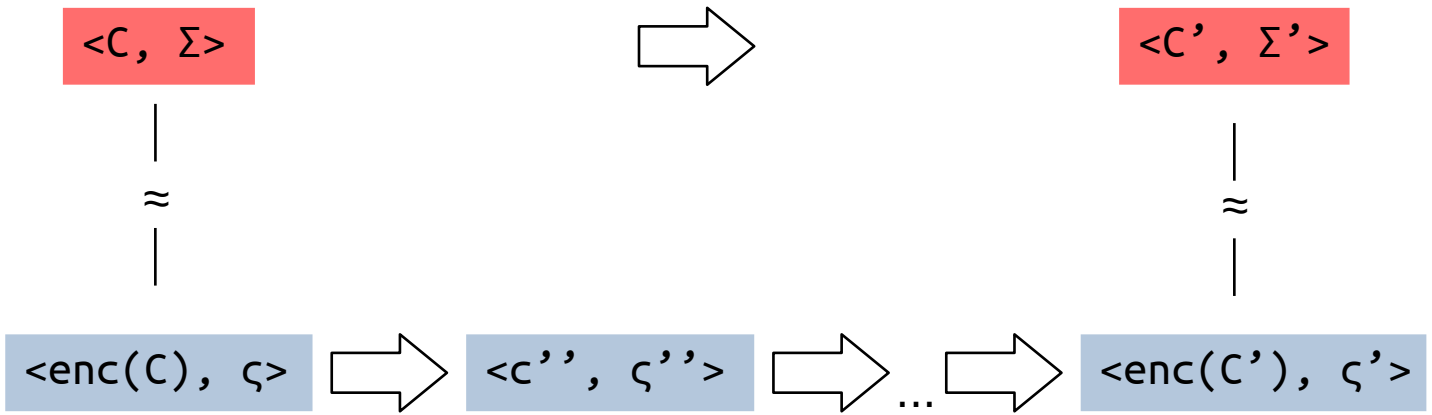
Encodings that encode *operational* behavior are always sound



Criterion: Operationality

Encodings that encode *operational* behavior are always sound

- Sufficient criterion, not necessary



Practical Relevance

- Typical: Frontend encodes *most* language features operationally
  - Main advantage of IVLs

Practical Relevance

- Typical: Frontend encodes *most* language features operationally
  - Main advantage of IVLs
- *Easily identify* non-operational parts
  - Often intentional to ensure modularity
  - Must be checked case-by-case
- Examples of non-operational encodings:
  - Dynamically-bound calls (Dafny, Nagini)
  - Local blocks (VCC)
  - Loops (Prusti)

Soundness Problem (3): Possible Solution

```
def bar(a: A):
  # ensures result1 == result2
  return a.foo()
```

```
assert
  (result1 == result2) ==>
  (result1 == result2)


def bar(a: A):
  # ensures result1 == result2
  assume result1 == result2
```

```
class A:
  def foo(self):
    # ensures result1 == result2
    return 1

class B(A):
  def foo(self):
    # ensures result1 == result2
    return 2
```

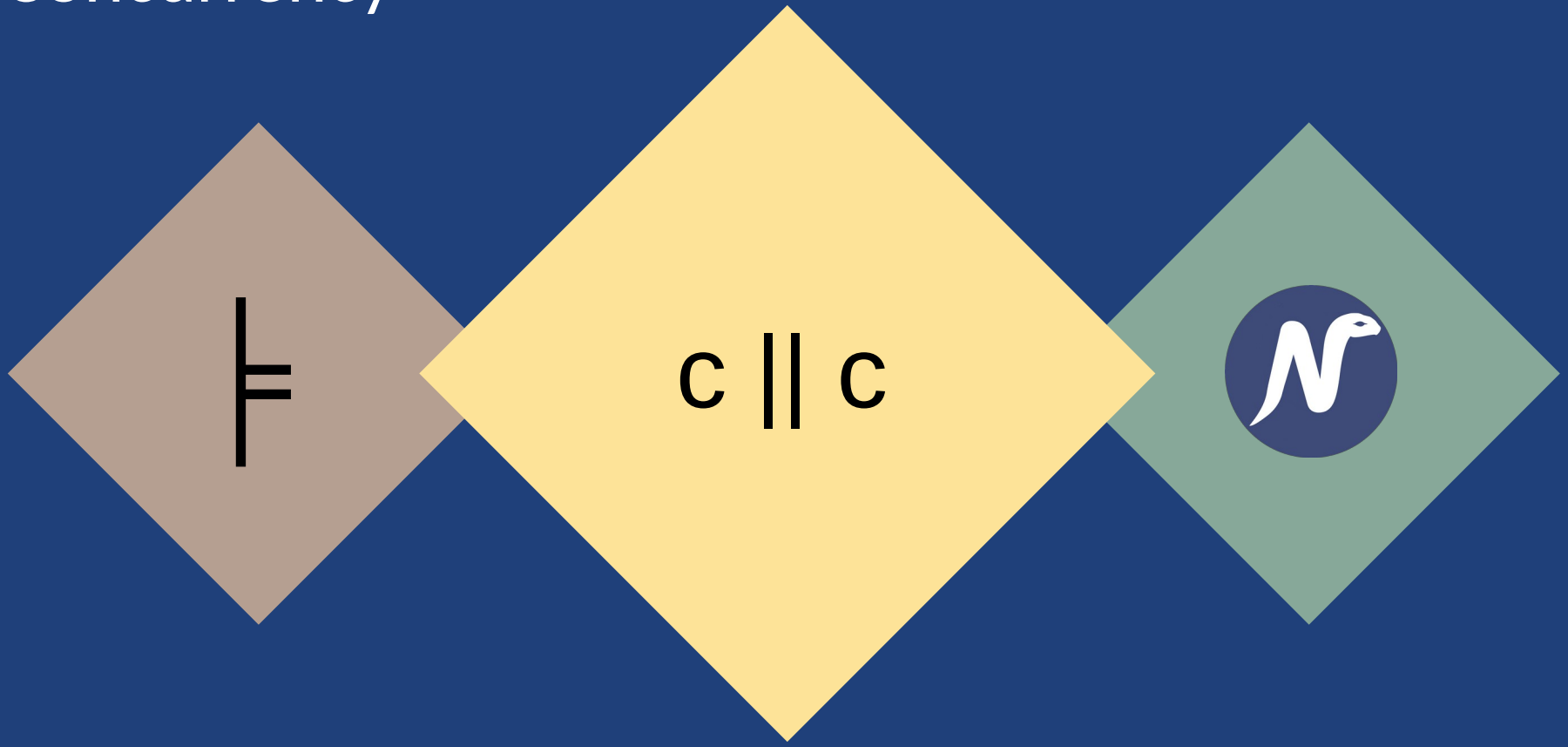
Soundness Problem (3): Possible Solution

```
def bar(a: A):
  # ensures result1 == result2
  return a.foo()
```

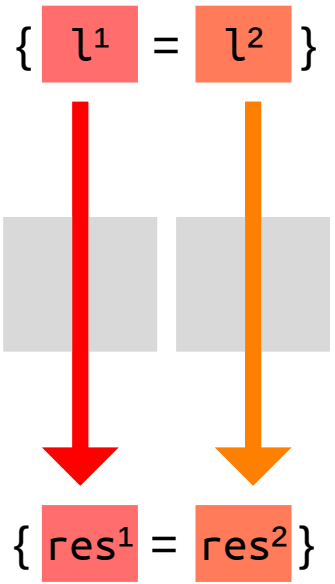
```
class A:
  def foo(self):
    # ensures result1 == result2
    return 1

class B(A):
  def foo(self):
    # ensures result1 == result2
    return 2
```

```
assert
  (result1 == result2) ==>
  (result1 == result2)


def bar(a: A):
  # ensures result1 == result2
  assume type(a1) == type(a2)
    ==> result1 == result2
```

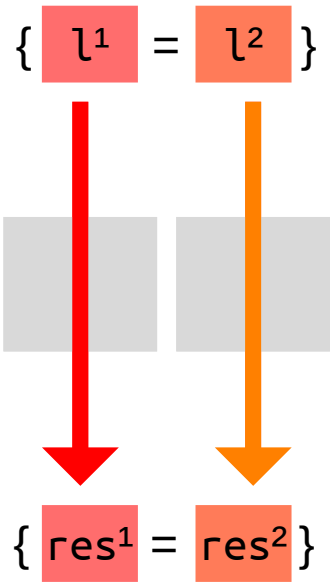
Concurrency



Concurrency: Noninterference Properties

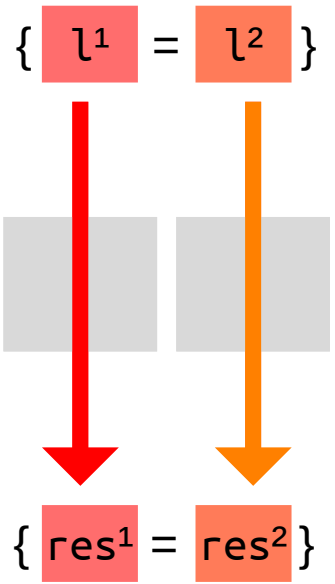


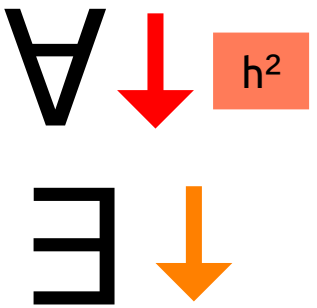
Concurrency: Noninterference Properties



Noninterference: Too strict

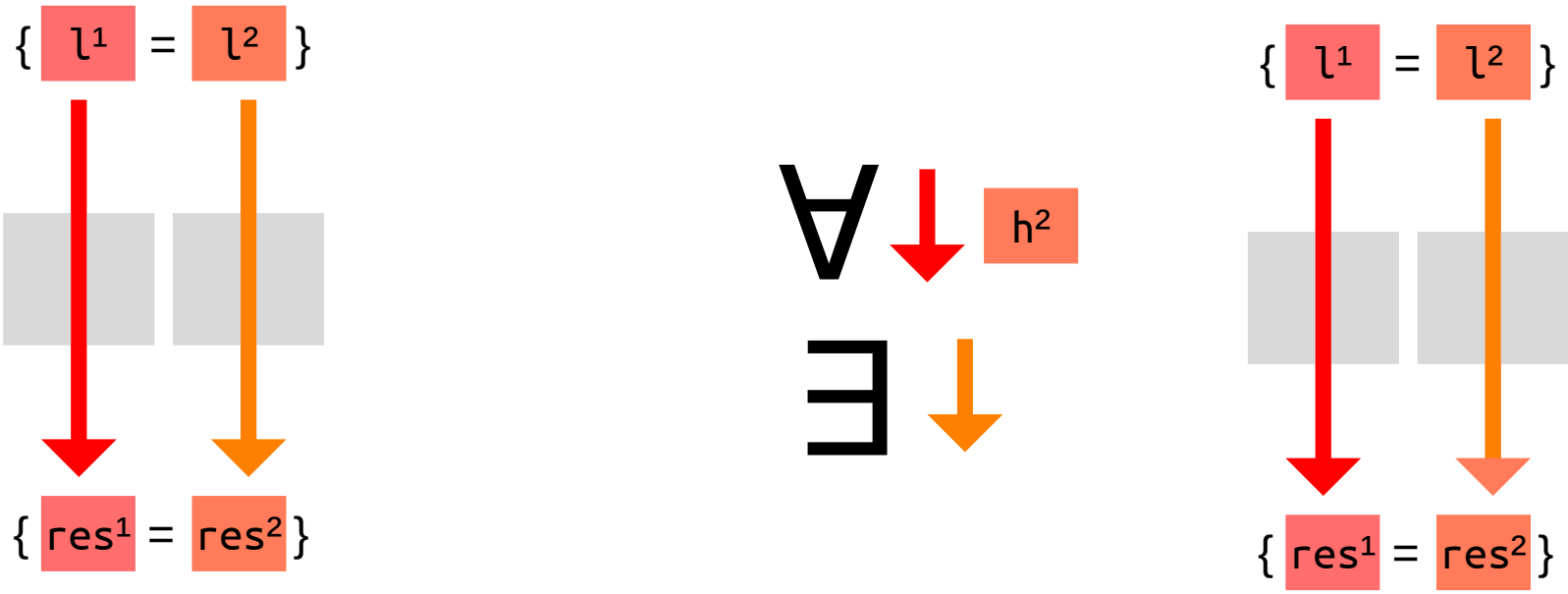
Concurrency: Noninterference Properties





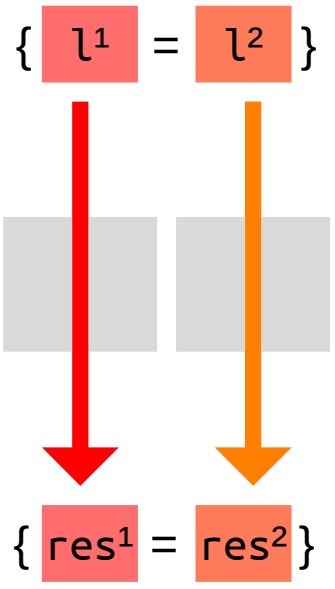
Noninterference: Too strict

Concurrency: Noninterference Properties

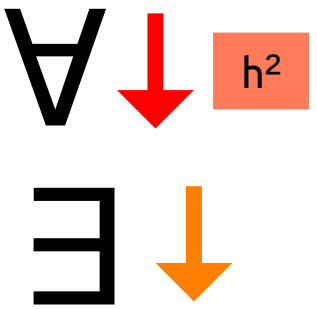


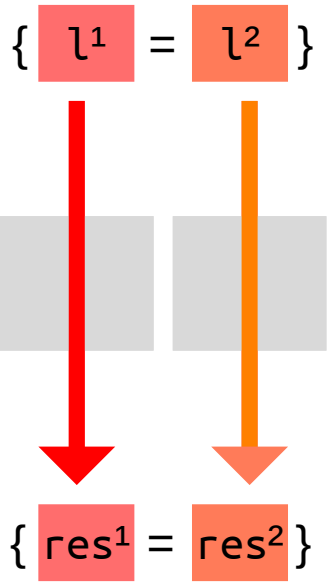
Noninterference: Too strict

Concurrency: Noninterference Properties



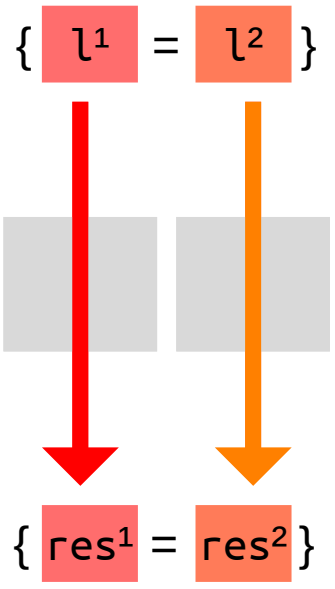
Noninterference: Too strict



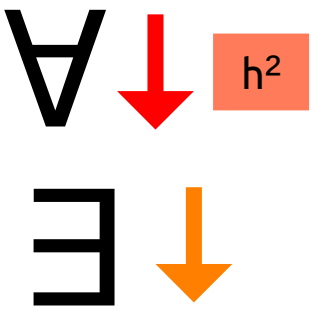


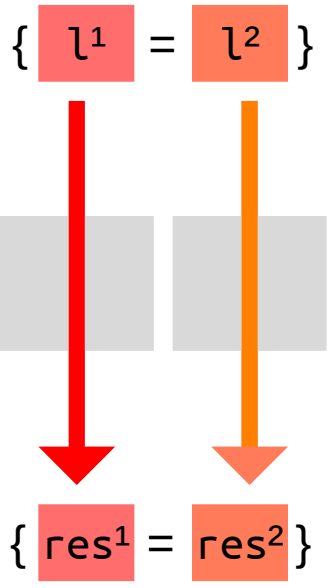
Possibilistic Noninterference

Concurrency: Noninterference Properties



Noninterference: Too strict





Possibilistic Noninterference
(Probabilistic Noninterference in the paper)

Concurrency: Existing Encodings

Concurrency: Existing Encodings

- Prove data race freedom

Concurrency: Existing Encodings

- Prove data race freedom
  - Use concurrent separation logic / ownership / …

Concurrency: Existing Encodings

- Prove data race freedom
  - Use concurrent separation logic / ownership / …
- Encode checks into sequential IVL

Concurrency: Existing Encodings

```
s
```

```
acquire l
```

```
enc(s)
```

```
<get ownership of l's heap memory>
assume Inv(l)
```

- Prove data race freedom
  - Use concurrent separation logic / ownership / …
- Encode checks into sequential IVL

Concurrency: Existing Encodings

s
acquire l
release l

enc(s)
<get ownership of l's heap memory> **assume** Inv(l)
assert Inv(l) <give up ownership of l's heap memory>

- Prove data race freedom
  - Use concurrent separation logic / ownership / …
- Encode checks into sequential IVL

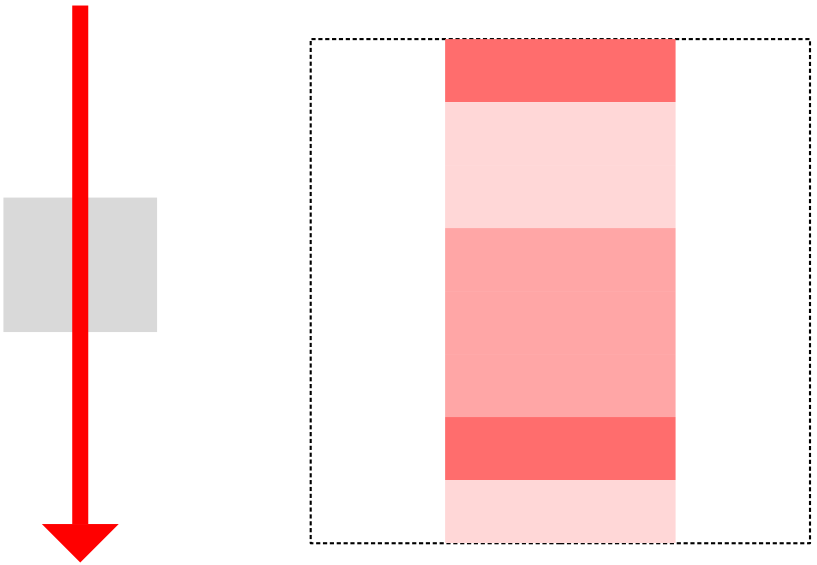
Concurrency: Existing Encodings

s
acquire l
release l

enc(s)
<get ownership of l's heap memory> **assume** Inv(l)
assert Inv(l) <give up ownership of l's heap memory>

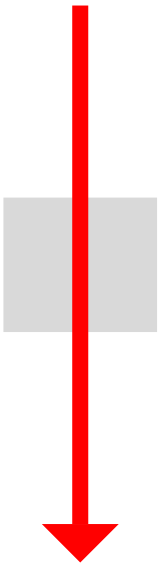
- Prove data race freedom
  - Use concurrent separation logic / ownership / …
- Encode checks into sequential IVL
- Goal: adapt this encoding

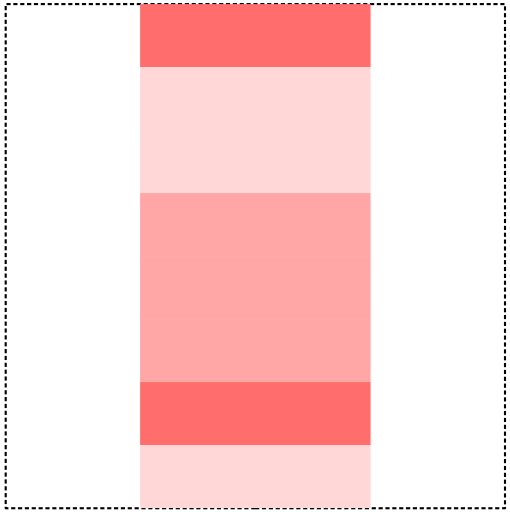
Possibilistic Noninterference: Approach



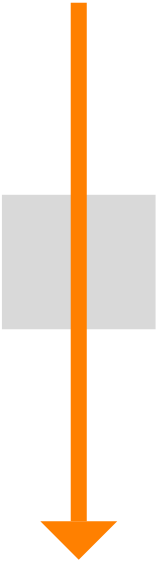
Given first trace,

Possibilistic Noninterference: Approach



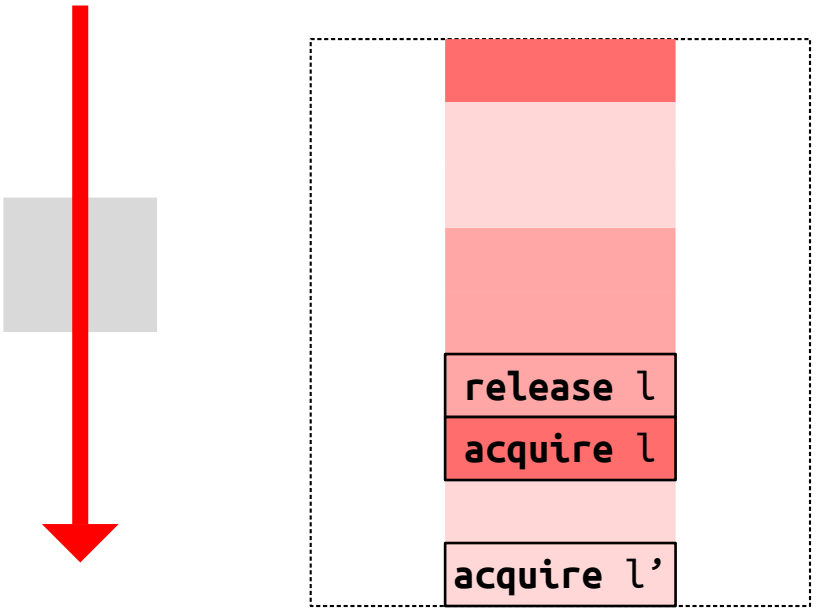


Given first trace,

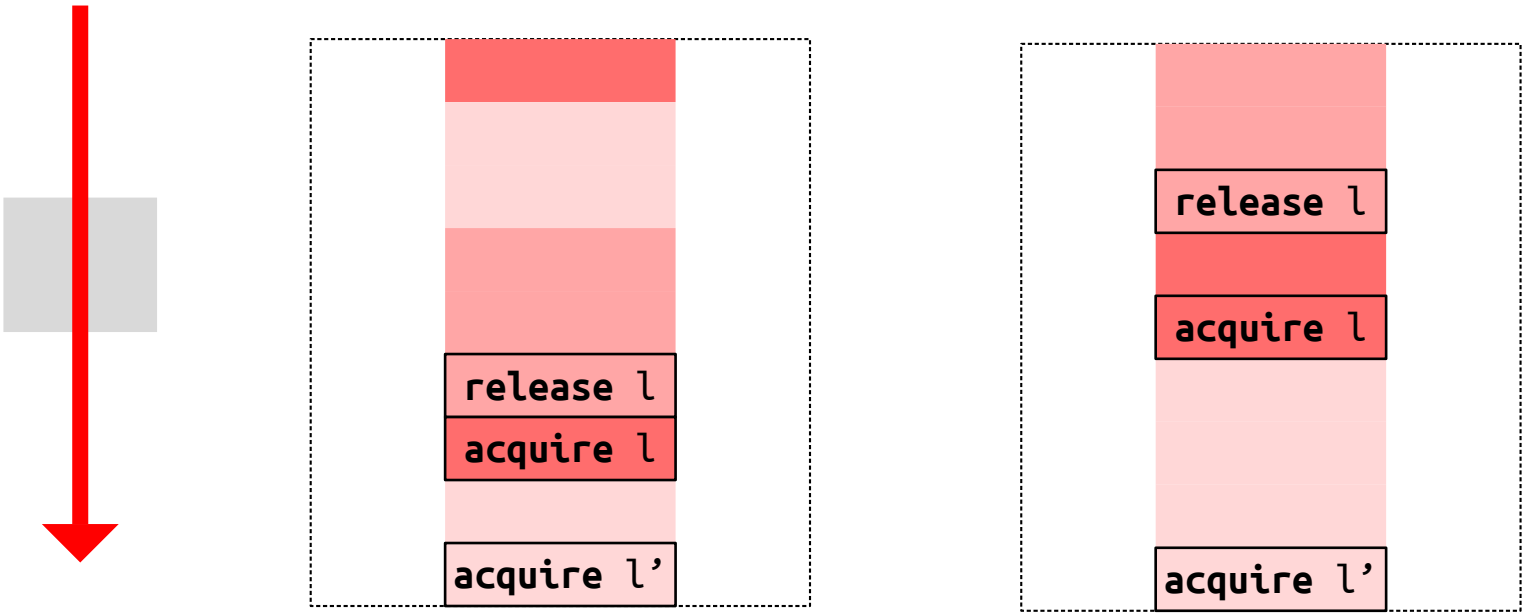


ensure second trace exists

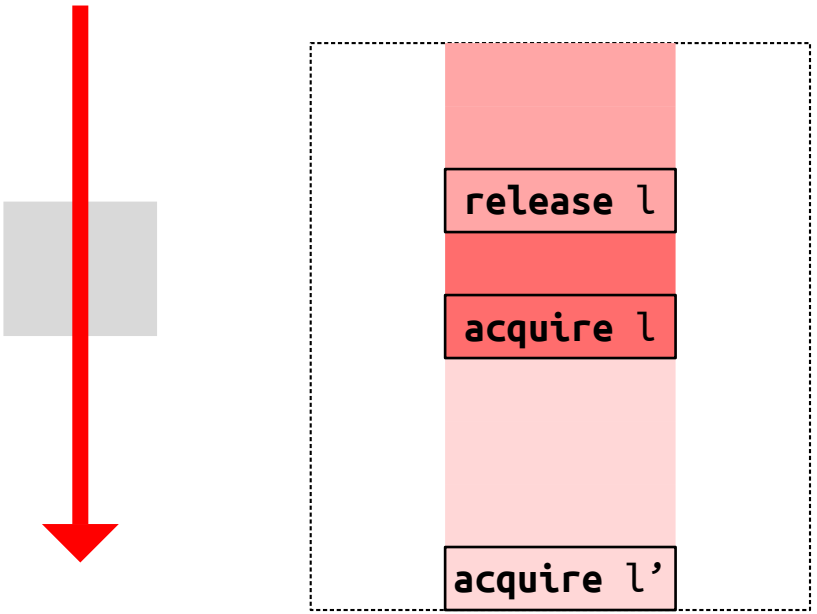
Possibilistic Noninterference: Simplification



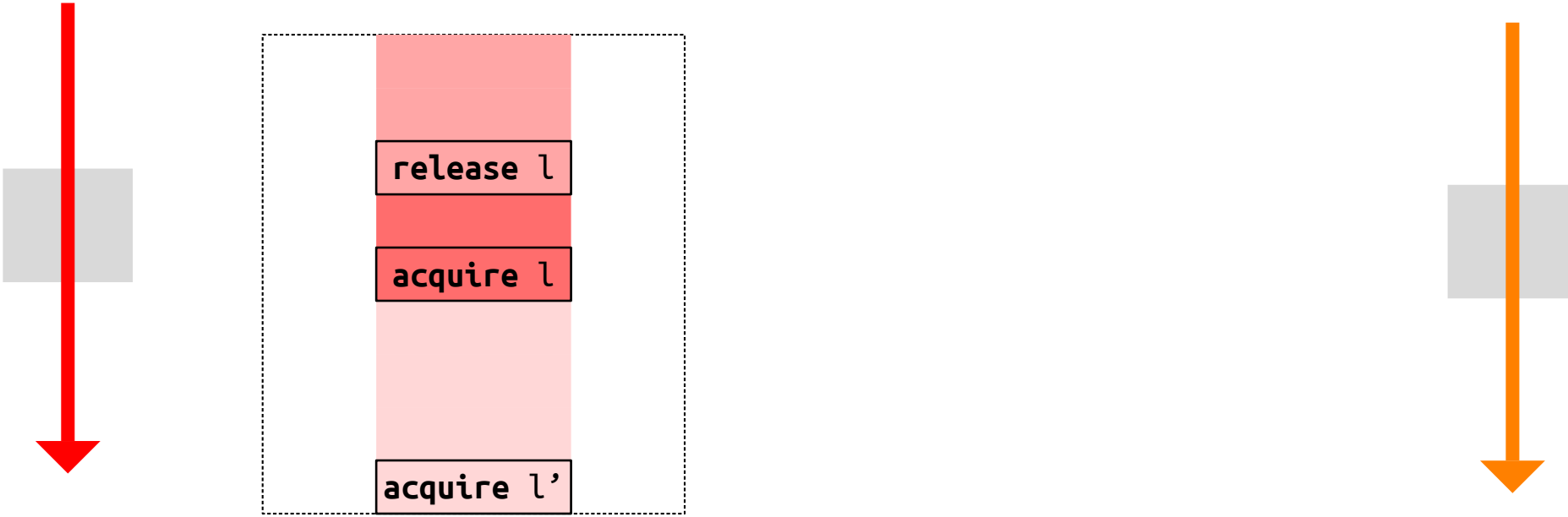
Possibilistic Noninterference: Simplification



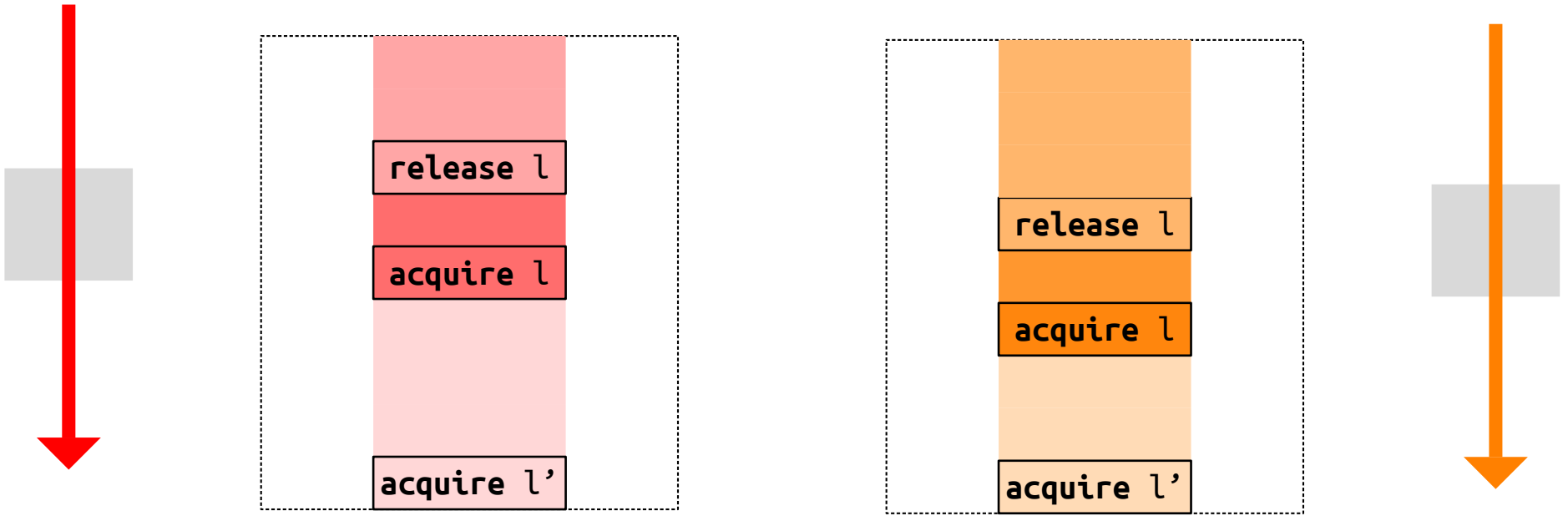
Possibilistic Noninterference: Goal



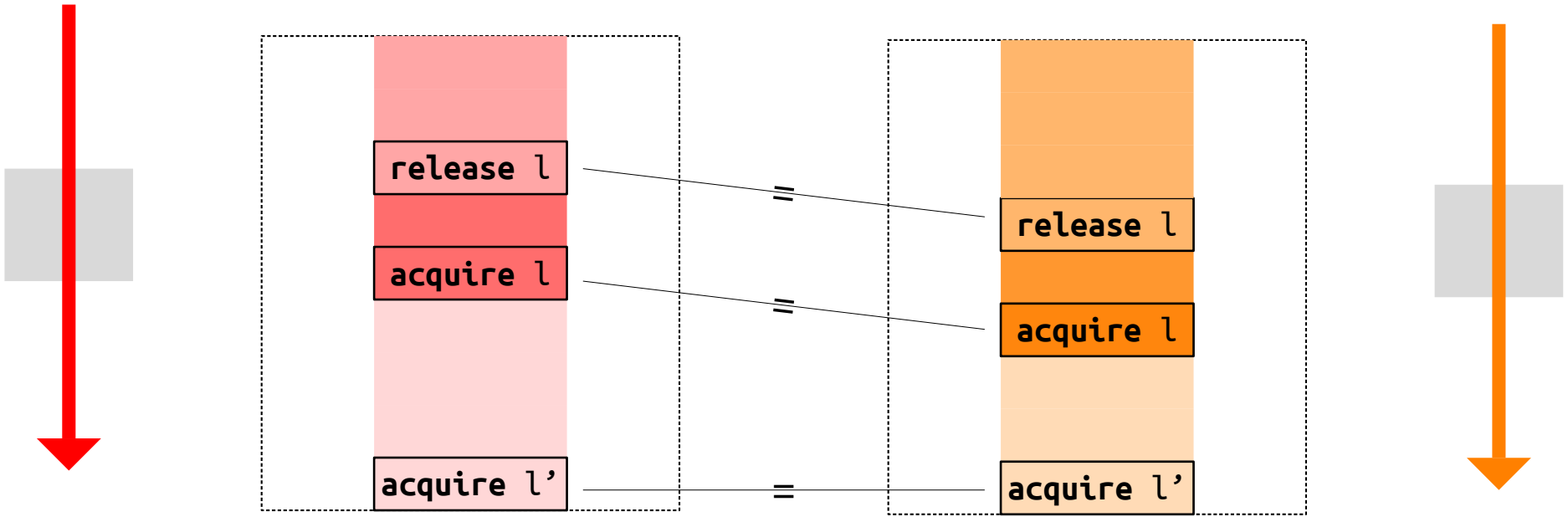
Possibilistic Noninterference: Goal



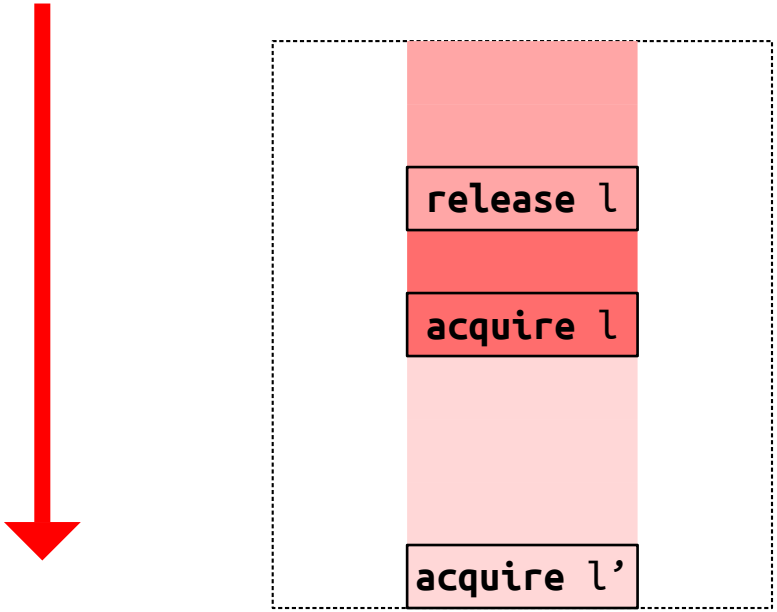
Possibilistic Noninterference: Goal



Possibilistic Noninterference: Goal

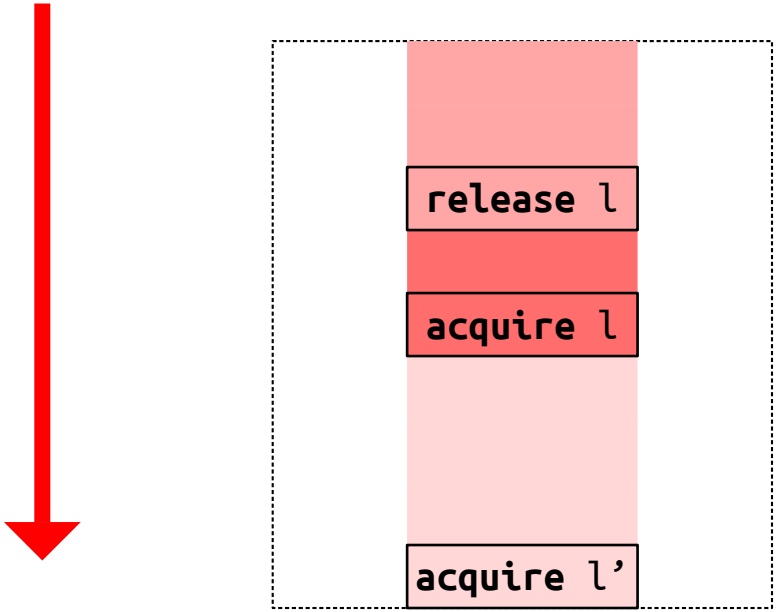


Possibilistic Noninterference: Conditions



Conditions for every lock operation:

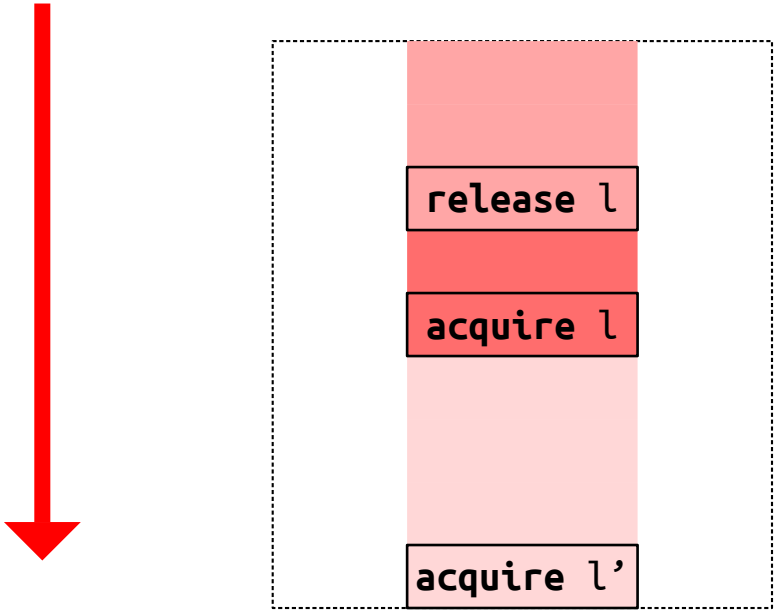
Possibilistic Noninterference: Conditions



Conditions for every lock operation:

- Not under high condition

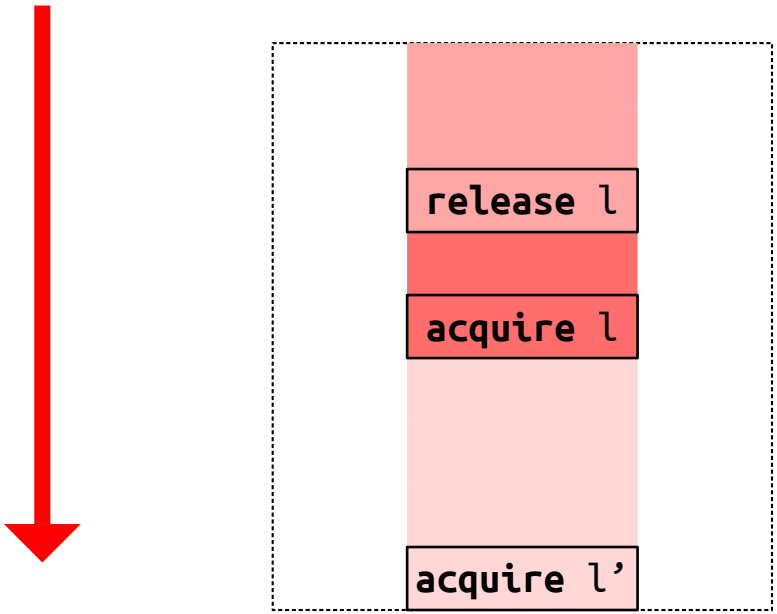
Possibilistic Noninterference: Conditions



Conditions for every lock operation:

- Not under high condition
- Code termination independent of high data

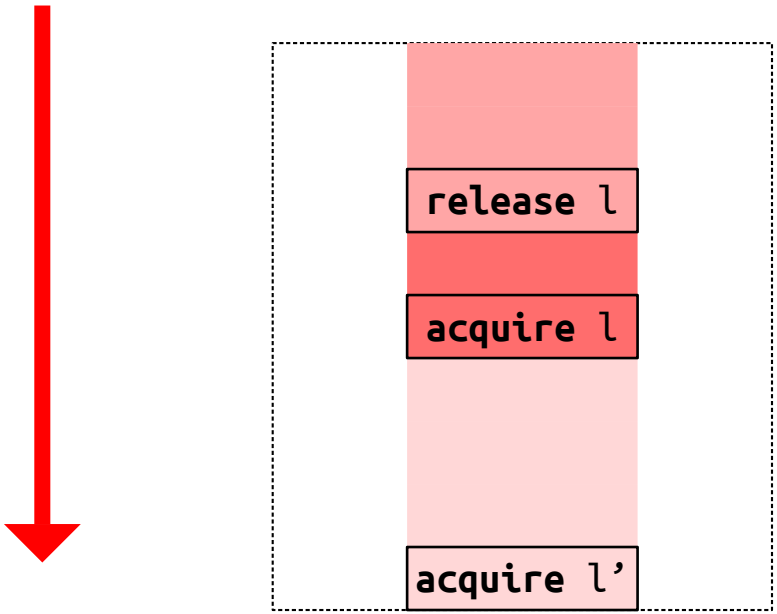
Possibilistic Noninterference: Conditions



Conditions for every lock operation:

- Not under high condition
- Code termination independent of high data
- Executed on same lock in both traces

Possibilistic Noninterference: Conditions



Conditions for every lock operation:

- Not under high condition
- Code termination independent of high data
- Executed on same lock in both traces
- Protected data identical in both traces

Possibilistic Noninterference: Encoding

- Not under high condition
- Code termination independent of high data
- Executed on same lock in both traces
- Protected data identical in both traces

s	enc'(s)
acquire l	<get ownership of l's heap memory> **assume** Inv(l)
release l	**assert** Inv(l) <give up ownership of l's heap memory>

Possibilistic Noninterference: Encoding

- Not under high condition
- Code termination independent of high data
- Executed on same lock in both traces
- Protected data identical in both traces

s	enc'(s)
acquire l	<get ownership of l's heap memory> **assume** Inv(l)
release l	**assert** Inv(l) <give up ownership of l's heap memory>

Possibilistic Noninterference: Encoding

- Not under high condition
- Code termination independent of high data
- Executed on same lock in both traces
- Protected data identical in both traces

s	enc'(s)
acquire l	**assert** <control flow is low> <get ownership of l's heap memory> **assume** Inv(l)
release l	**assert** <control flow is low> **assert** Inv(l) <give up ownership of l's heap memory>

Possibilistic Noninterference: Encoding

- Not under high condition
- Code termination independent of high data
- Executed on same lock in both traces
- Protected data identical in both traces

s	enc'(s)
acquire l	**assert** <control flow is low> <get ownership of l's heap memory> **assume** Inv(l)
release l	**assert** <control flow is low> **assert** Inv(l) <give up ownership of l's heap memory>

Possibilistic Noninterference: Encoding

- Not under high condition
- Code termination independent of high data
- Executed on same lock in both traces
- Protected data identical in both traces

s	enc'(s)
acquire l	**assert** <control flow is low> <get ownership of l's heap memory> **assume** Inv(l)
release l	**assert** <control flow is low> **assert** Inv(l) <give up ownership of l's heap memory>
while e **do** s	**assert** e1 == e2 enc(**while** e **do** s; **assert** e1 == e2)

Possibilistic Noninterference: Encoding

- Not under high condition
- Code termination independent of high data
- Executed on same lock in both traces
- Protected data identical in both traces

s	enc'(s)
acquire l	**assert** <control flow is low> <get ownership of l's heap memory> **assume** Inv(l)
release l	**assert** <control flow is low> **assert** Inv(l) <give up ownership of l's heap memory>
while e **do** s	**assert** e1 == e2 enc(**while** e **do** s; **assert** e1 == e2)

Possibilistic Noninterference: Encoding

- Not under high condition
- Code termination independent of high data
- Executed on same lock in both traces
- Protected data identical in both traces

s	enc'(s)
acquire l	**assert** <control flow is low> **assert** l1 == l2 <get ownership of l's heap memory> **assume** Inv(l)
release l	**assert** <control flow is low> **assert** l1 == l2 **assert** Inv(l) <give up ownership of l's heap memory>
while e **do** s	**assert** e1 == e2 enc(**while** e **do** s; **assert** e1 == e2)

Possibilistic Noninterference: Encoding

- Not under high condition
- Code termination independent of high data
- Executed on same lock in both traces
- Protected data identical in both traces

s	enc'(s)
acquire l	**assert** <control flow is low> **assert** l1 == l2 <get ownership of l's heap memory> **assume** Inv(l)
release l	**assert** <control flow is low> **assert** l1 == l2 **assert** Inv(l) <give up ownership of l's heap memory>
while e **do** s	**assert** e1 == e2 enc(**while** e **do** s; **assert** e1 == e2)

Possibilistic Noninterference: Encoding

- Not under high condition
- Code termination independent of high data
- Executed on same lock in both traces
- Protected data identical in both traces

s	enc'(s)
acquire l	**assert** <control flow is low> **assert** l1 == l2 <get ownership of l's heap memory> **assume** Inv(l1, l2)
release l	**assert** <control flow is low> **assert** l1 == l2 **assert** Inv(l1, l2) <give up ownership of l's heap memory>
while e **do** s	**assert** e1 == e2 enc(**while** e **do** s; **assert** e1 == e2)

Possibilistic Noninterference: Example Violation

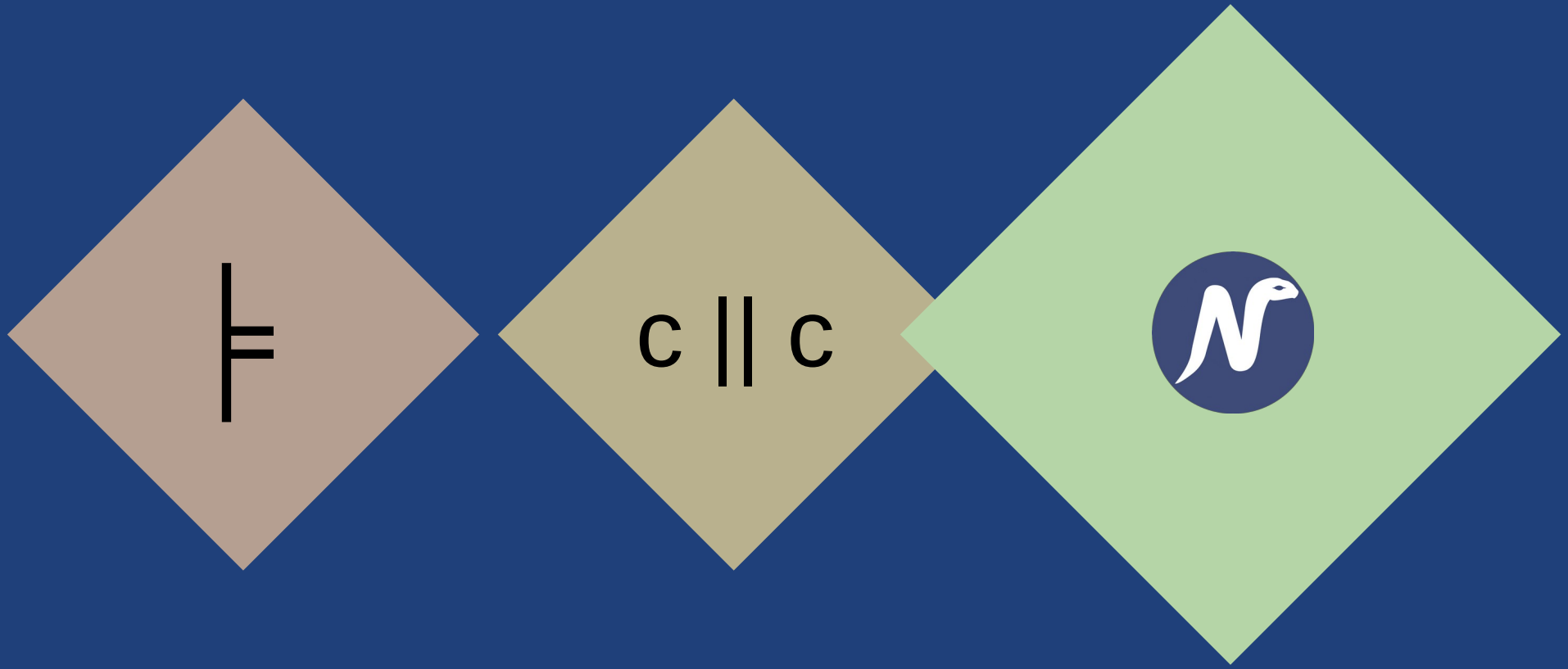
```
def main(h: bool):
  c := new Cell()
  l := new CellLock(c)
  fork thread2(c, l)
  c.val := 4
  if h:
    release l
    acquire l
  c.val := 5
  release l
```

Possibilistic Noninterference: Example Violation

```
def main(h: bool):
  c := new Cell()
  l := new CellLock(c)
  fork thread2(c, l)
  c.val := 4
  if h:
    release l
    acquire l
  c.val := 5
  release l
```

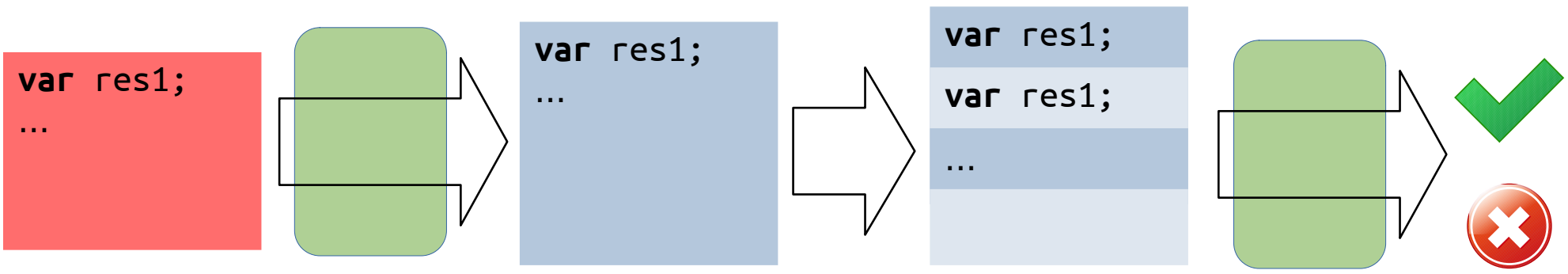
```
def thread2(c: Cell, l: CellLock):
  acquire l
  print(c.val)
  # if 4 is printed, h is true
  release l
```

Implementation and Evaluation



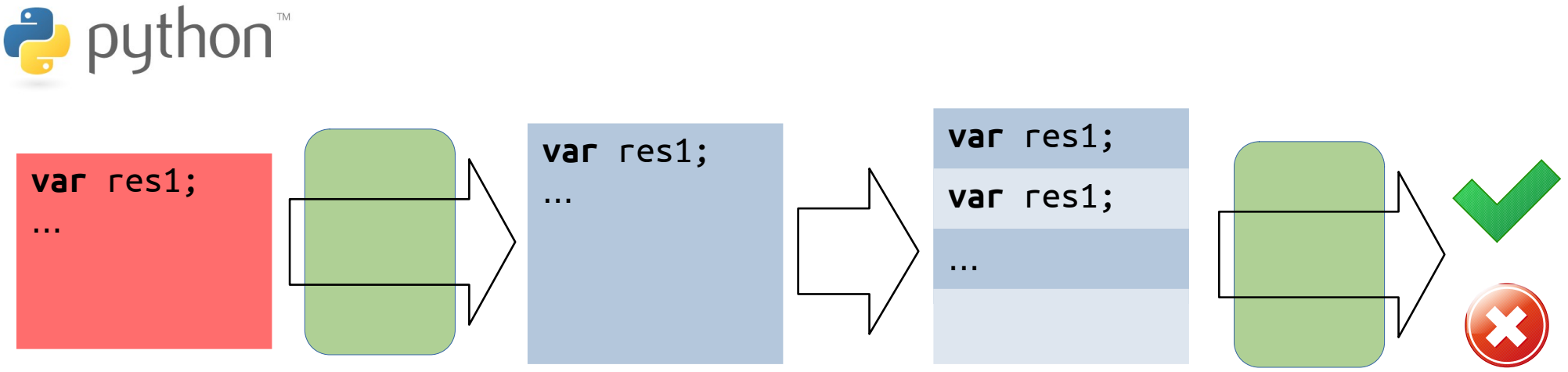
Implementation in Nagini / Viper

[Eilers & Müller, CAV 2018]
[Müller, Schwerhoff, Summers, VMCAI 2016]



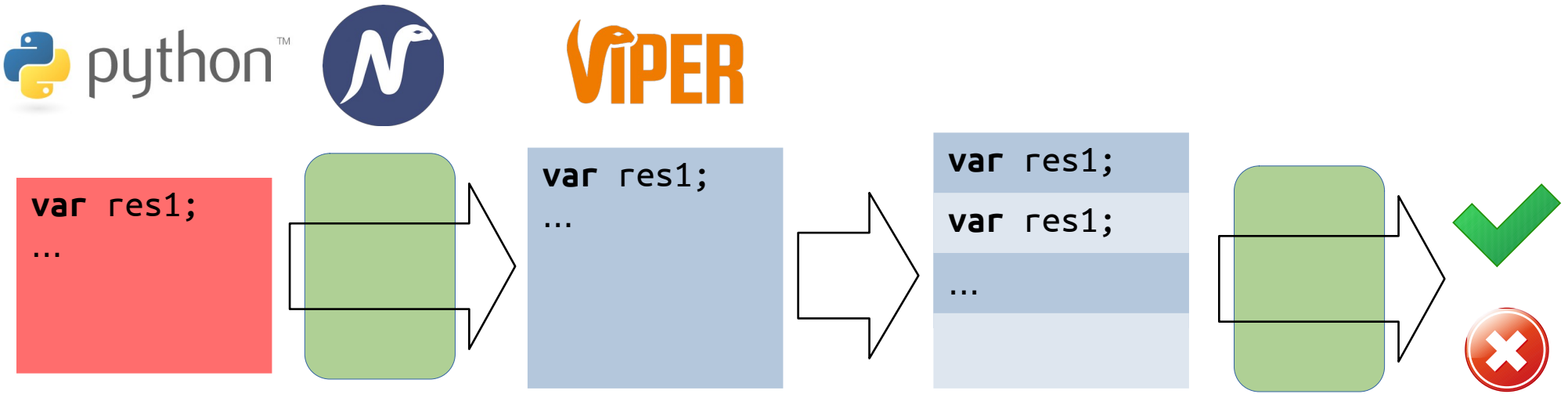
Implementation in Nagini / Viper

[Eilers & Müller, CAV 2018]
[Müller, Schwerhoff, Summers, VMCAI 2016]



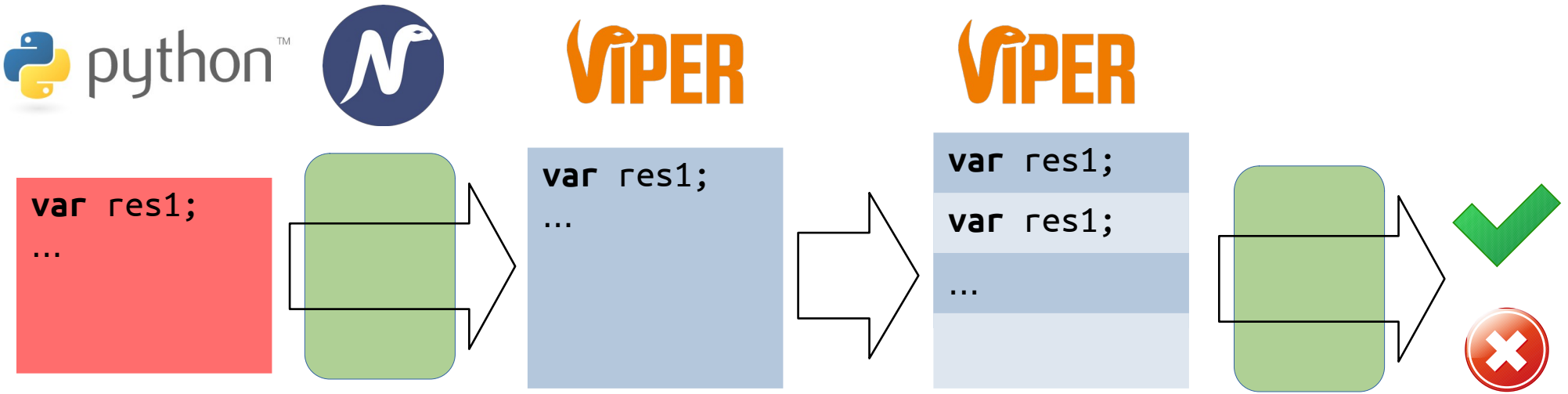
Implementation in Nagini / Viper

[Eilers & Müller, CAV 2018]
[Müller, Schwerhoff, Summers, VMCAI 2016]



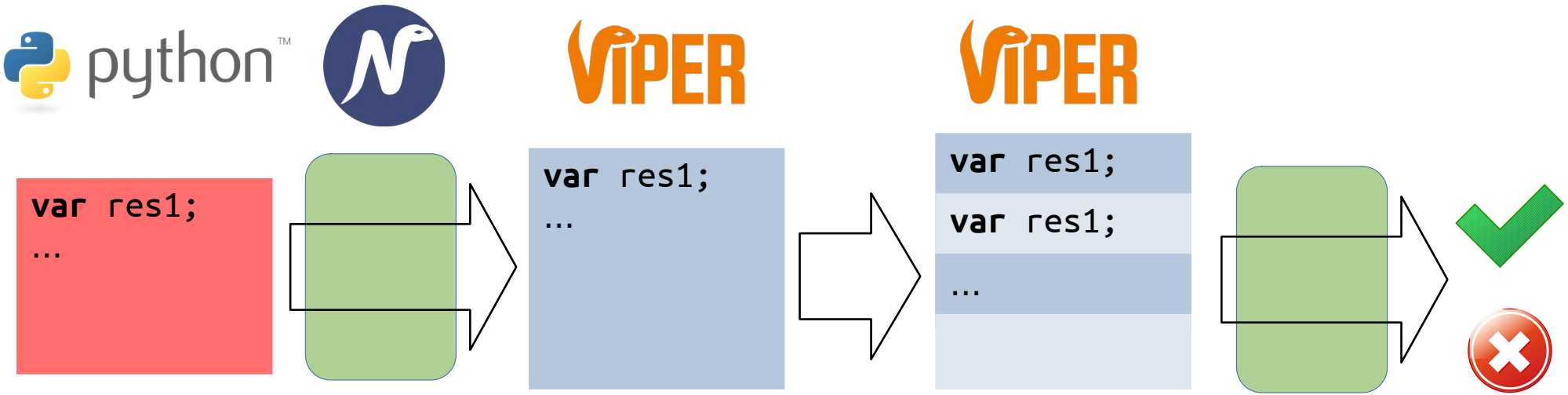
Implementation in Nagini / Viper

[Eilers & Müller, CAV 2018]
[Müller, Schwerhoff, Summers, VMCAI 2016]



Implementation in Nagini / Viper

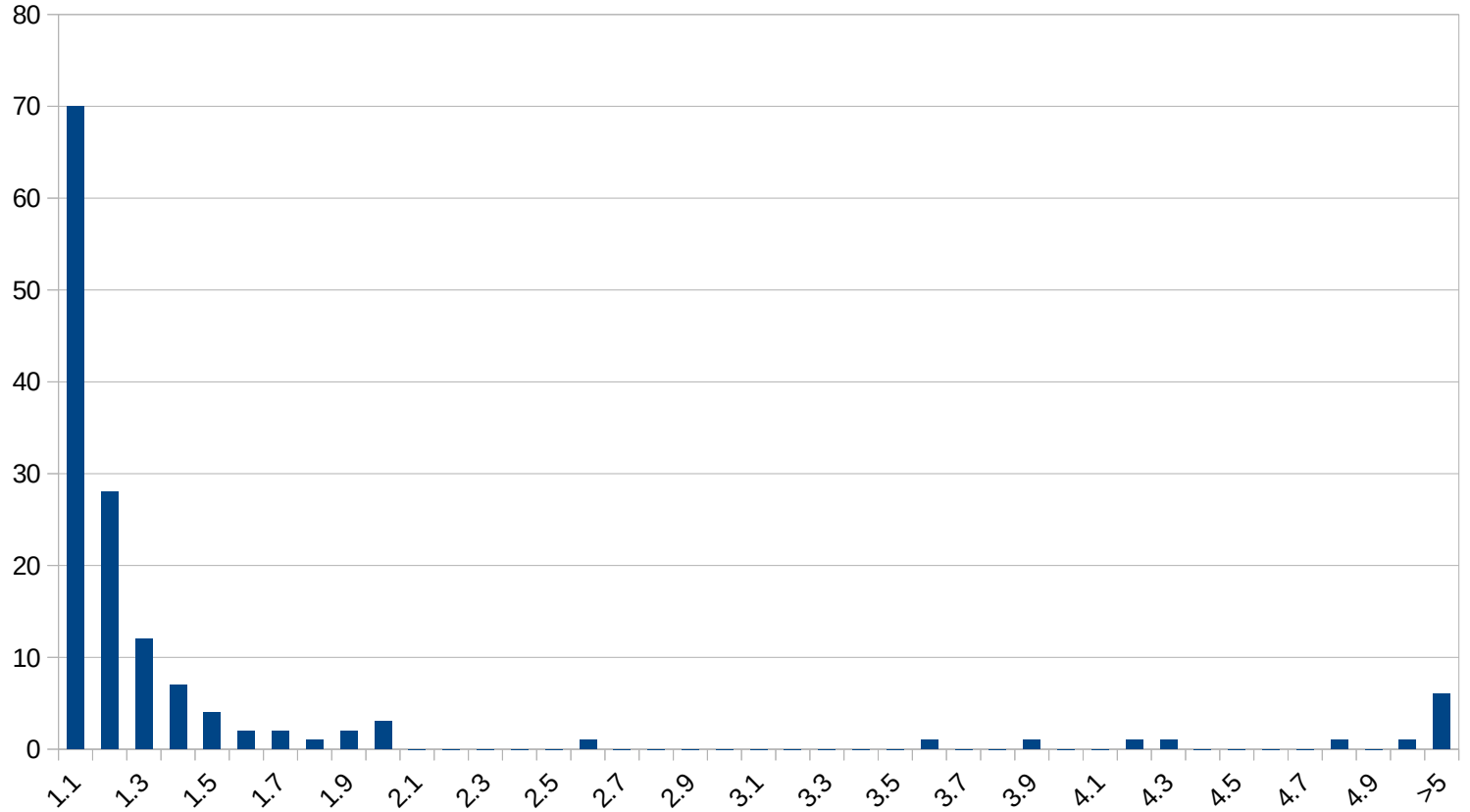
[Eilers & Müller, CAV 2018]
[Müller, Schwerhoff, Summers, VMCAI 2016]



Evaluation: Performance Overhead

- Slowdown introduced by product construction over test suite (144 tests)

Evaluation: Performance Overhead



- Slowdown introduced by product construction over test suite (144 tests)

Evaluation: Expressiveness

Evaluation: Expressiveness

- 27 sequential examples from the literature
  - Declassification, termination-sensitive noninterference, value-dependent sensitivity

Evaluation: Expressiveness

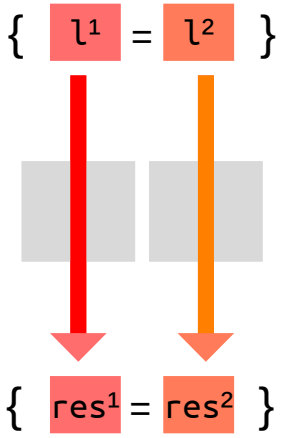
- 27 sequential examples from the literature
  - Declassification, termination-sensitive noninterference, value-dependent sensitivity

- Comparison with SecC [Ernst & Murray, CAV 2019]
- 5 examples from SecC test suite
  - Includes CDDC case study

Evaluation: Expressiveness

- 27 sequential examples from the literature
  - Declassification, termination-sensitive noninterference, value-dependent sensitivity

- Comparison with SecC [Ernst & Murray, CAV 2019]
- 5 examples from SecC test suite
  - Includes CDDC case study
- Nagini matches expressiveness of SecC
- SecC substantially faster
- Substantially lower development effort for extended Nagini version
  - Reused entire Python encoding and backend
  - Reusable product construction




```
assert
  (result1 == result2) ==>
  (result1 == result2)


def bar(a: A):
  # ensures result1 == result2
  assume result1 == result2
```