Proving Information Flow Security for Concurrent Programs

Marco Eilers
Thibault Dardinier
Peter Müller

ETH zürich

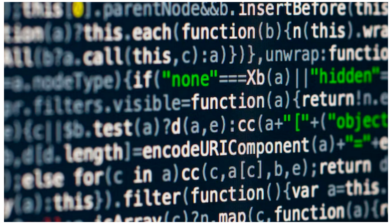
Proving Information Flow Security for Concurrent Programs

Marco Eilers
Thibault Dardinier
Peter Müller

ETH zürich

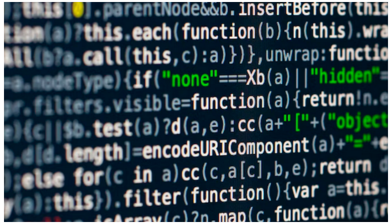
(Automatic) Program Verification

(Automatic) Program Verification



Source code
(e.g., sort algorithm)

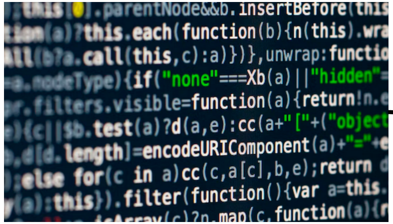
(Automatic) Program Verification



Source code
(e.g., sort algorithm)

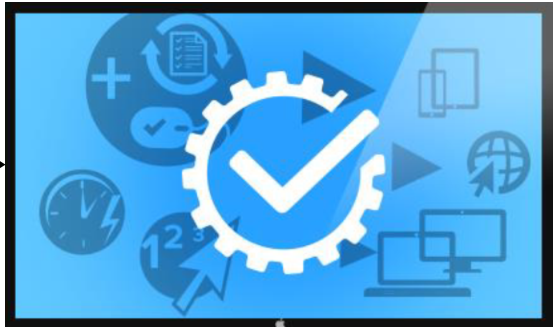
Specification
(e.g., the output is sorted)

(Automatic) Program Verification



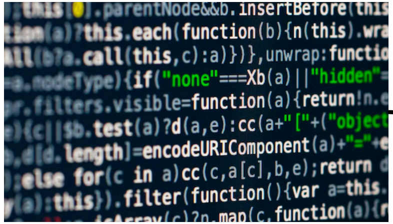
Source code
(e.g., sort algorithm)

Specification
(e.g., the output is sorted)



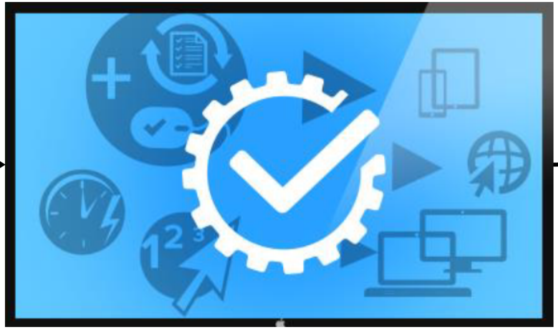
Program verifier

(Automatic) Program Verification



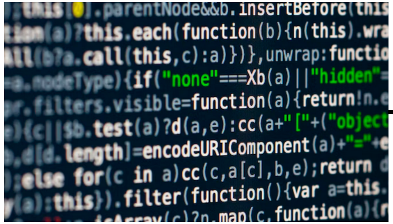
Source code
(e.g., sort algorithm)

Specification
(e.g., the output is sorted)



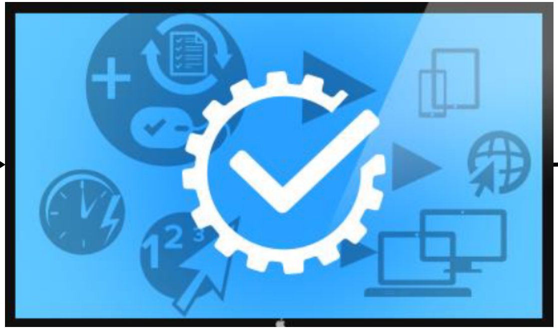
Program verifier

(Automatic) Program Verification



Source code
(e.g., sort algorithm)

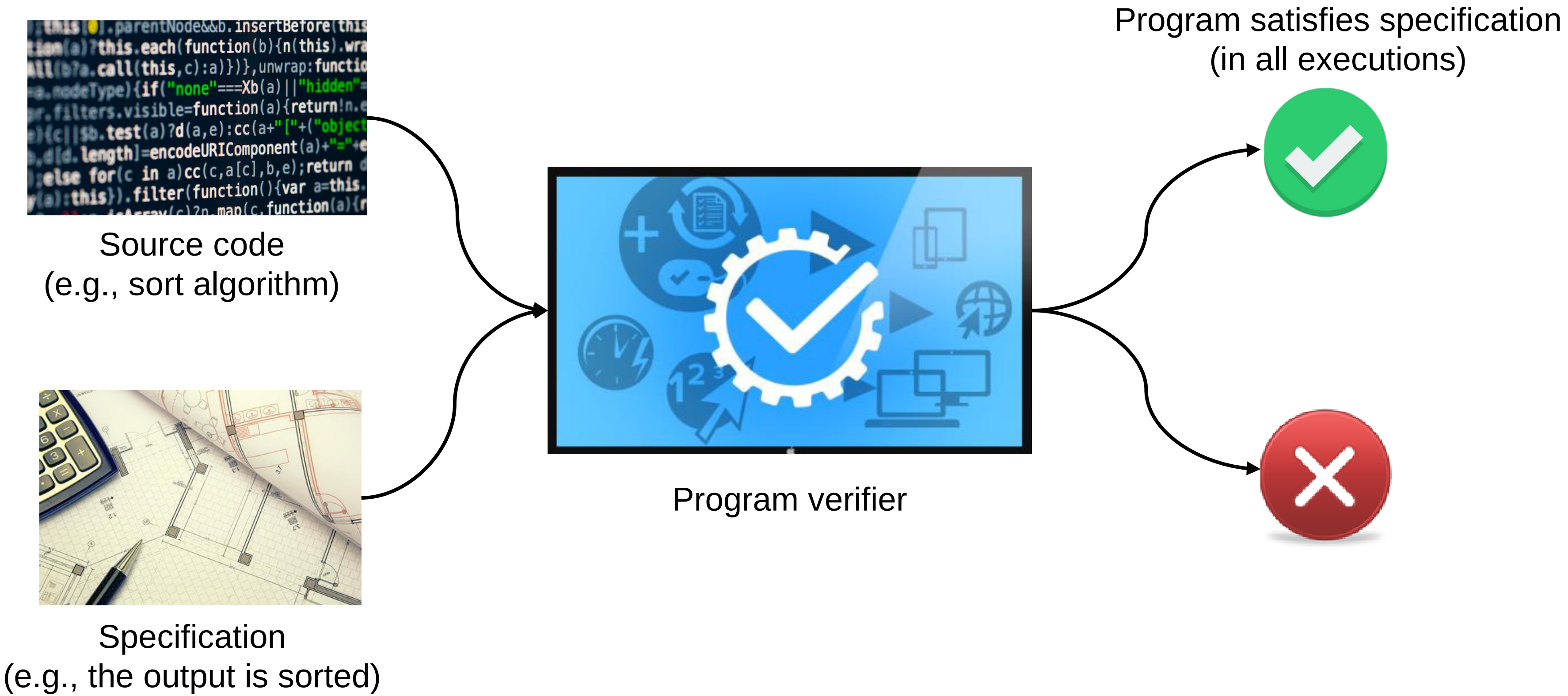
Specification
(e.g., the output is sorted)



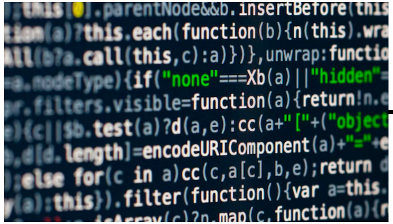
Program verifier

Program satisfies specification
(in all executions)

(Automatic) Program Verification

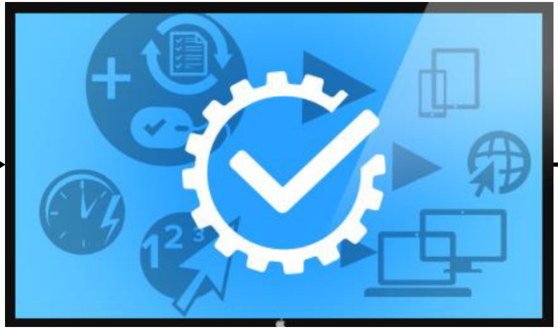


(Automatic) Program Verification



Source code
(e.g., sort algorithm)

Specification
(e.g., the output is sorted)

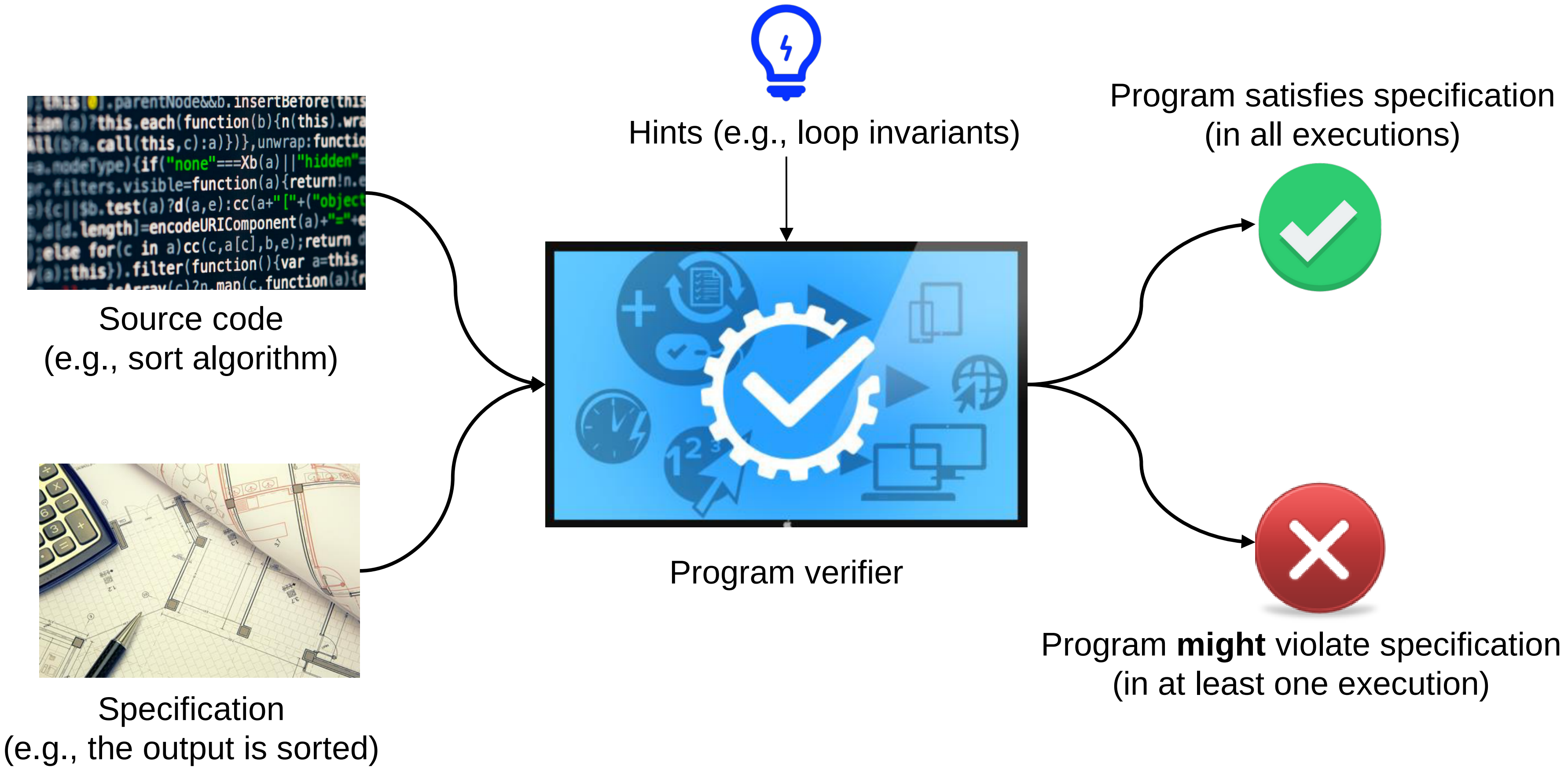


Program verifier

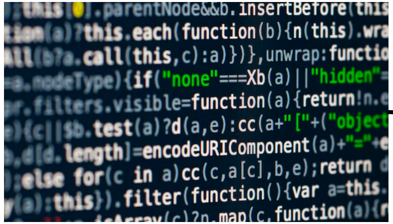
Program satisfies specification
(in all executions)

Program **might** violate specification
(in at least one execution)

(Automatic) Program Verification



(Automatic) Program Verification

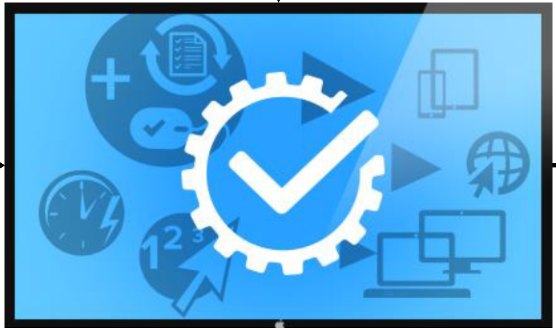


Source code
(e.g., sort algorithm)

Specification
(e.g., the output is sorted)

Can include **security** properties
(e.g., information flow security)

Hints (e.g., loop invariants)



Program verifier

Program satisfies specification
(in all executions)

Program **might** violate specification
(in at least one execution)

Secure Information Flow: Value Channel

```
def compute(h: int, l: int):
  if h > 0:
    res = 1
  else:
    res = 2
  return res
```

Secure Information Flow: Value Channel

high-sensitivity (secret)

```
def compute(h: int, l: int):
  if h > 0:
    res = 1
  else:
    res = 2
  return res
```

Secure Information Flow: Value Channel

high-sensitivity (secret)

low-sensitivity (public)

```python
def compute(h: int, l: int):
  if h > 0:
    res = 1
  else:
    res = 2
  return res
```

Secure Information Flow: Value Channel

high-sensitivity (secret)

low-sensitivity (public)

```
def compute(h: int, l: int):
  if h > 0:
    res = 1
  else:
    res = 2
  return res
```

Does *res* leak information about *h*?

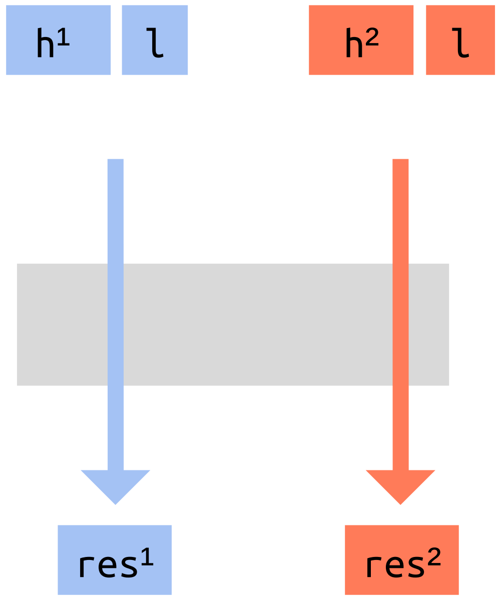
Secure Information Flow: Value Channel

high-sensitivity (secret)

low-sensitivity (public)

```python
def compute(h: int, l: int):
  if h > 0:
    res = 1
  else:
    res = 2
  return res
```

Does *res* leak information about *h*?



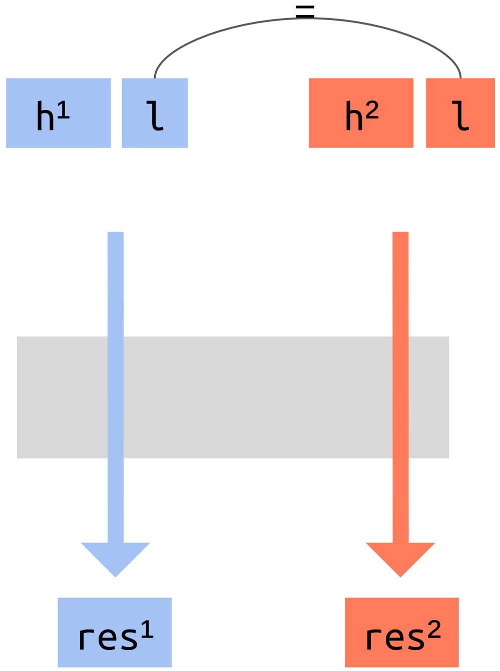
Secure Information Flow: Value Channel

high-sensitivity (secret)

low-sensitivity (public)

```
def compute(h: int, l: int):
  if h > 0:
    res = 1
  else:
    res = 2
  return res
```

Does *res* leak information about *h*?



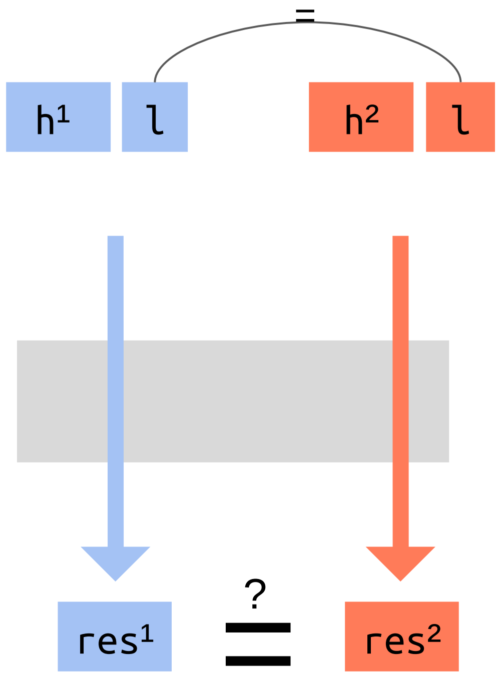
Secure Information Flow: Value Channel

high-sensitivity (secret)

low-sensitivity (public)

```
def compute(h: int, l: int):
  if h > 0:
    res = 1
  else:
    res = 2
  return res
```

Does *res* leak information about *h*?



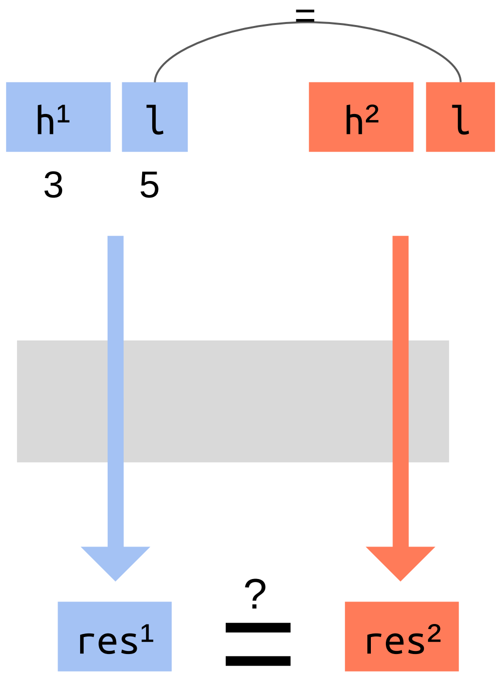
Secure Information Flow: Value Channel

high-sensitivity (secret)

low-sensitivity (public)

```
def compute(h: int, l: int):
  if h > 0:
    res = 1
  else:
    res = 2
  return res
```

Does *res* leak information about *h*?



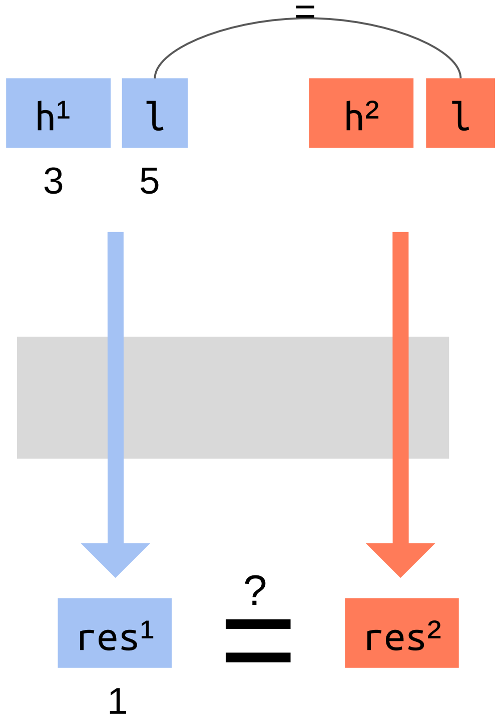
Secure Information Flow: Value Channel

high-sensitivity (secret)

low-sensitivity (public)

```
def compute(h: int, l: int):
  if h > 0:
    res = 1
  else:
    res = 2
  return res
```

Does *res* leak information about *h*?



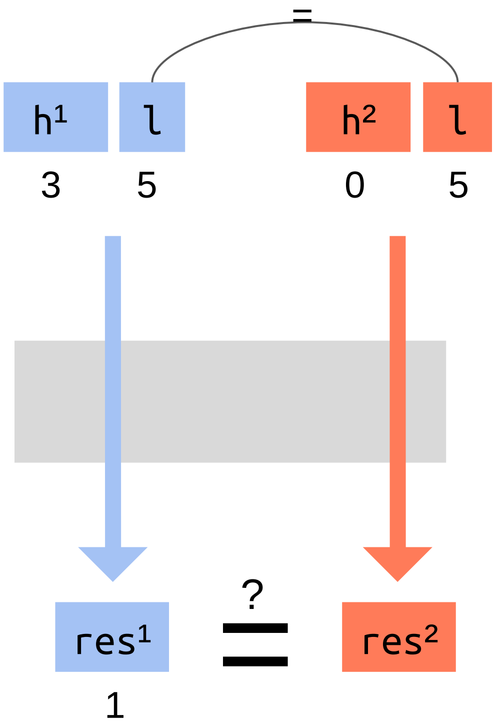
Secure Information Flow: Value Channel

high-sensitivity (secret)

low-sensitivity (public)

```
def compute(h: int, l: int):
  if h > 0:
    res = 1
  else:
    res = 2
  return res
```

Does *res* leak information about *h*?



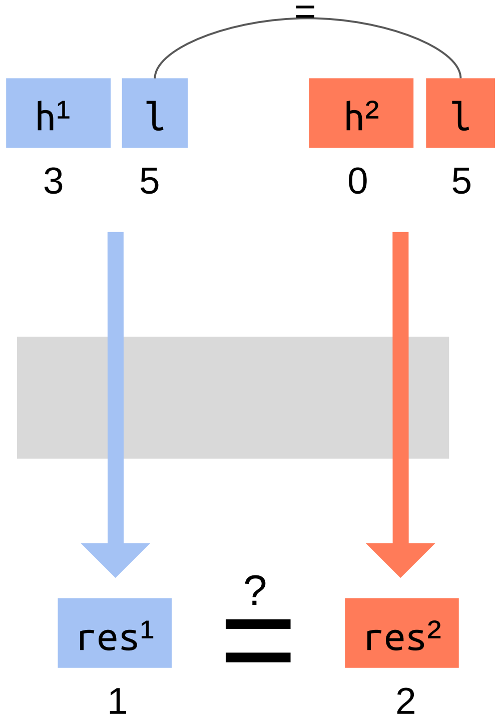
Secure Information Flow: Value Channel

high-sensitivity (secret)

low-sensitivity (public)

```
def compute(h: int, l: int):
  if h > 0:
    res = 1
  else:
    res = 2
  return res
```

Does *res* leak information about *h*?



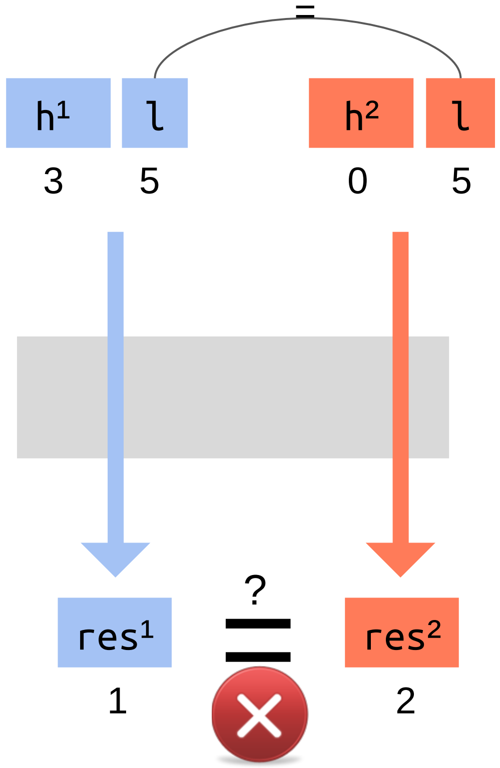
Secure Information Flow: Value Channel

high-sensitivity (secret)

low-sensitivity (public)

```python
def compute(h: int, l: int):
  if h > 0:
    res = 1
  else:
    res = 2
  return res
```

Does *res* leak information about *h*?



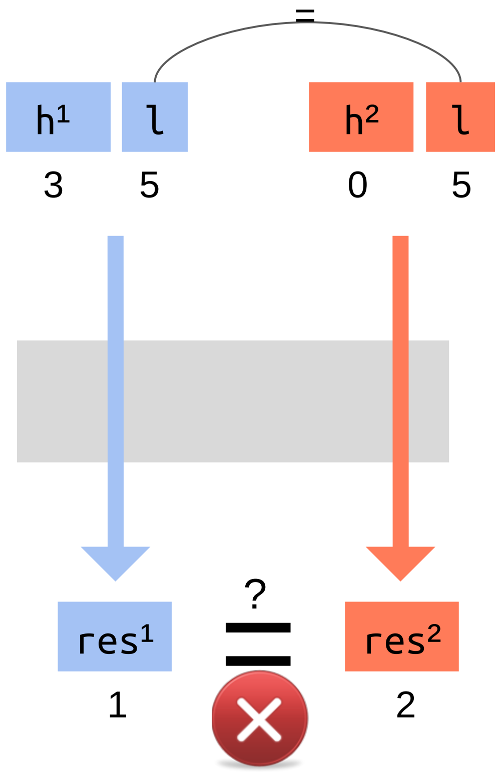
Secure Information Flow: Value Channel

high-sensitivity (secret)

low-sensitivity (public)

```python
def compute(h: int, l: int):
  if h > 0:
    res = 1
  else:
    res = 2
  return res
```

Does *res* leak information about *h*?



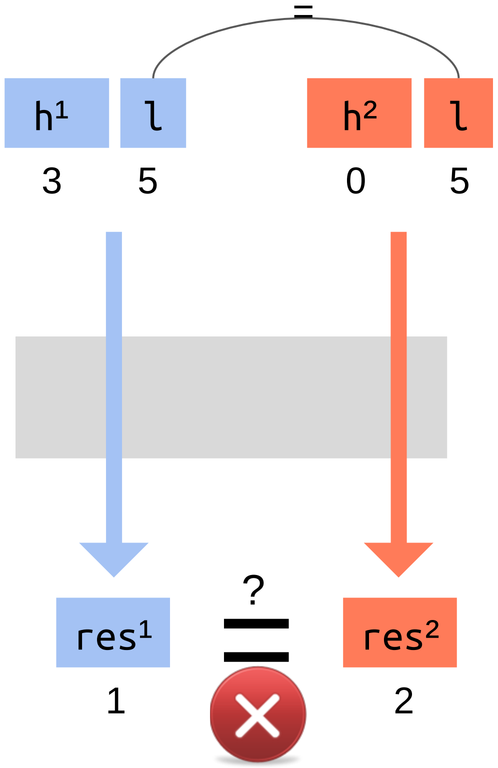
Secure Information Flow: Value Channel

high-sensitivity (secret)

low-sensitivity (public)

```
def compute(h: int, l: int):
  if h > 0:
    res = 1
  else:
    res = 2
  return res
```

Does *res* leak information about *h*?



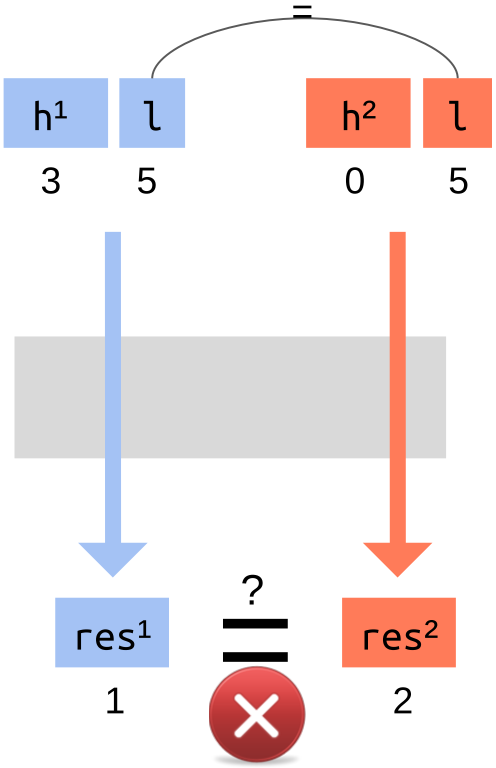
Secure Information Flow: Value Channel

high-sensitivity (secret)

low-sensitivity (public)

```
def compute(h: int, l: int):
  if h > 0:
    res = 1
  else:
    res = 2
  return res
```

Does *res* leak information about *h*?



Secure Information Flow: Timing Channel

Secure Information Flow: Timing Channel

Secure Information Flow: Timing Channel

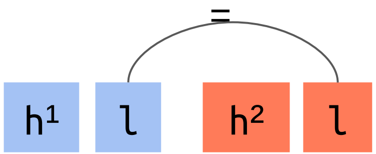
```python
def compute(h: int, l: int):
  res = 0
  if h > 0:
    res += 1
    res += 4
    res -= 7
  return 1
```

Secure Information Flow: Timing Channel

```python
def compute(h: int, l: int):
  res = 0
  if h > 0:
    res += 1
    res += 4
    res -= 7
  return 1
```

Does the execution time leak information about *h*?

Secure Information Flow: Timing Channel



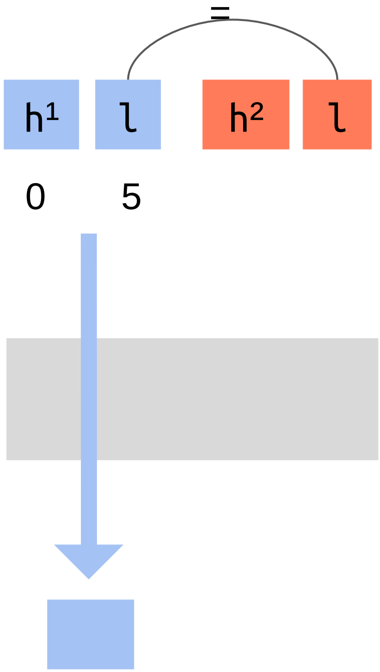

```
def compute(h: int, l: int):
  res = 0
  if h > 0:
    res += 1
    res += 4
    res -= 7
  return 1
```

Does the execution time leak information about *h*?

Secure Information Flow: Timing Channel

```python
def compute(h: int, l: int):
  res = 0
  if h > 0:
    res += 1
    res += 4
    res -= 7
  return 1
```

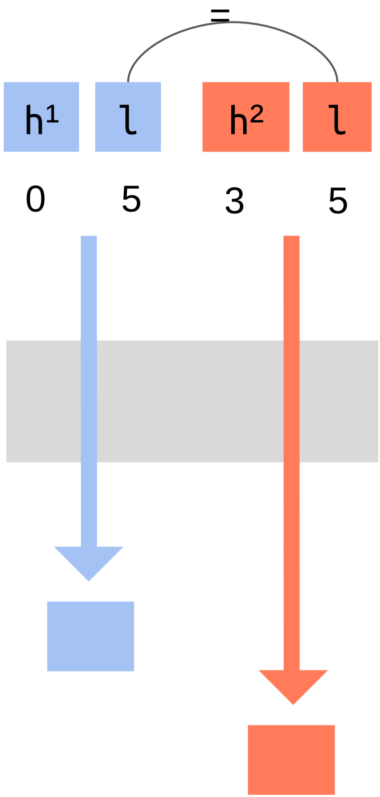
Does the execution time leak information about *h*?



Secure Information Flow: Timing Channel

```python
def compute(h: int, l: int):
  res = 0
  if h > 0:
    res += 1
    res += 4
    res -= 7
  return 1
```

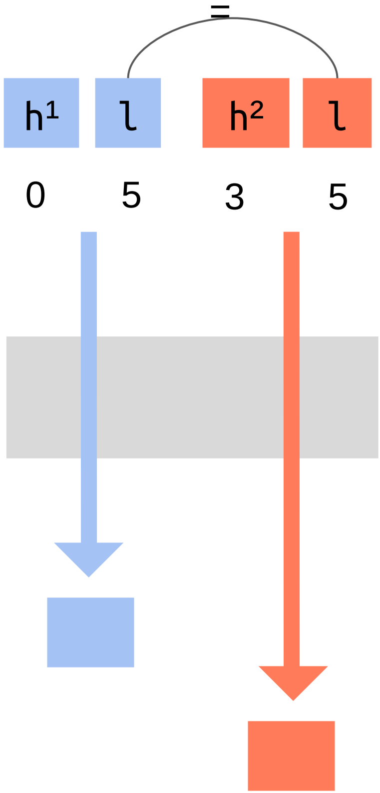
Does the execution time leak information about *h*?



Secure Information Flow: Timing Channel

```python
def compute(h: int, l: int):
  res = 0
  if h > 0:
    res += 1
    res += 4
    res -= 7
  return 1
```

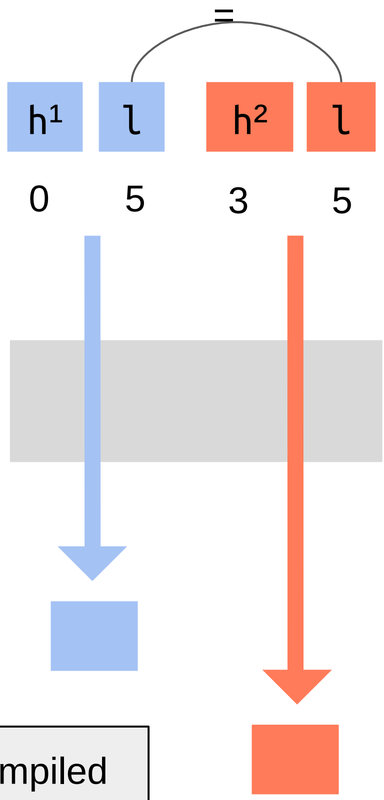
Does the execution time leak information about *h*?



Secure Information Flow: Timing Channel

```python
def compute(h: int, l: int):
  res = 0
  if h > 0:
    res += 1
    res += 4
    res -= 7
  return 1
```

Does the execution time leak information about *h*?

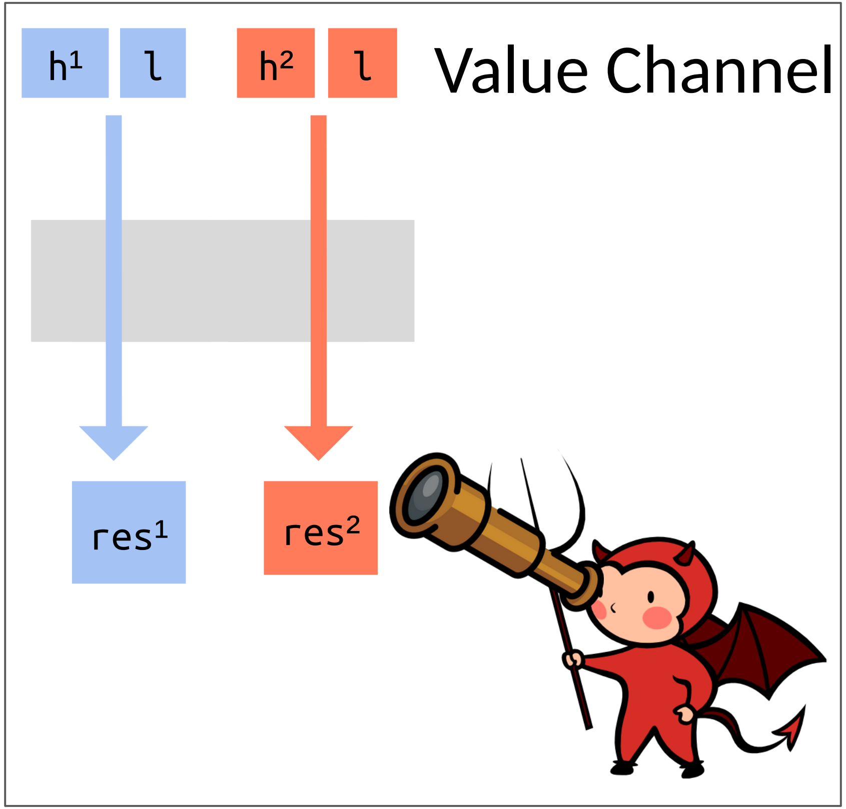



```
          section   .text
_start:   mov       rax, 1
          mov       rdi, 1
          mov       rsi, message
          syscall
          ...
```

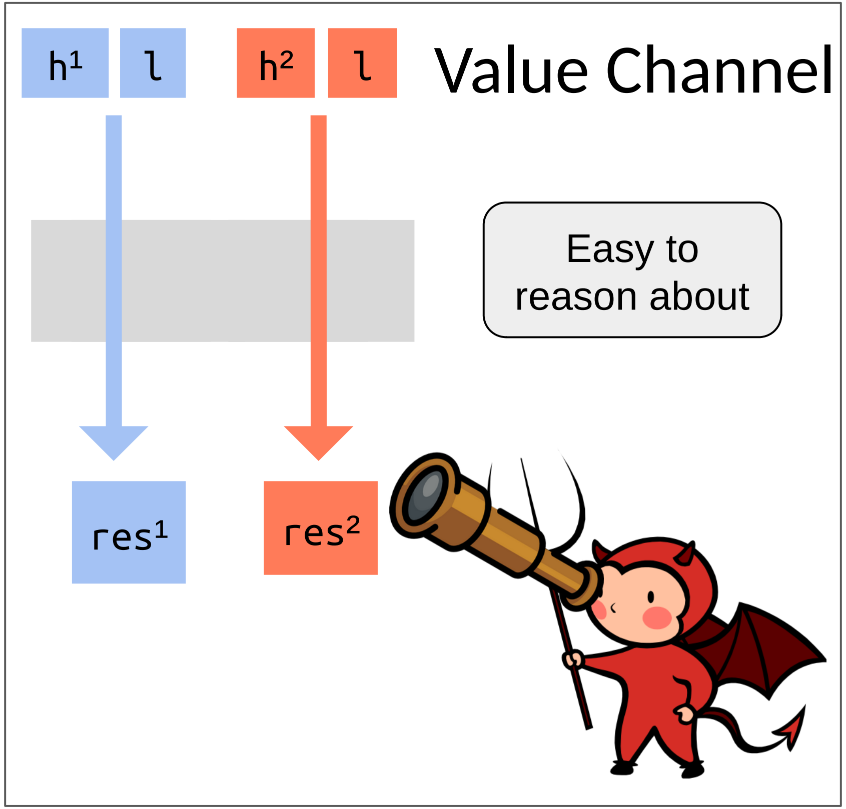
The execution time of the compiled program typically depends on values

Secure Information Flow

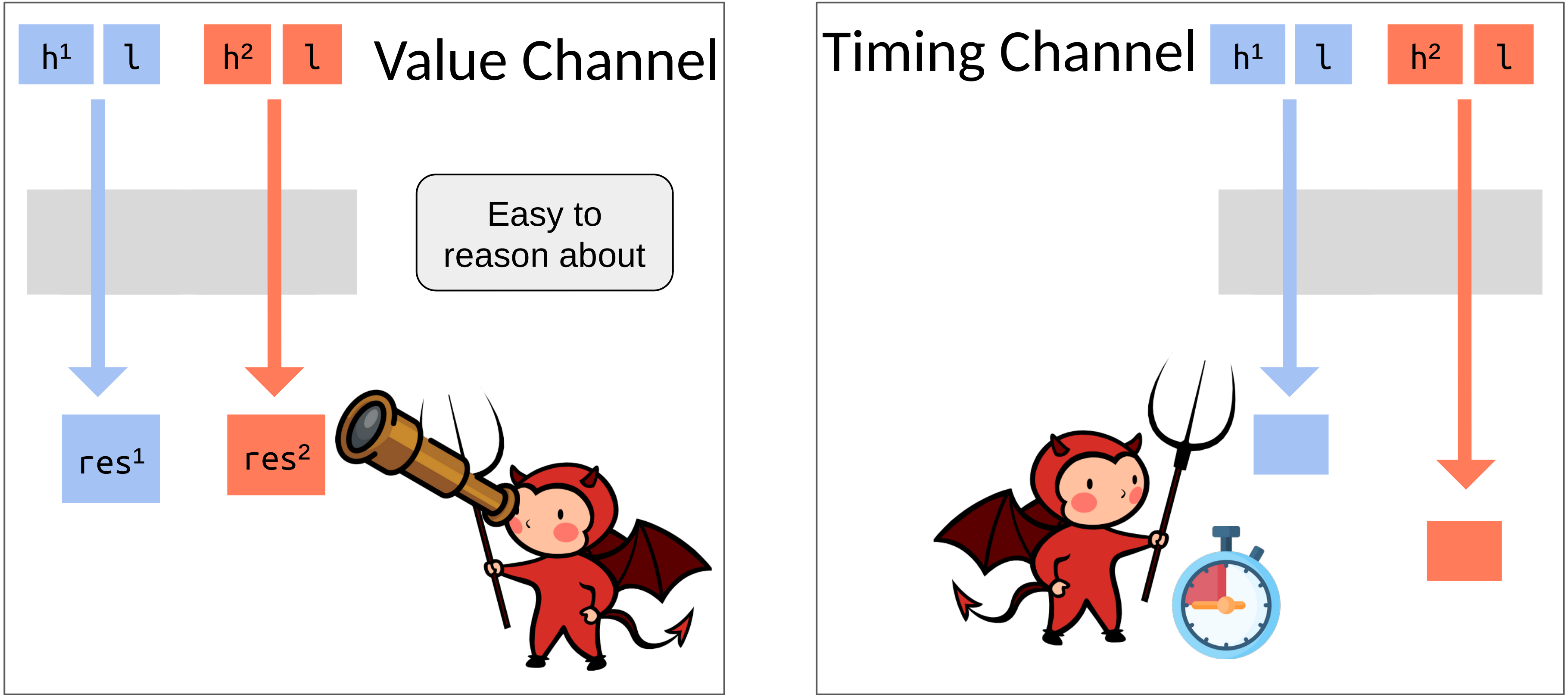
Secure Information Flow



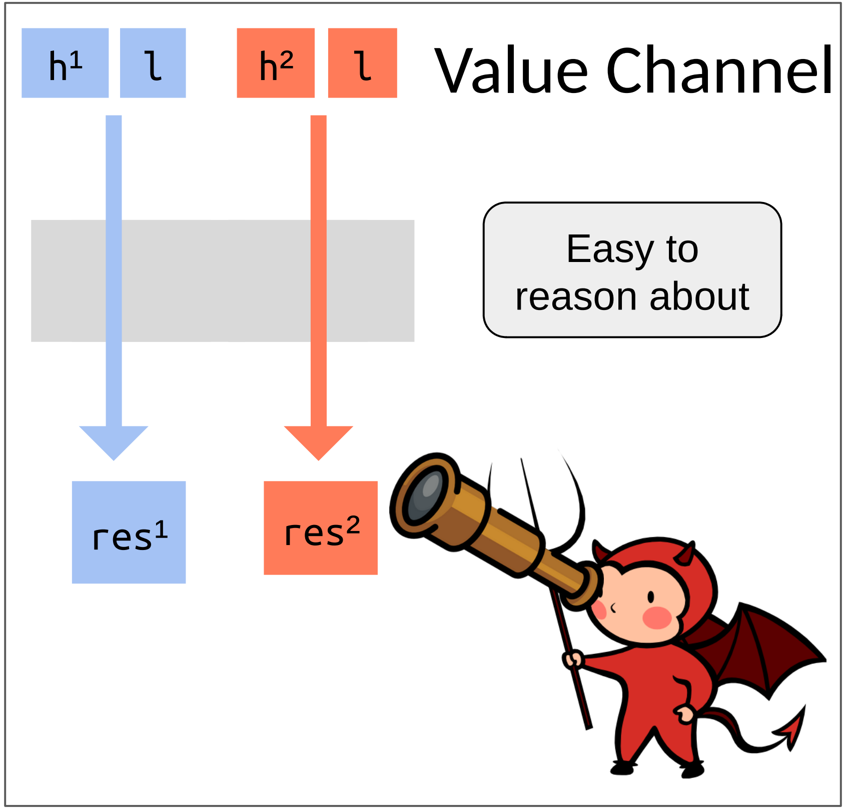
Secure Information Flow

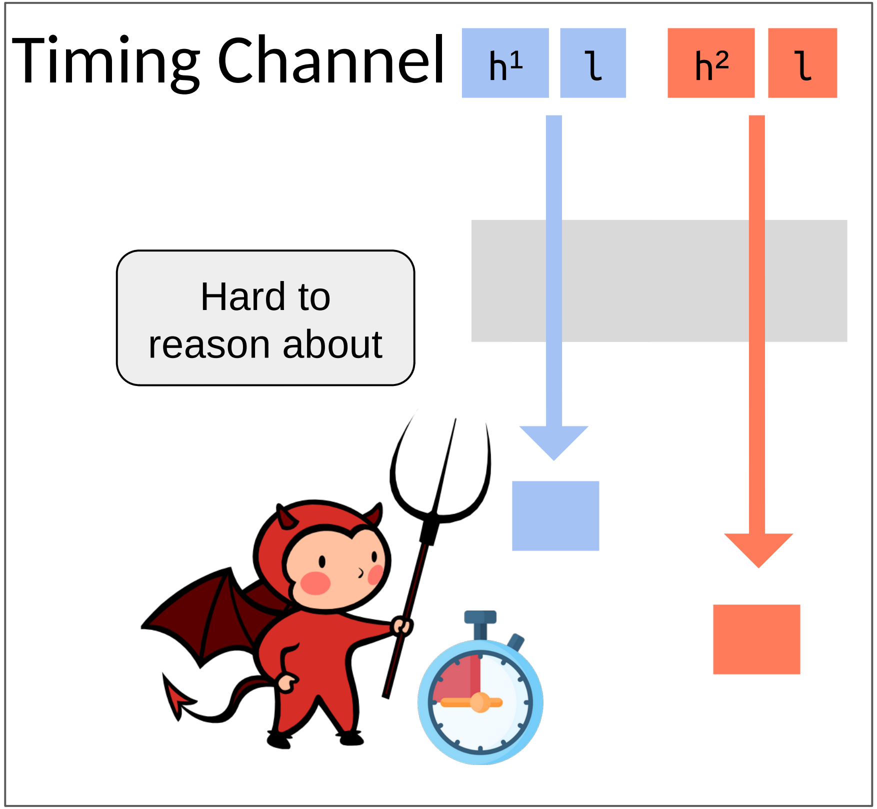


Secure Information Flow

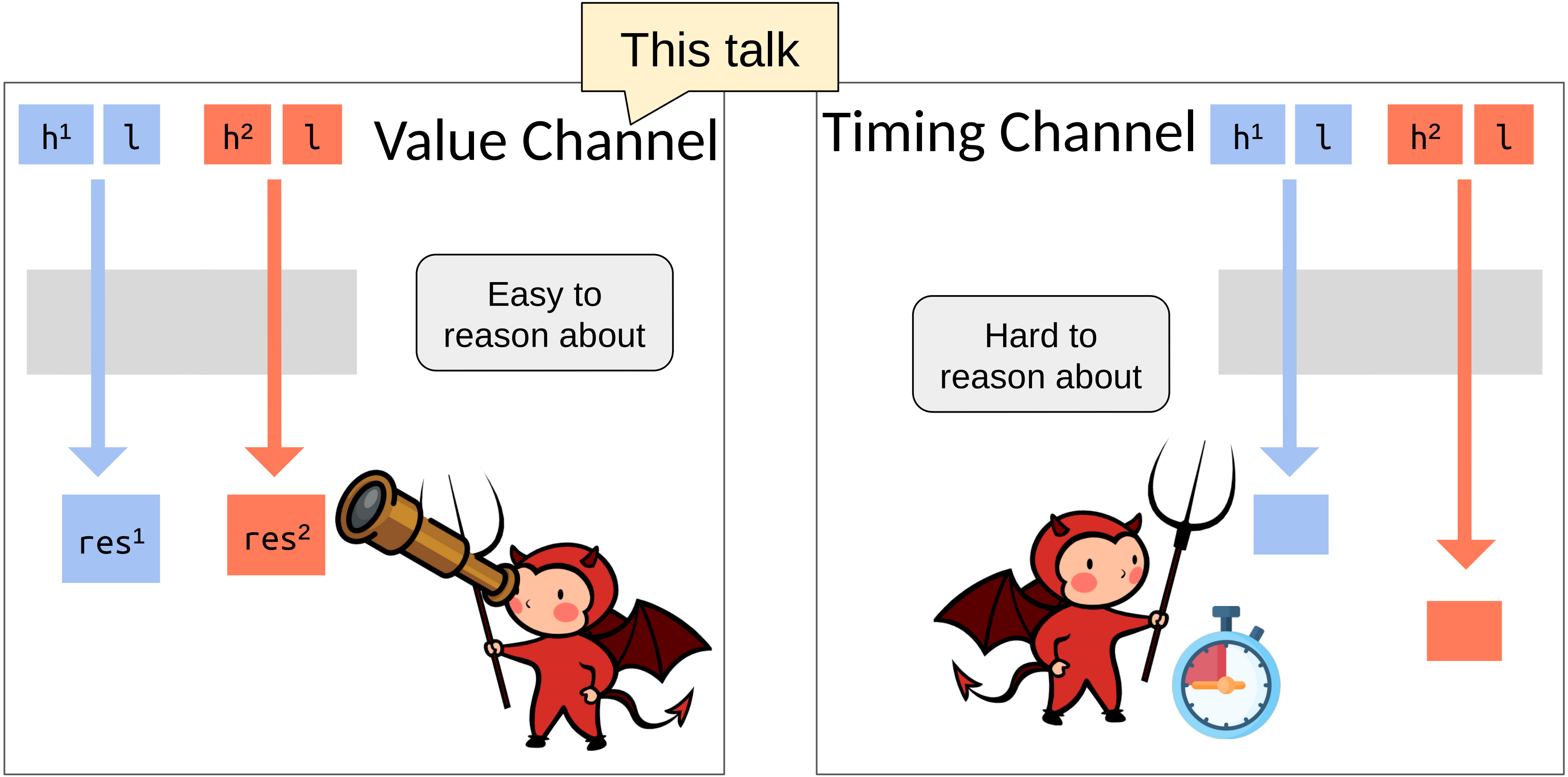


Secure Information Flow





Secure Information Flow



Attacker:
Observes **final results**,
not intermediate state or **timing**

Shared-Memory Concurrency Ruins Everything

Shared-Memory Concurrency Ruins Everything

```
while i < h:        ||  while j < 100:
  i += 1            ||    j += 1
shared = 6          ||  shared = 7

        return shared
```

Shared-Memory Concurrency Ruins Everything

Secret-dependent execution time

```
while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
shared = 6            ||   shared = 7

        return shared
```

Shared-Memory Concurrency Ruins Everything

Secret-dependent execution time

Secret-**in**dependent execution time

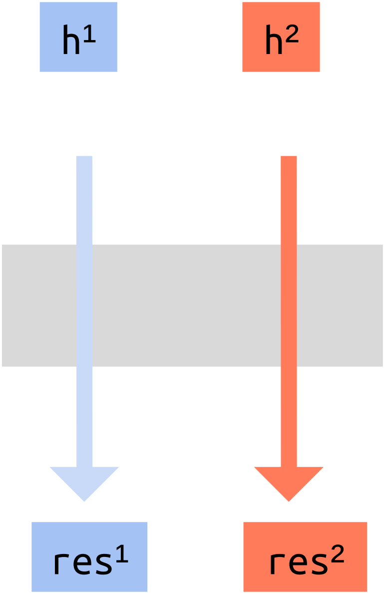
```
while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
shared = 6            ||   shared = 7

          return shared
```

Shared-Memory Concurrency Ruins Everything

Secret-dependent execution time

Secret-**in**dependent execution time

```
while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
shared = 6            ||   shared = 7

        return shared
```

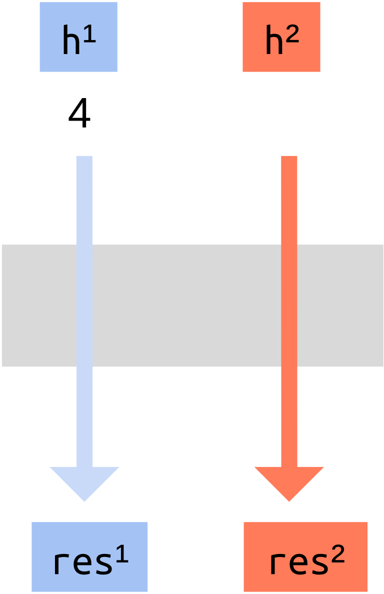


Shared-Memory Concurrency Ruins Everything

Secret-dependent execution time

Secret-**in**dependent execution time

```
while i < h:         ||   while j < 100:
  i += 1             ||     j += 1
shared = 6           ||   shared = 7

        return shared
```

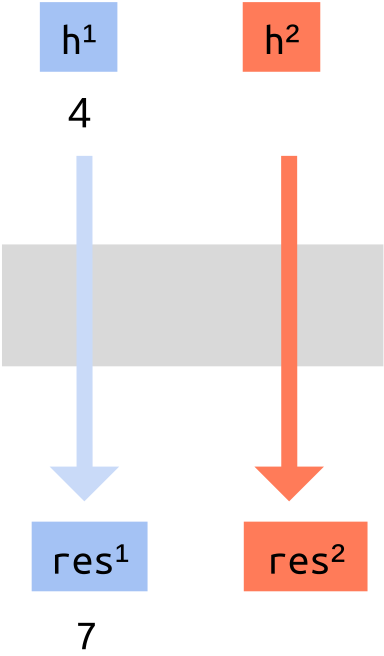


Shared-Memory Concurrency Ruins Everything

Secret-dependent execution time

Secret-**in**dependent execution time

```
while i < h:        ||   while j < 100:
  i += 1            ||     j += 1
shared = 6          ||   shared = 7

            return shared
```

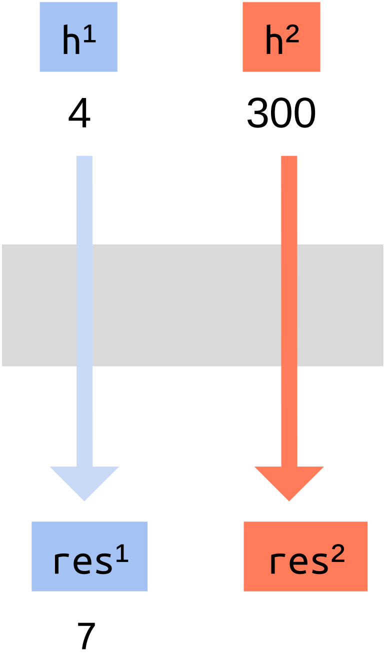


Shared-Memory Concurrency Ruins Everything

Secret-dependent execution time

Secret-**in**dependent execution time

```
while i < h:        ||   while j < 100:
  i += 1            ||     j += 1
shared = 6          ||   shared = 7

        return shared
```

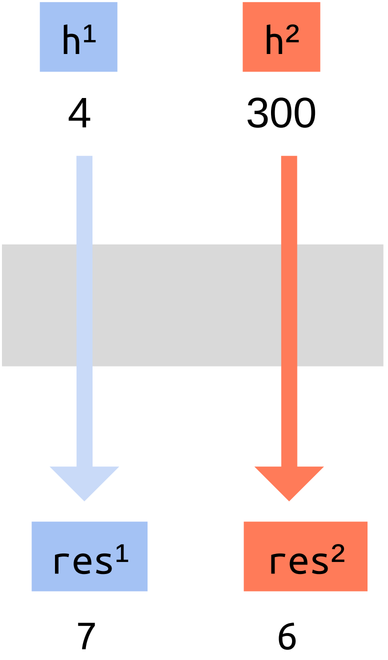


Shared-Memory Concurrency Ruins Everything

Secret-dependent execution time

Secret-**in**dependent execution time

```
while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
shared = 6            ||   shared = 7

        return shared
```

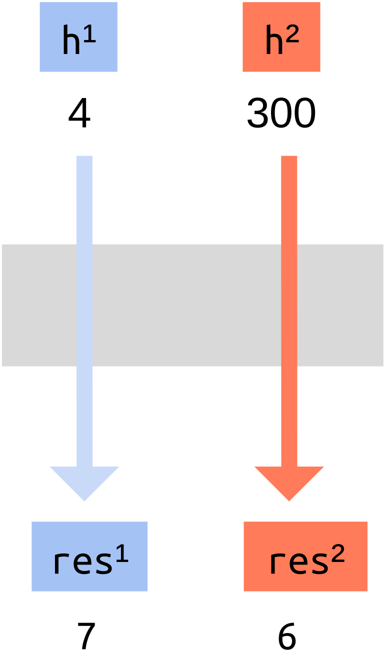


Shared-Memory Concurrency Ruins Everything

Secret-dependent execution time

Secret-**in**dependent execution time

```
while i < h:       ||   while j < 100:
  i += 1           ||     j += 1
shared = 6         ||   shared = 7

          return shared
```

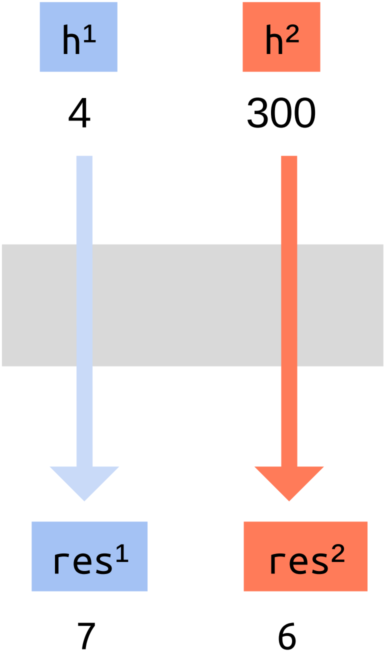


Shared-Memory Concurrency Ruins Everything 😢

Secret-dependent execution time

Secret-**in**dependent execution time

```
while i < h:      ||  while j < 100:
  i += 1          ||    j += 1
shared = 6        ||  shared = 7

return shared
```



Reasoning about Value Channel

Easy

Reasoning about Value Channel

Easy

Reasoning about Value Channel + Concurrency

Easy

Reasoning about Value Channel + Concurrency

⇓

Reasoning about Timing Channel

Easy

Reasoning about Value Channel + Concurrency

Reasoning about Timing Channel

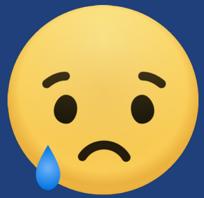
Hard

Easy

Reasoning about Value Channel + Concurrency

Reasoning about Timing Channel

Hard



Shared-Memory Concurrency Ruins Everything

```
while i < h:
  i += 1
shared = 6
```

‖

```
while j < 100:
  j += 1
shared = 7
```

```
return shared
```

Shared-Memory Concurrency Ruins Everything

```
while i < h:      ||   while j < 100:
  i += 1          ||     j += 1
shared = 6        ||   shared = 7

                  return shared
```

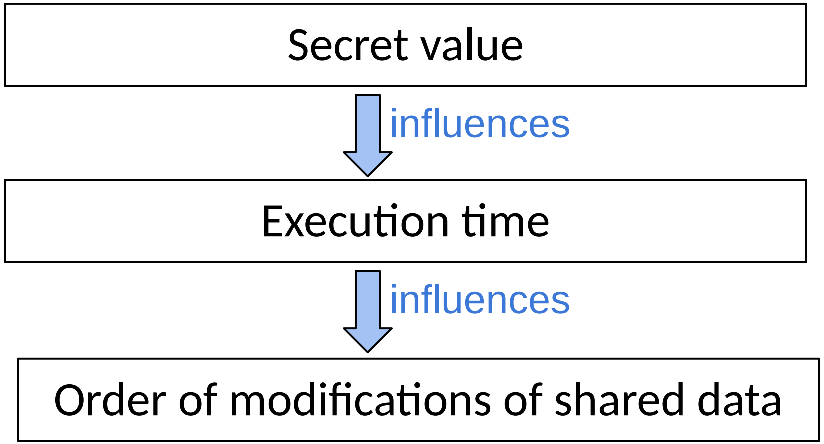
Secret value

↓ influences

Execution time

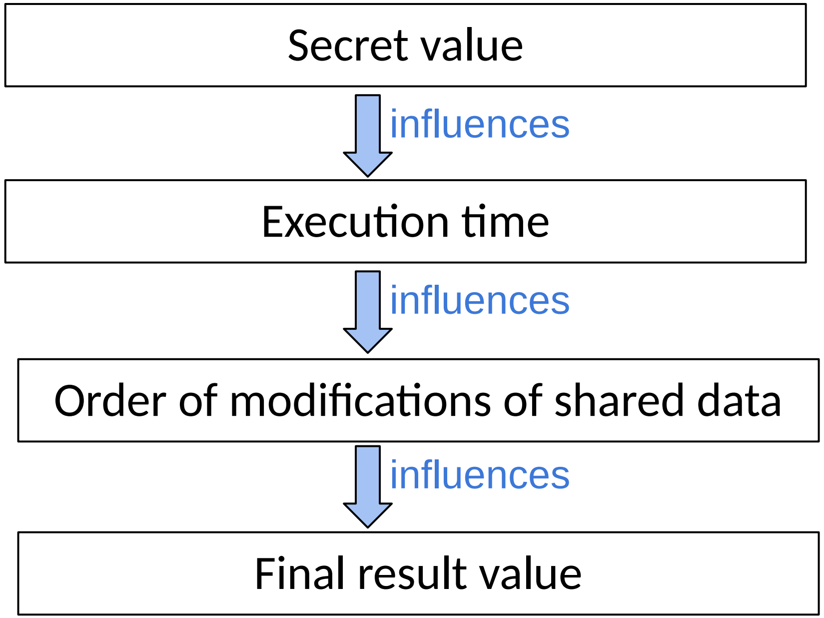
Shared-Memory Concurrency Ruins Everything

```
while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
shared = 6            ||   shared = 7

                 return shared
```



Shared-Memory Concurrency Ruins Everything

```
while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
shared = 6            ||   shared = 7

               return shared
```



Shared-Memory Concurrency Ruins Everything

```
while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
shared = 6            ||   shared = 7

                return shared
```

Secret value

↓ influences

Execution time

↓ influences

Order of modifications of shared data

↓ influences

Final result value

Existing (Modular) Solutions

```
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
shared = 6            ||  shared = 7

                return shared
```

Secret value

↓ influences

Execution time

↓ influences

Order of modifications of shared data

↓ influences

Final result value

Existing (Modular) Solutions

```
while i < h:        ||  while j < 100:
  i += 1            ||    j += 1
shared = 6          ||  shared = 7

                return shared
```

Secret value

↓ ~~influences~~

Execution time

↓ influences

Order of modifications of shared data

↓ influences

Final result value

Existing (Modular) Solutions

```
while i < h:         ||   while j < 100:
  i += 1             ||     j += 1
shared = 6           ||   shared = 7

          return shared
```

Secret value

~~in ences~~

Execution time

influences

Order of modifications of shared data

influences

Final result value

Existing (Modular) Solutions

Insecure

```
while i < h:          ||    while j < 100:
  i += 1                      j += 1
shared = 6                  shared = 7

                  return shared
```

Secret value

~~influences~~

Execution time

influences

Order of modifications of shared data

influences

Final result value

Existing (Modular) Solutions

Insecure

```
while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
shared = 6            ||   shared = 7

          return shared
```

```
          shared = l

while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
atomic:               ||   atomic:
  shared += 6         ||     shared += 7

          return shared
```

Secret value

↓ ~~influences~~

Execution time

↓ influences

Order of modifications of shared data

↓ influences

Final result value

Existing (Modular) Solutions

Insecure

```
while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
shared = 6            ||   shared = 7

              return shared
```

```
              shared = l
while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
atomic:               ||   atomic:
  shared += 6         ||     shared += 7

              return shared
```

Secret value

~~influences~~

Execution time

influences

Order of modifications of shared data

influences

Final result value

Existing (Modular) Solutions

Insecure

```
while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
shared = 6            ||   shared = 7

          return shared
```

Secure

```
          shared = l
while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
atomic:               ||   atomic:
  shared += 6         ||     shared += 7

          return shared
```

Secret value

~~influences~~

Execution time

influences

Order of modifications of shared data

influences

Final result value

Problem Statement

Reason about **values** in concurrent programs
without reasoning about **timing**
and without considering all **interleavings**

Key Idea

Order does not influence result if modifications **commute**

Our Solution: Commutativity

Secret values

↓ influence

Execution time

↓ influences

Order of modifications of shared data

↓ influences

Final result value

Our Solution: Commutativity

Secret values

↓ influence

Execution time

↓ influences

Order of modifications of shared data

↓ ~~influences~~

Final result value

Our Solution: Commutativity

Secure

```
              shared = l
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared += 6         ||    shared += 7
              return shared
```

Secret values

↓ influence

Execution time

↓ influences

Order of modifications of shared data

↓ ~~influences~~

Final result value

Our Solution: Commutativity

Secure

```
shared = l

while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
atomic:               ||   atomic:
  shared += 6         ||     shared += 7

return shared
```

Secret values

influence

Execution time

influences

Order of modifications of shared data

~~influences~~

Final result value

Our Solution: Commutativity

Secret values

↓ influence

Execution time

↓ influences

Order of modifications of shared data

↓ ~~influences~~

Final result value

Secure

```
            shared = l

while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared += 6         ||    shared += 7

            return shared
```

Our Solution: Commutativity

Insecure

```
while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
shared = 6            ||   shared = 7
                 return shared
```

Secure

```
                 shared = l
while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
atomic:               ||   atomic:
  shared += 6         ||     shared += 7
                 return shared
```

Secret values

↓ influence

Execution time

↓ influences

Order of modifications of shared data

↓ ~~influences~~

Final result value

Our Solution: Commutativity

Insecure

```
while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
shared = 6            ||   shared = 7

                return shared
```

Secure

```
                shared = l

while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
atomic:               ||   atomic:
  shared += 6         ||     shared += 7

                return shared
```

Secret values

↓ influence

Execution time

↓ influences

Order of modifications of shared data

↓ ~~influences~~

Final result value

Our Solution: Commutativity

Insecure

```
while i < h:              ||   while j < 100:
  i += 1                  ||     j += 1
shared = 6                ||   shared = 7

                     return shared
```

Secure

```
                     shared = l

while i < h:              ||   while j < 100:
  i += 1                  ||     j += 1
atomic:                   ||   atomic:
  shared += 6             ||     shared += 7

                     return shared
```

Secret values

↓ influence

Execution time

↓ influences

Order of modifications of shared data

↓ ~~influences~~

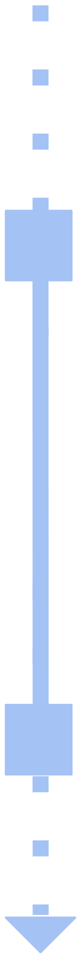
Final result value

Basic Solution

```
             shared = …
atomic:                 ||  atomic:
  shared = A            ||    shared = B
atomic:                 ||
  shared = C            ||
                  ...
```

Basic Solution

```
shared = …
atomic:            ||  atomic:
  shared = A       ||    shared = B
atomic:            ||
  shared = C       ||
                 …
```



Basic Solution

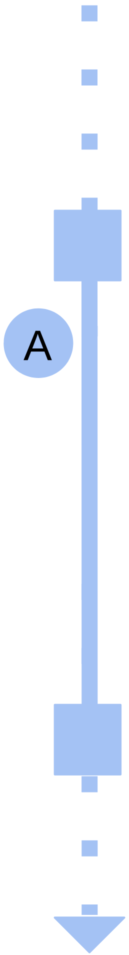
shared = …

```
atomic:
  shared = A
atomic:
  shared = C
```

||

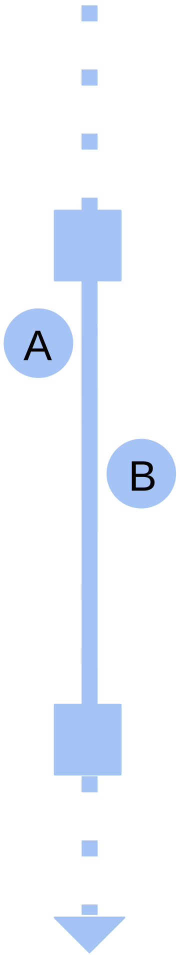
```
atomic:
  shared = B
```

…



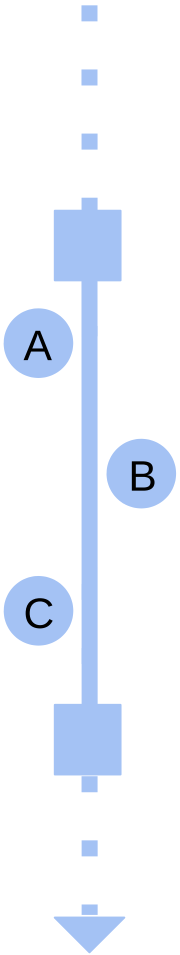
Basic Solution

```
        shared = …
atomic:              ||  atomic:
   shared = A        ||     shared = B
atomic:              ||
   shared = C        ||
           …
```



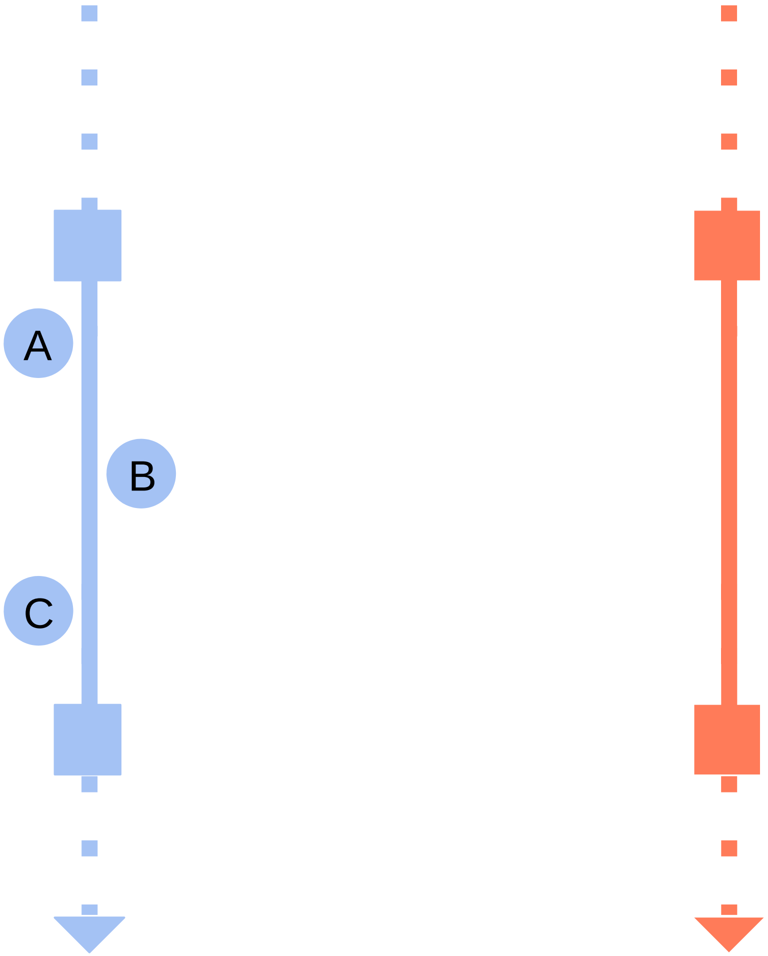
Basic Solution

```
shared = …
atomic:              ||  atomic:
  shared = A         ||    shared = B
atomic:              ||
  shared = C         ||
          …
```



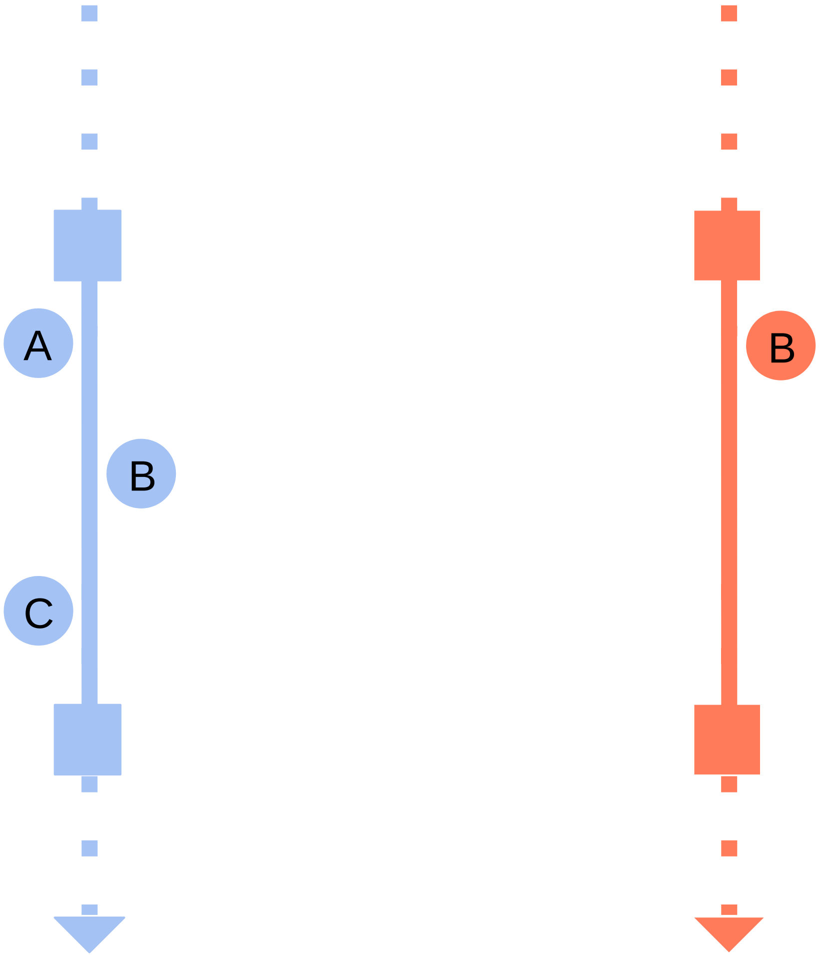
Basic Solution

```
shared = …
atomic:            ||  atomic:
  shared = A       ||    shared = B
atomic:            ||
  shared = C       ||
                   …
```



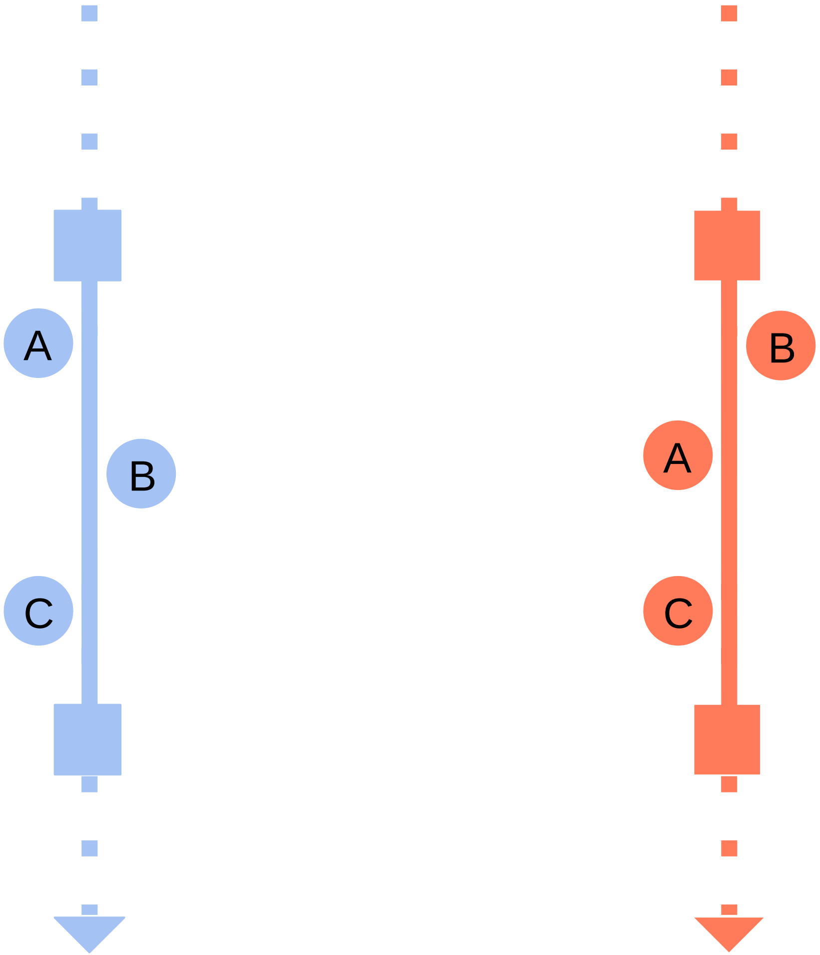
Basic Solution

```
          shared = …
atomic:                  ||  atomic:
   shared = A            ||     shared = B
atomic:                  ||
   shared = C            ||
               …
```



Basic Solution

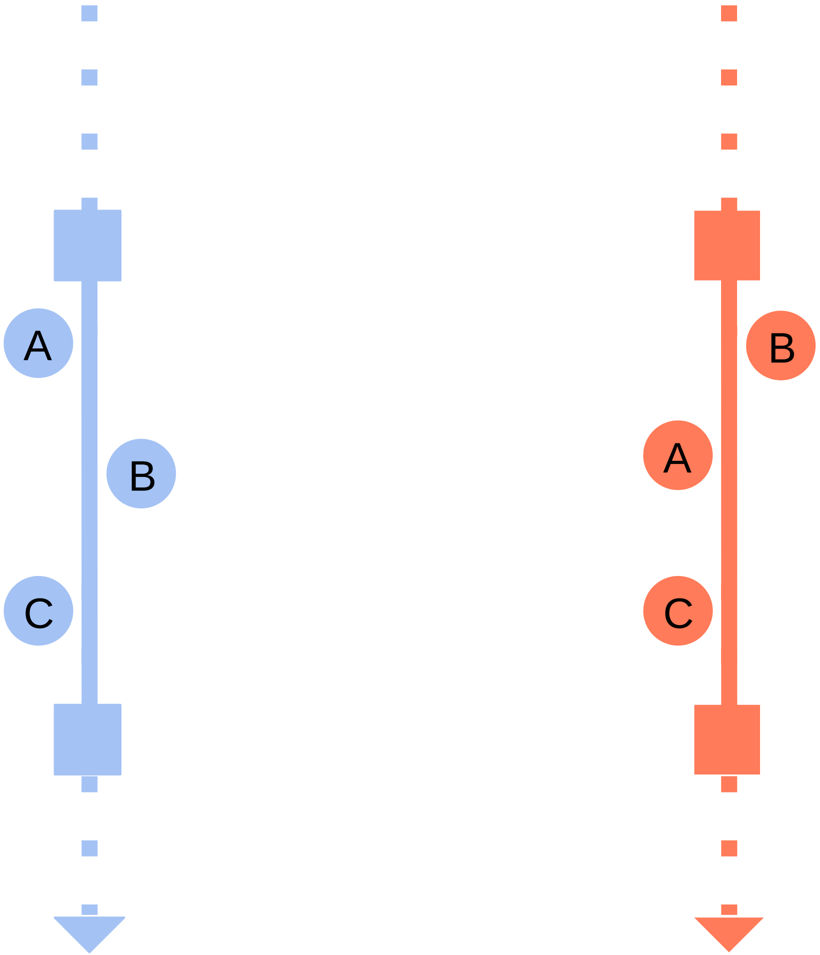
```
shared = …
atomic:              ||  atomic:
  shared = A         ||    shared = B
atomic:              ||
  shared = C         ||
            …
```



Basic Solution

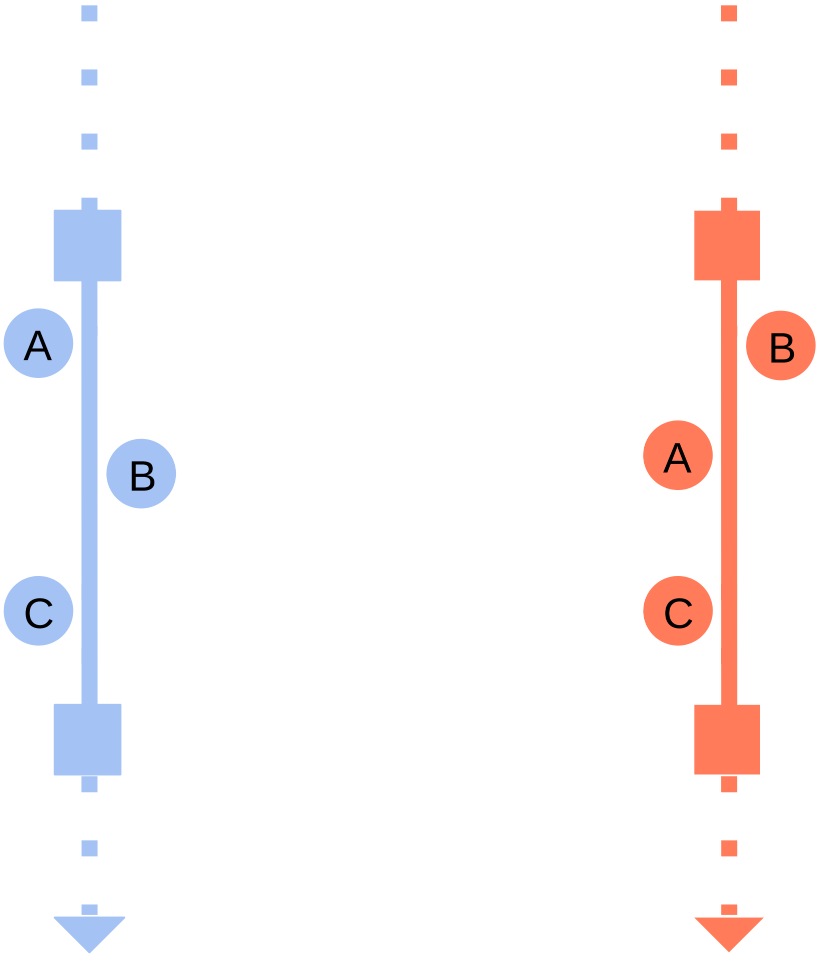
```
shared = …
atomic:            ||  atomic:
   shared = A      ||     shared = B
atomic:            ||
   shared = C      ||
        …
```

If



Basic Solution

```
            shared = …
atomic:              ||  atomic:
  shared = A         ||    shared = B
atomic:              ||
  shared = C         ||
               …
```

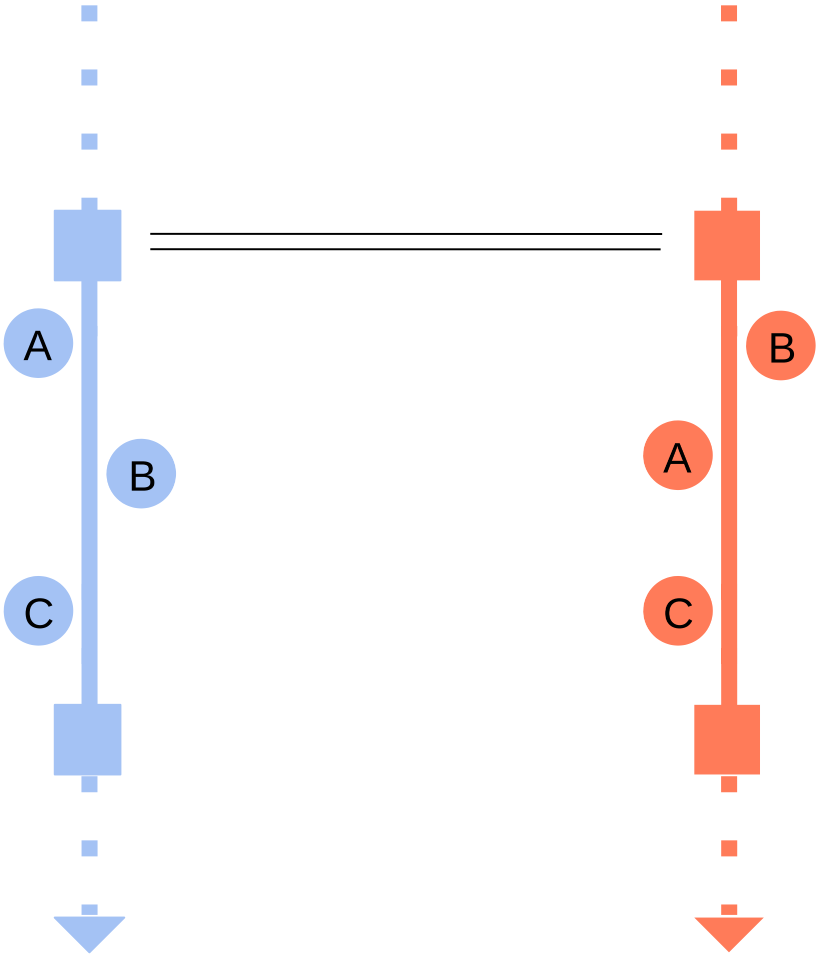


If

(1) *shared* has the same initial value in both executions

Basic Solution

```
shared = …
atomic:                 ||  atomic:
  shared = A            ||    shared = B
atomic:                 ||
  shared = C            ||
             …
```



If

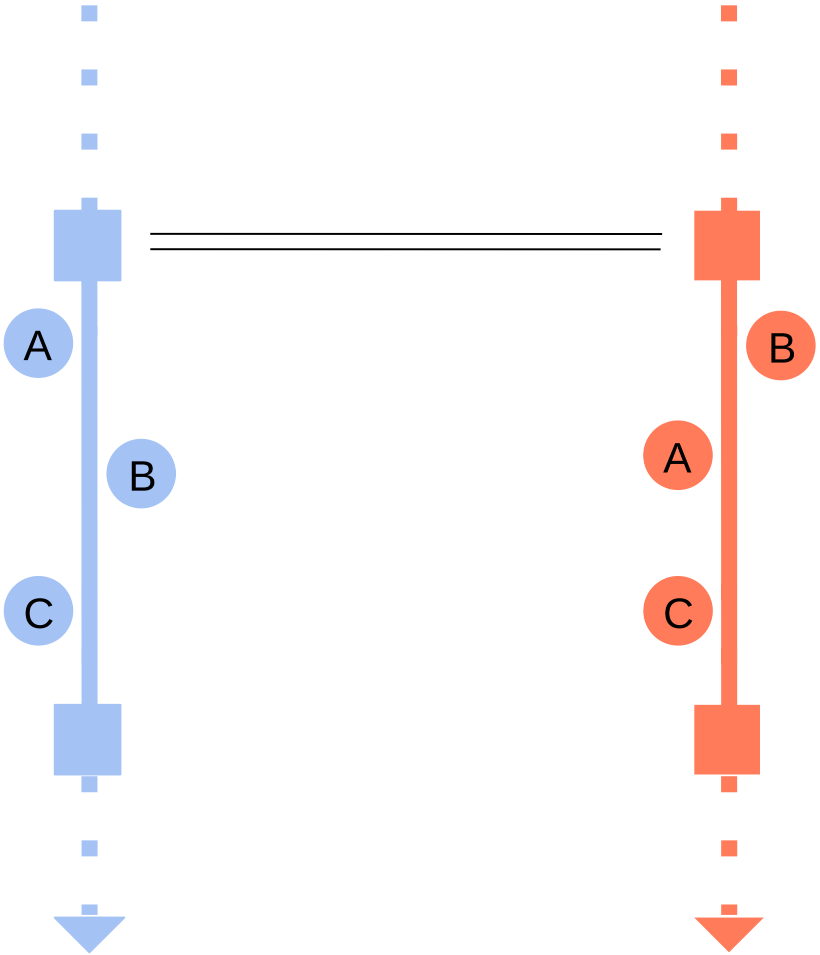
(1) *shared* has the same initial value in both executions

Basic Solution

shared = …

```
atomic:                   ||  atomic:
  shared = A              ||    shared = B
atomic:                   ||
  shared = C              ||
```

…



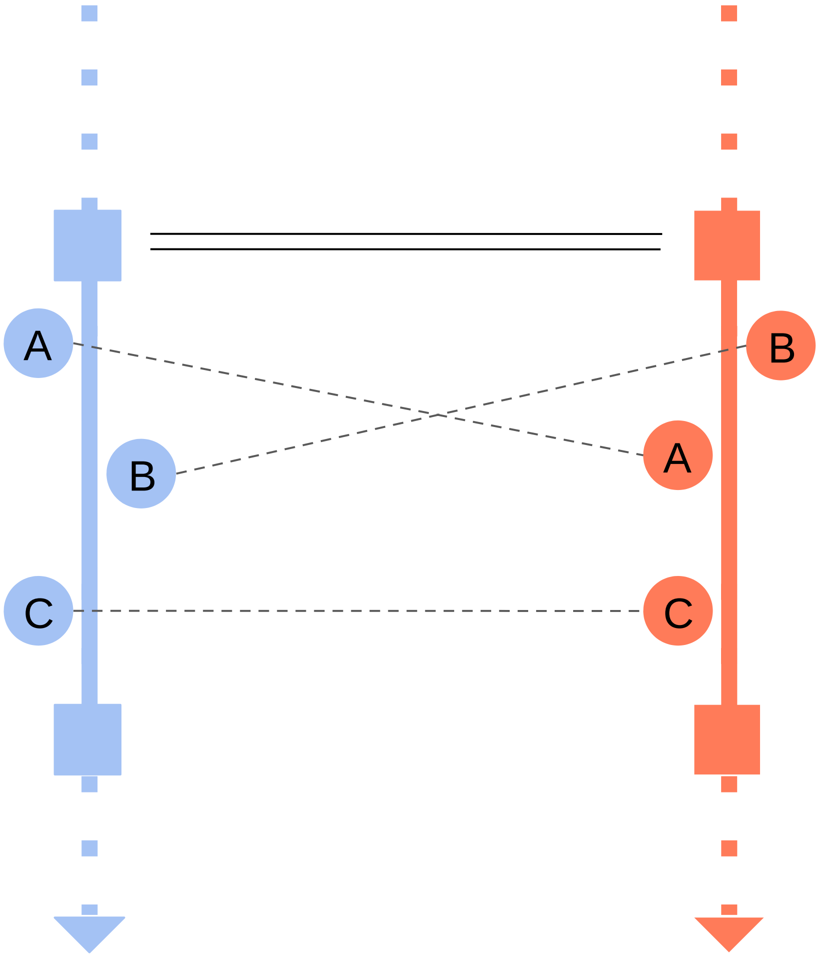
If

(1) *shared* has the same initial value in both executions

(2) the two executions perform the “same” modifications

Basic Solution

```
shared = …
atomic:                 ||  atomic:
  shared = A            ||    shared = B
atomic:                 ||
  shared = C            ||
          …
```



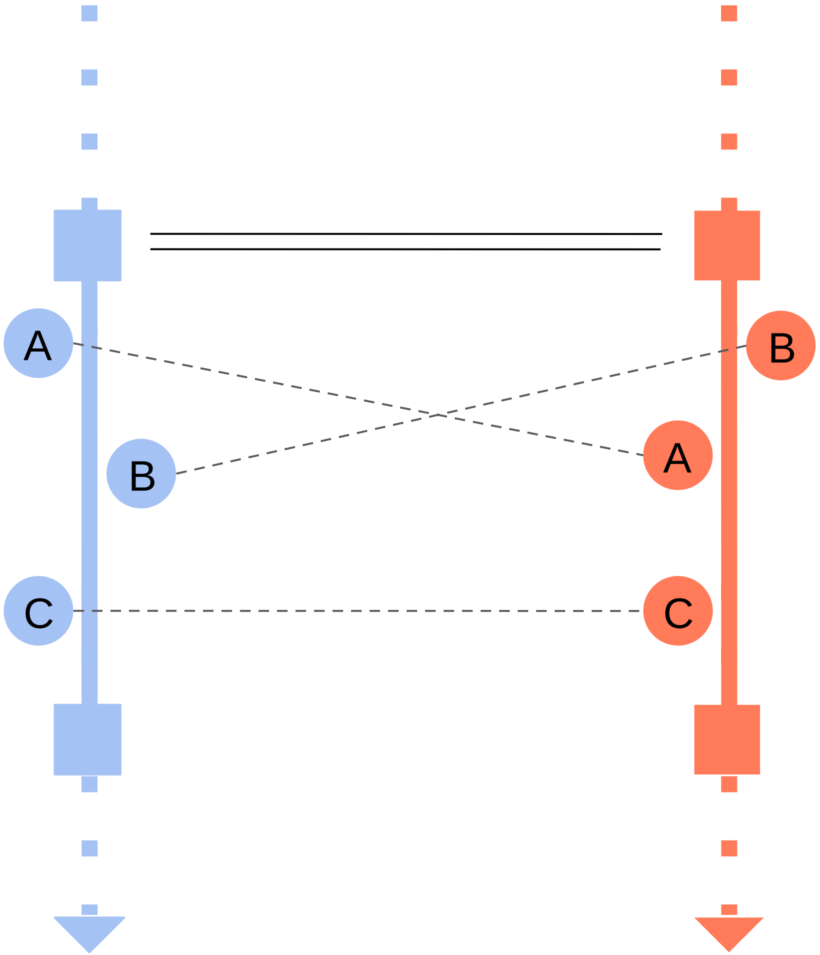
If

(1) *shared* has the same initial value in both executions

(2) the two executions perform the “same” modifications

Basic Solution

```
            shared = …
atomic:                  ||  atomic:
  shared = A             ||    shared = B
atomic:                  ||
  shared = C             ||
                 …
```



If

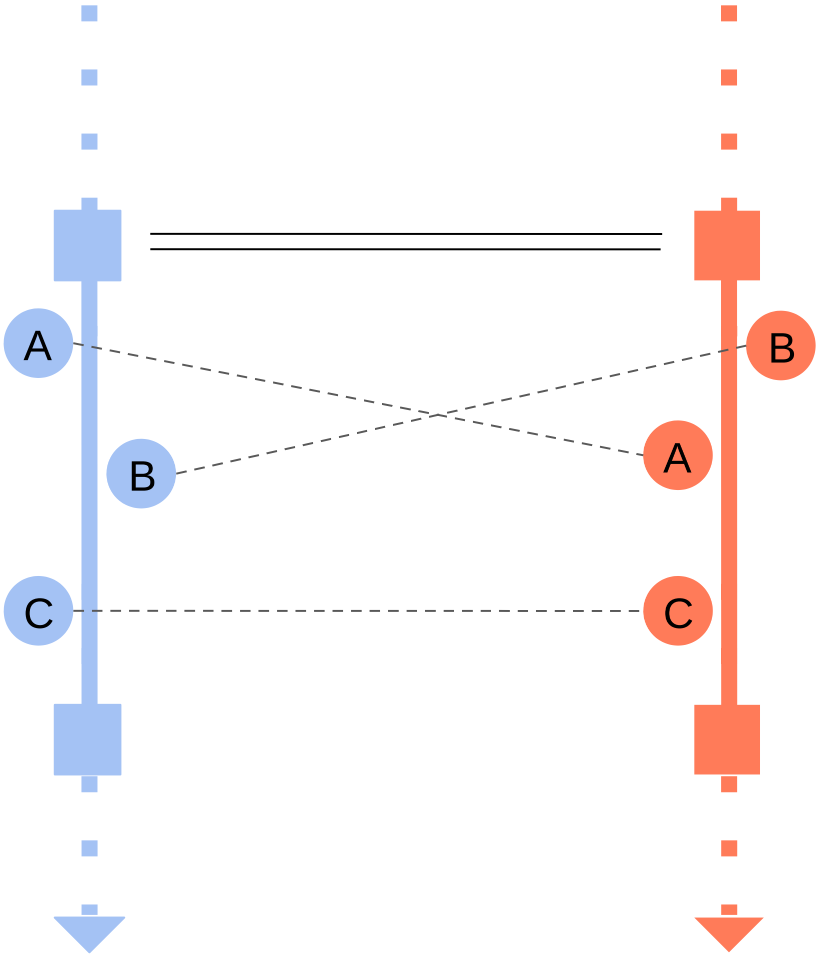
(1) *shared* has the same initial value in both executions

(2) the two executions perform the “same” modifications

(3) the modifications commute

Basic Solution

```
            shared = …
atomic:                ||  atomic:
   shared = A          ||     shared = B
atomic:                ||
   shared = C          ||
                 …
```



If

(1) *shared* has the same initial value in both executions

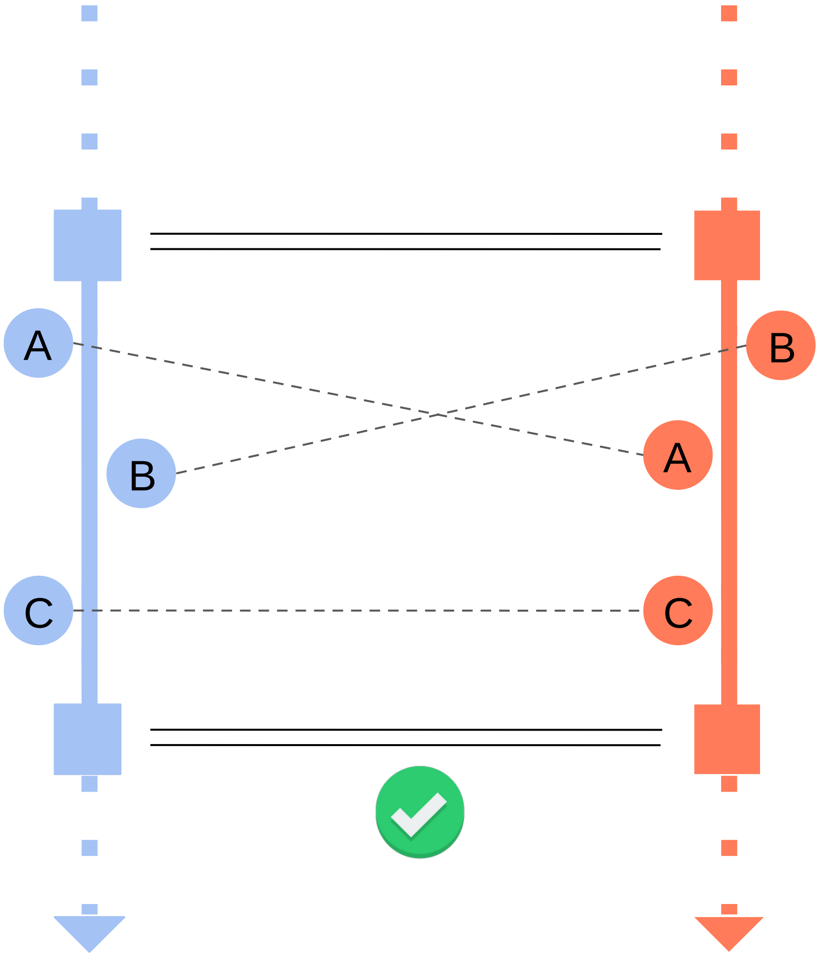
(2) the two executions perform the "same" modifications

(3) the modifications commute

then *shared* has the same final value in both executions

Basic Solution

```
            shared = …
atomic:                ||  atomic:
   shared = A          ||     shared = B
atomic:                ||
   shared = C          ||
                ...
```



If

(1) *shared* has the same initial value in both executions

(2) the two executions perform the "same" modifications

(3) the modifications commute

then *shared* has the same final value in both executions

Basic Solution

shared = 0

```
atomic:                 ||  ...
   shared += 1          ||  atomic:
atomic:                 ||     shared += 5
   shared += 3          ||
                 ...
```

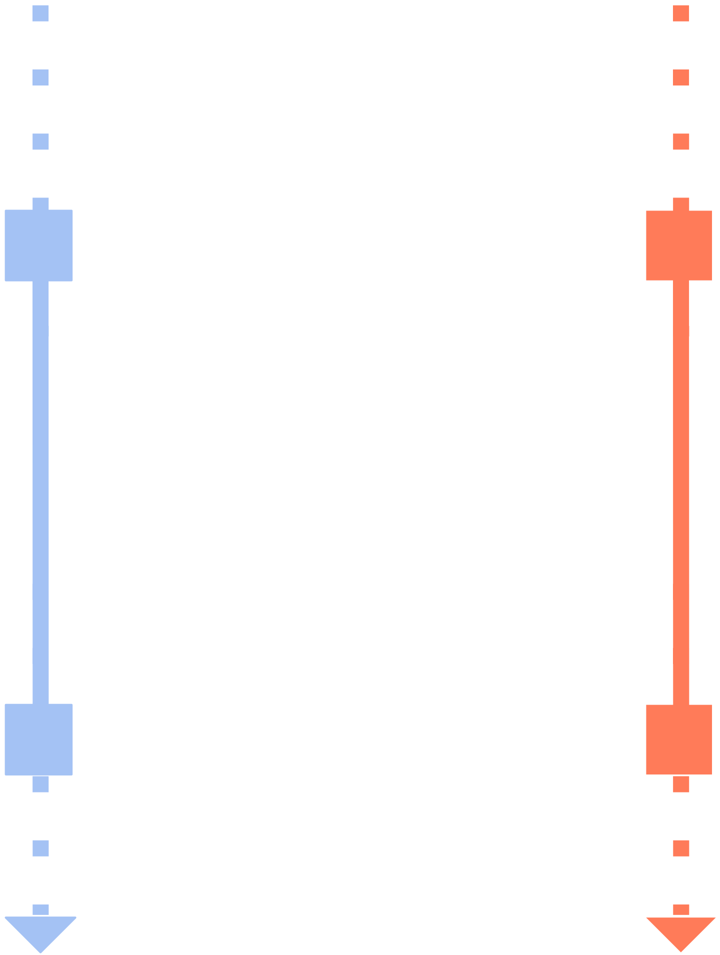
If

(1) *shared* has the same initial value in both executions

(2) the two executions perform the “same” modifications

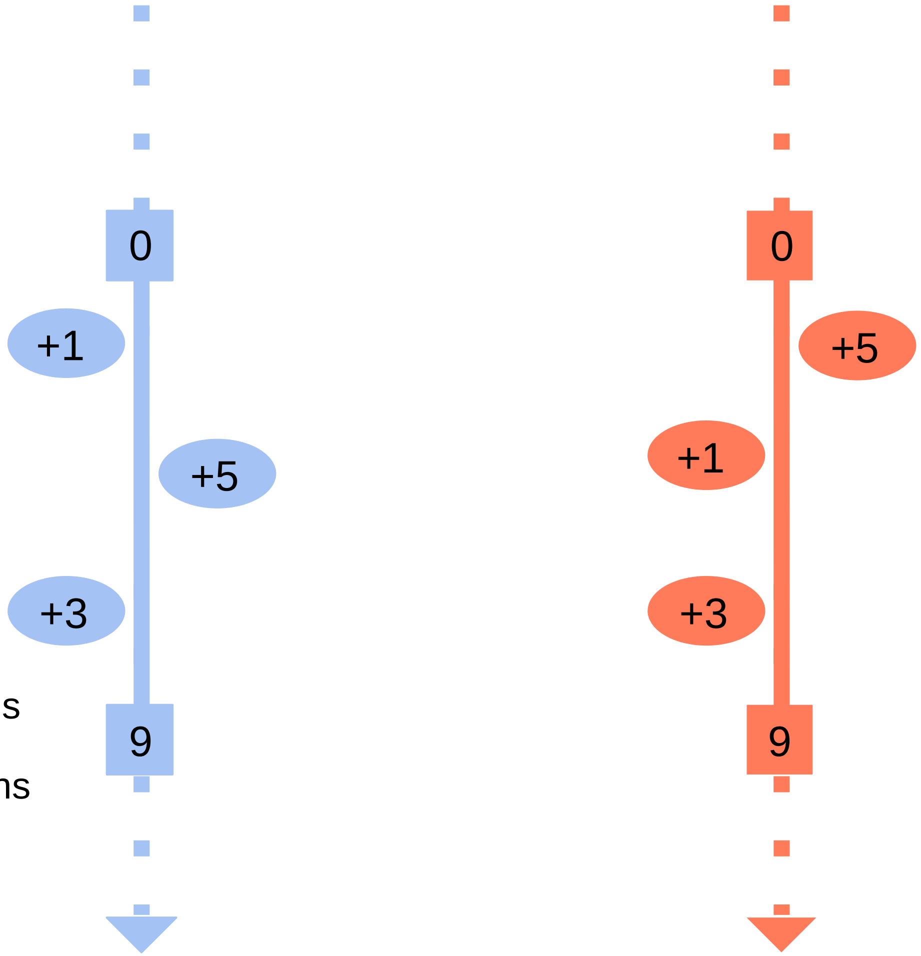
(3) the modifications commute

then *shared* has the same final value in both executions



Basic Solution

```
              shared = 0
atomic:                ||  ...
  shared += 1          ||  atomic:
atomic:                ||    shared += 5
  shared += 3          ||
                  ...
```



If

(1) *shared* has the same initial value in both executions

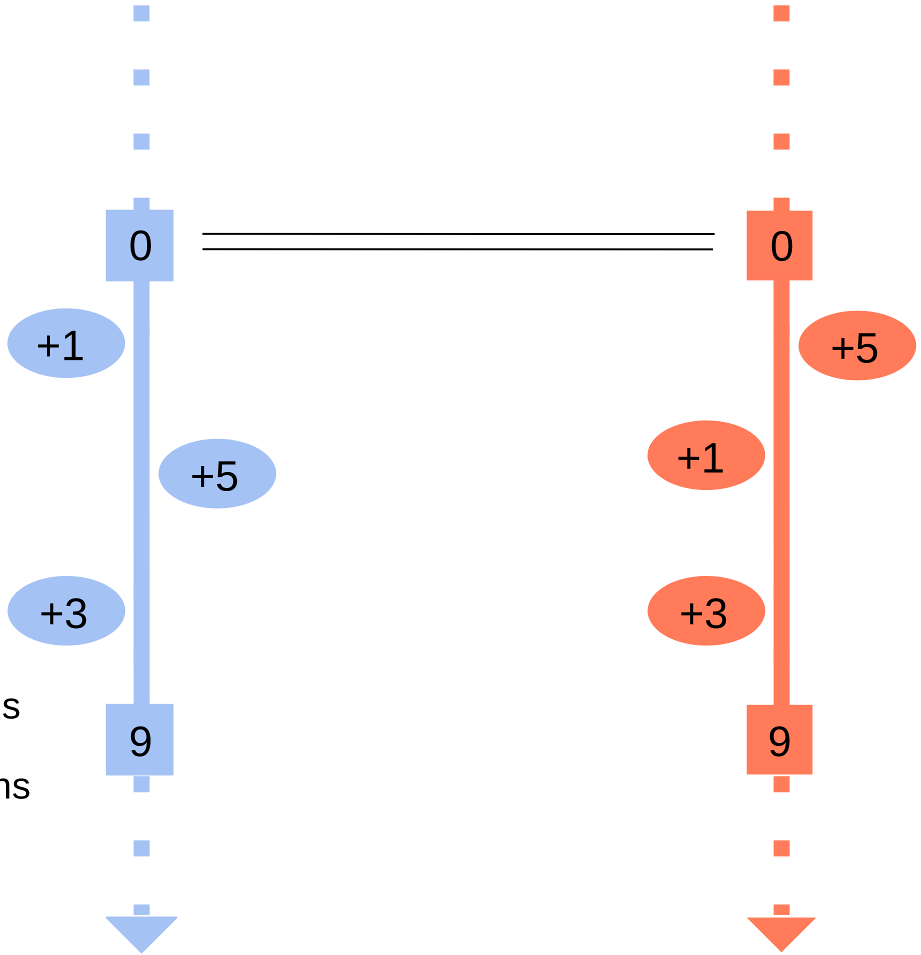
(2) the two executions perform the “same” modifications

(3) the modifications commute

then *shared* has the same final value in both executions

Basic Solution

```
shared = 0
atomic:              ||  ...
  shared += 1        ||  atomic:
atomic:              ||    shared += 5
  shared += 3        ||
             ...
```



If

(1) *shared* has the same initial value in both executions

(2) the two executions perform the “same” modifications

(3) the modifications commute

then *shared* has the same final value in both executions

Basic Solution

```
shared = 0
atomic:                 ||   ...
   shared += 1          ||   atomic:
atomic:                 ||      shared += 5
   shared += 3          ||
              ...
```

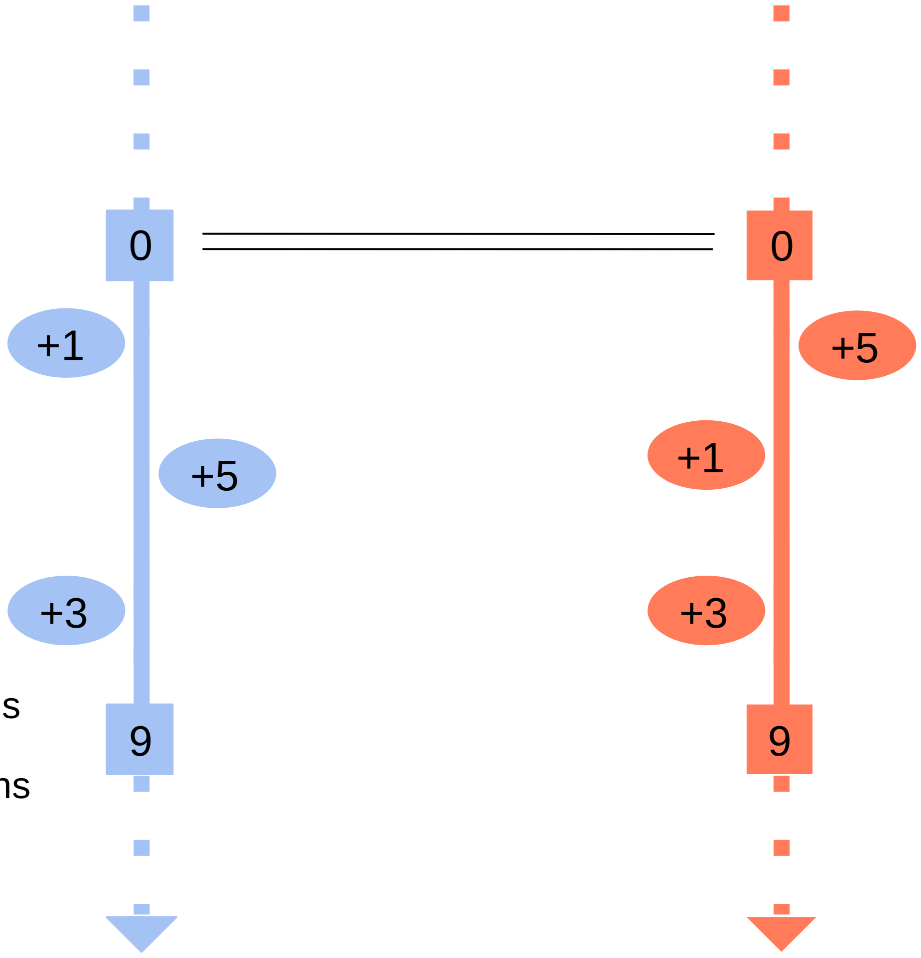
If

shared has the same initial value in both executions

(2) the two executions perform the “same” modifications

(3) the modifications commute

then *shared* has the same final value in both executions



Basic Solution

```
            shared = 0
atomic:              ||  ...
  shared += 1        ||  atomic:
atomic:              ||    shared += 5
  shared += 3        ||
                ...
```

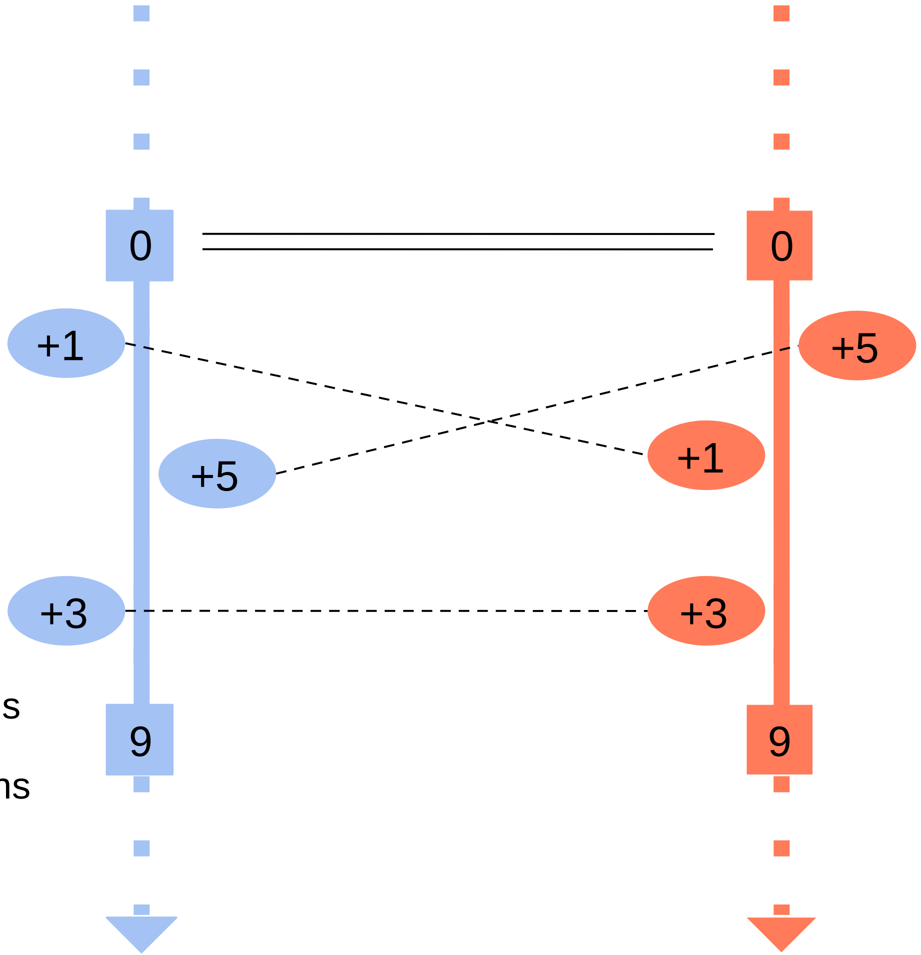
If

shared has the same initial value in both executions

(2) the two executions perform the “same” modifications

(3) the modifications commute

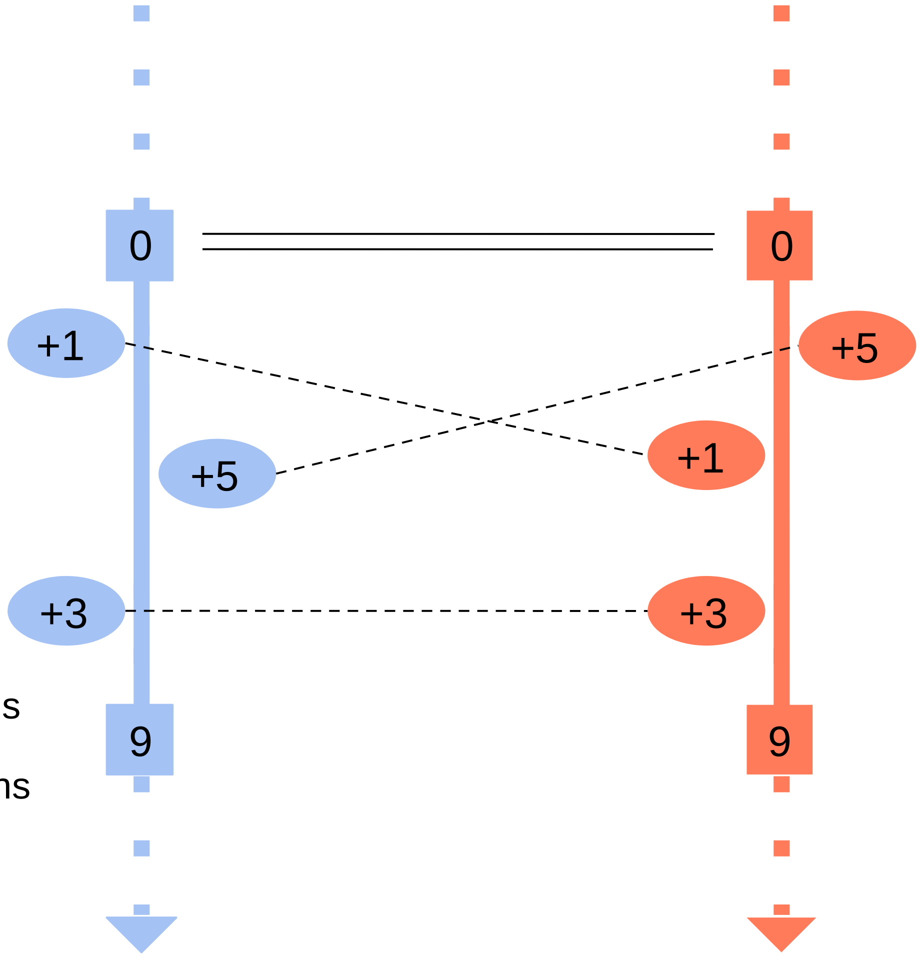
then *shared* has the same final value in both executions



Basic Solution

shared = 0

```
atomic:                ||  ...
  shared += 1          ||  atomic:
atomic:                ||    shared += 5
  shared += 3          ||
                 ...
```



If

- ✅ *shared* has the same initial value in both executions
- ✅ the two executions perform the "same" modifications

(3) the modifications commute

then *shared* has the same final value in both executions

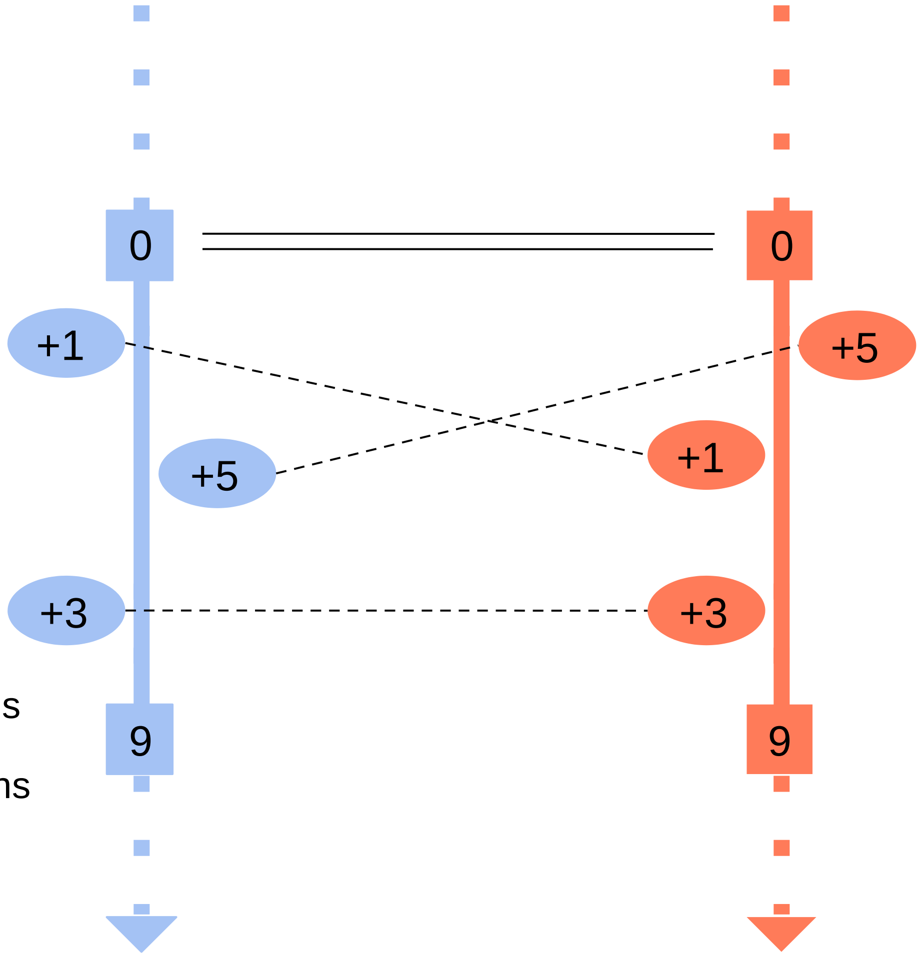
Basic Solution

```
shared = 0
atomic:              ||   ...
  shared += 1        ||   atomic:
atomic:              ||     shared += 5
  shared += 3        ||
          ...
```

If

- *shared* has the same initial value in both executions
- the two executions perform the “same” modifications
- the modifications commute

then *shared* has the same final value in both executions



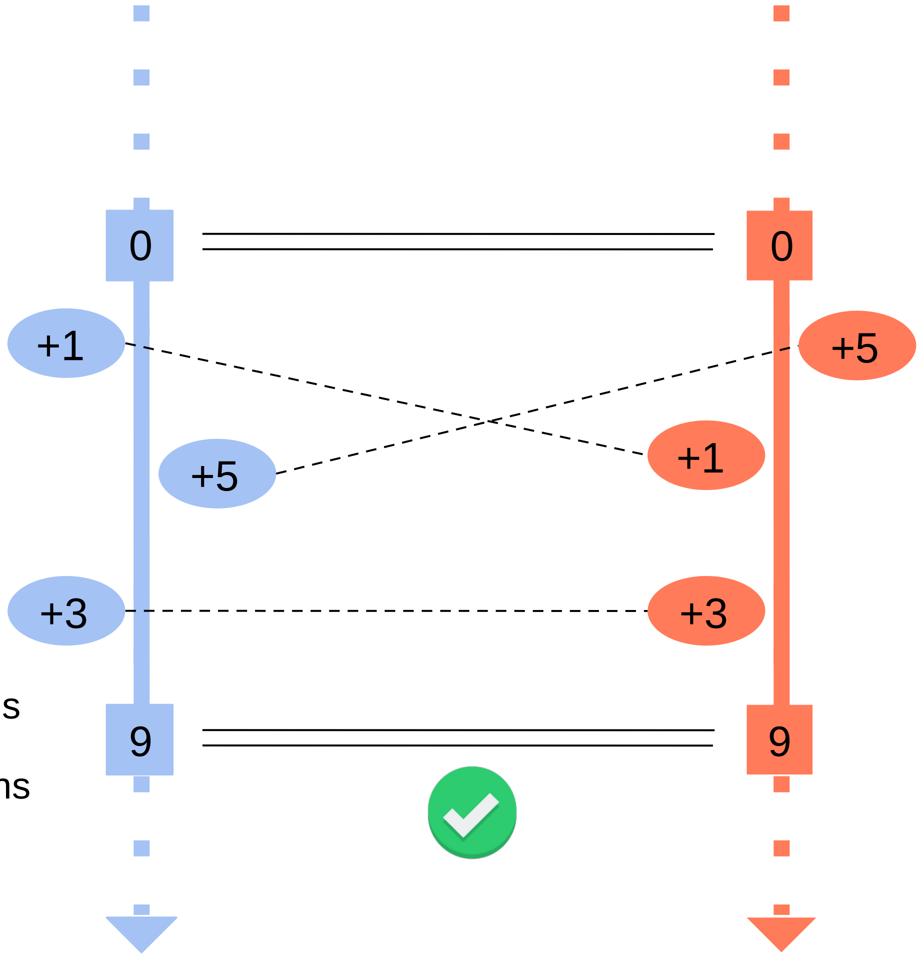
Basic Solution

```
shared = 0
atomic:              ||  ...
  shared += 1        ||  atomic:
atomic:              ||    shared += 5
  shared += 3        ||
                    ...
```

If

- *shared* has the same initial value in both executions
- the two executions perform the "same" modifications
- the modifications commute

then *shared* has the same final value in both executions



Basic Solution

shared = I

```
atomic:                 ||  atomic:
  shared = A            ||    shared = B
atomic:                 ||  if h > 0:
  shared = C            ||    atomic:
                        ||      shared = B
                        ...
```

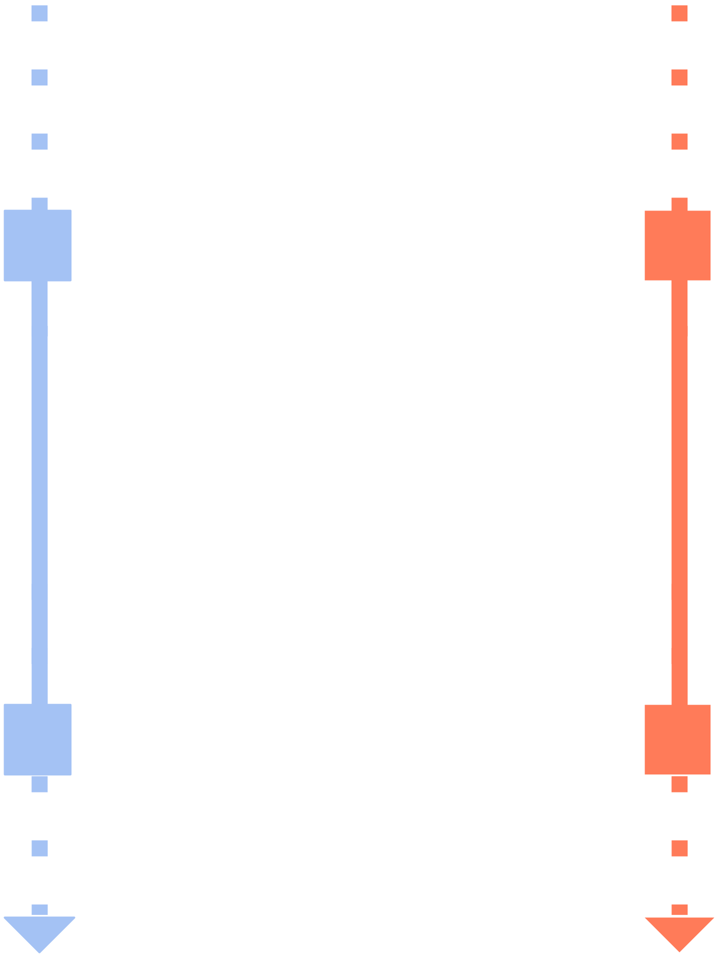
If

(1) *shared* has the same initial value in both executions

(2) the two executions perform the “same” modifications

(3) the modifications commute

then *shared* has the same final value in both executions



Basic Solution

```
              shared = I
atomic:                    ||  atomic:
   shared = A              ||     shared = B
atomic:                    ||  if h > 0:
   shared = C              ||     atomic:
                           ||        shared = B
                          ...
```

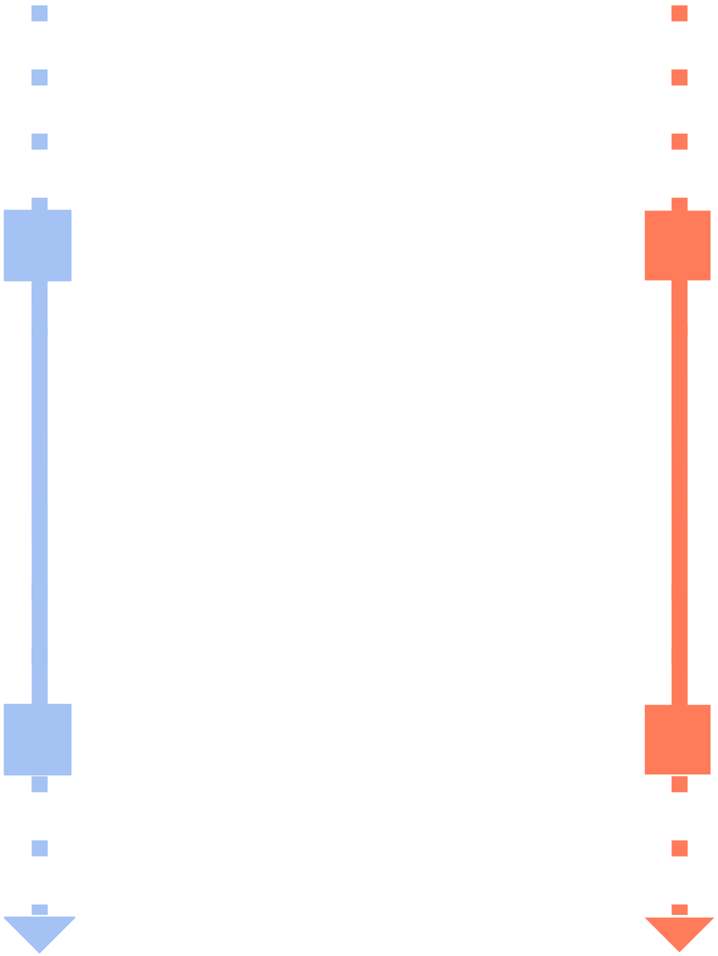
If

(1) *shared* has the same initial value in both executions

(2) the two executions perform the “same” modifications

(3) the modifications commute

then *shared* has the same final value in both executions



Basic Solution

```
              shared = I
atomic:                 ||  atomic:
  shared = A            ||    shared = B
atomic:                 ||  if h > 0:
  shared = C            ||    atomic:
                        ||      shared = B
                       ...
```

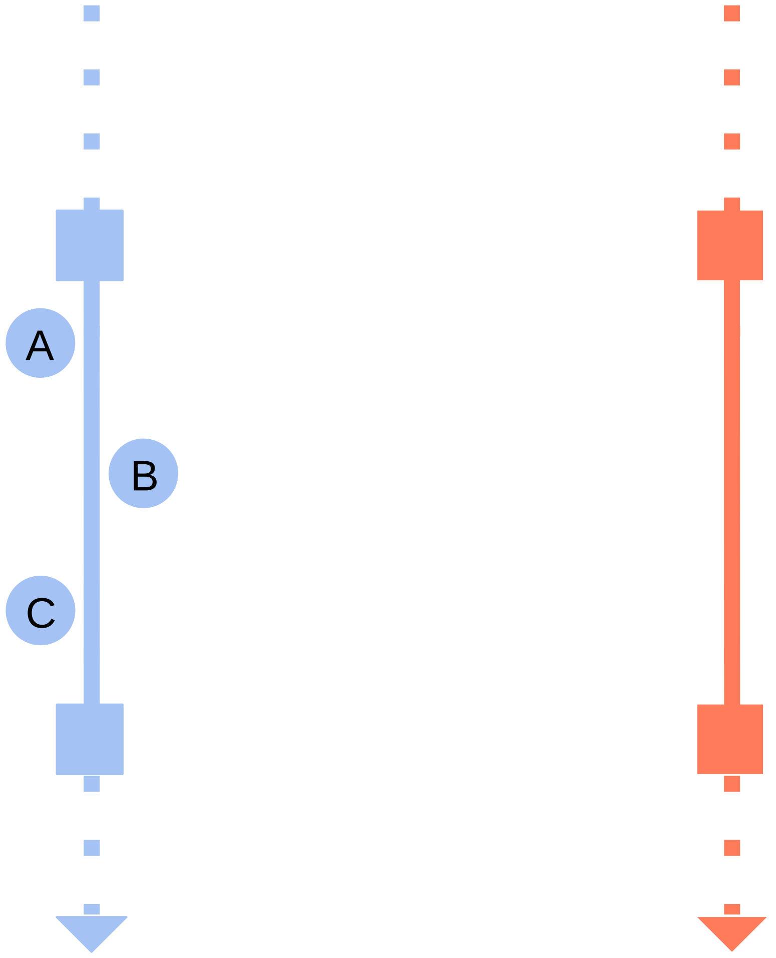
If

(1) *shared* has the same initial value in both executions

(2) the two executions perform the “same” modifications

(3) the modifications commute

then *shared* has the same final value in both executions



Basic Solution

```
            shared = I
atomic:                ||  atomic:
  shared = A           ||    shared = B
atomic:                ||  if h > 0:
  shared = C           ||    atomic:
                       ||      shared = B
                      ...
```

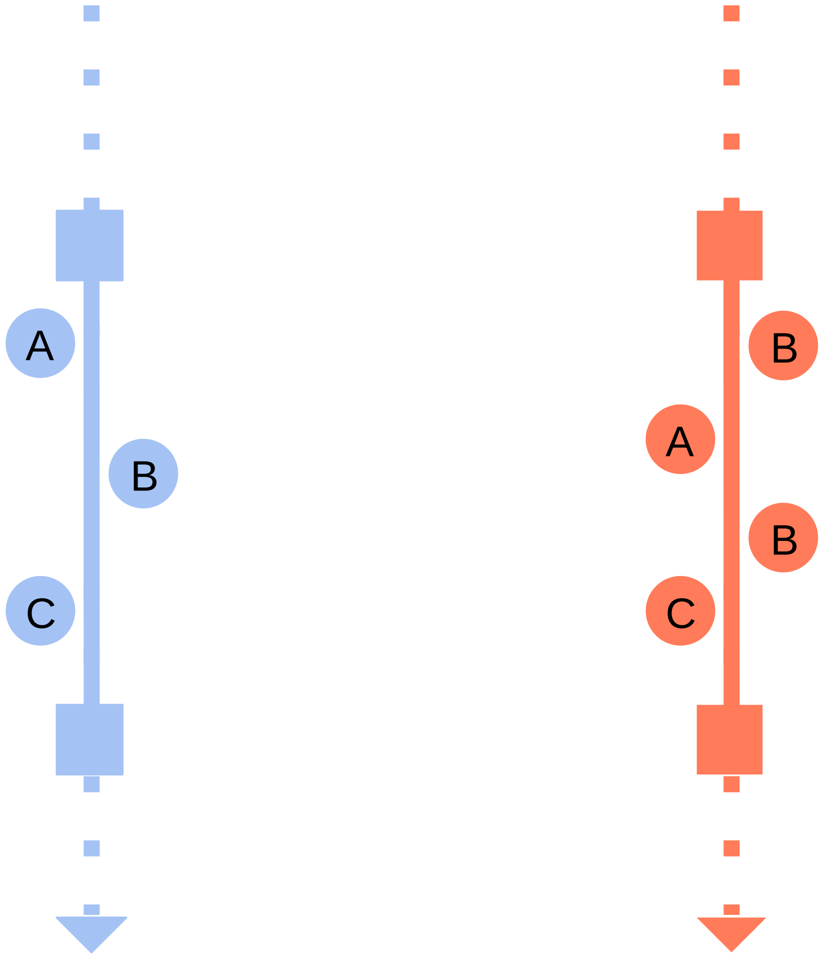
If

(1) *shared* has the same initial value in both executions

(2) the two executions perform the “same” modifications

(3) the modifications commute

then *shared* has the same final value in both executions



Basic Solution

```
            shared = I
atomic:                  ||  atomic:
  shared = A             ||    shared = B
atomic:                  ||  if h > 0:
  shared = C             ||    atomic:
                         ||      shared = B
                        ...
```

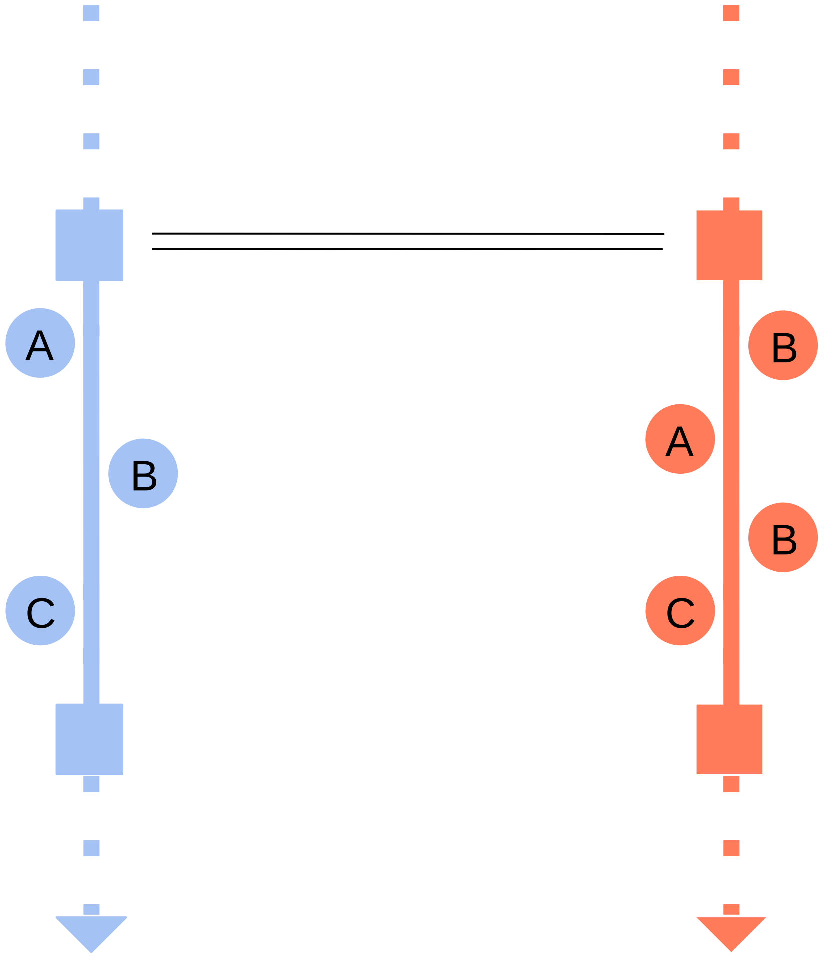
If

shared has the same initial value in both executions

(2) the two executions perform the “same” modifications

(3) the modifications commute

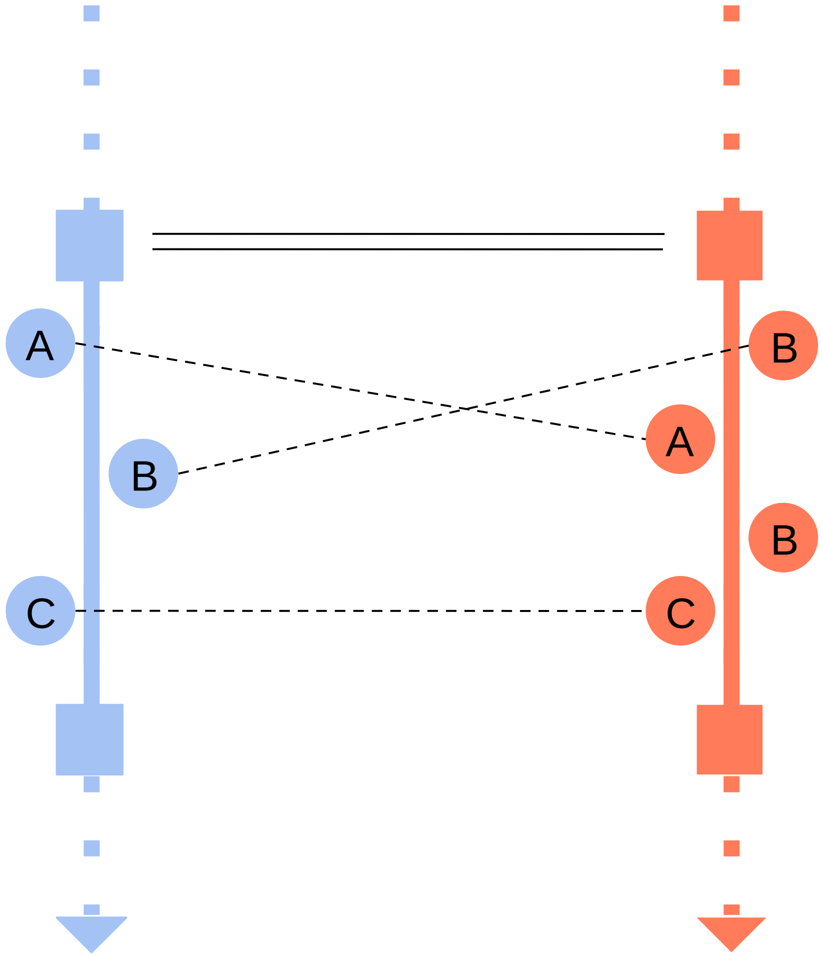
then *shared* has the same final value in both executions



Basic Solution

shared = l

```
atomic:                  ||   atomic:
  shared = A             ||     shared = B
atomic:                  ||   if h > 0:
  shared = C             ||     atomic:
                         ||       shared = B
                        ...
```



If

shared has the same initial value in both executions

(2) the two executions perform the “same” modifications

(3) the modifications commute

then *shared* has the same final value in both executions

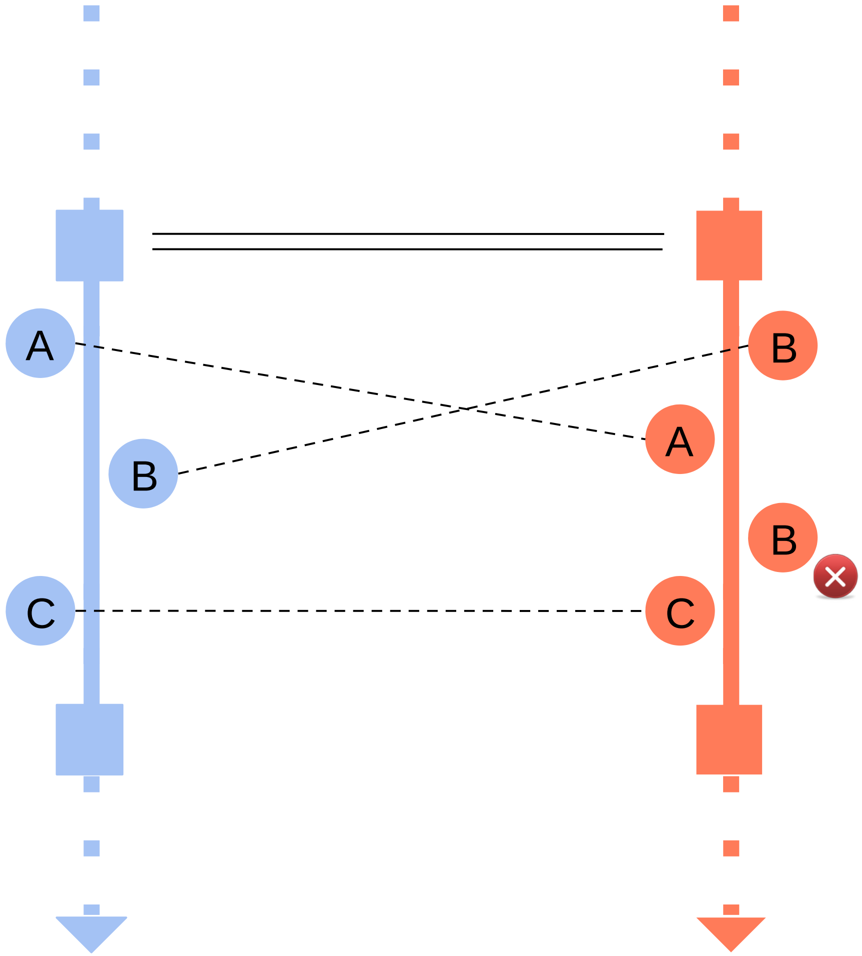
Basic Solution

```
              shared = I
atomic:                 ||  atomic:
  shared = A            ||    shared = B
atomic:                 ||  if h > 0:
  shared = C            ||    atomic:
                        ||      shared = B
                       ...
```

If

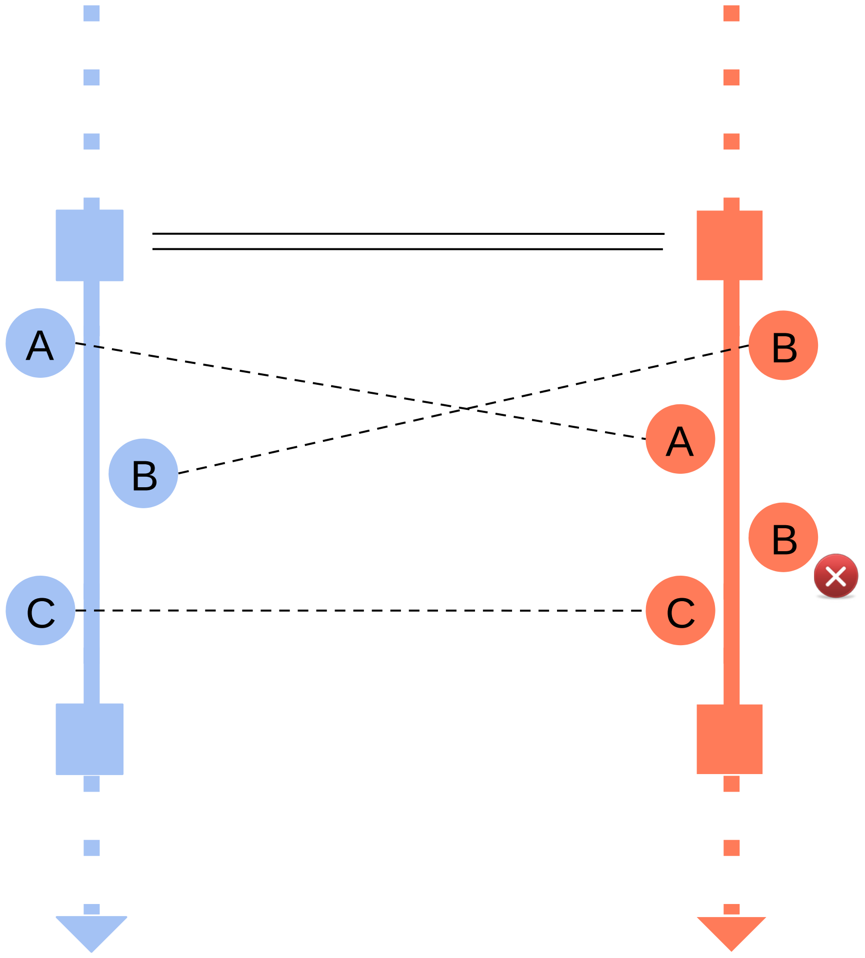
- *shared* has the same initial value in both executions
- (2) the two executions perform the “same” modifications
- (3) the modifications commute

then *shared* has the same final value in both executions



Basic Solution

```
shared = I

atomic:              ||  atomic:
  shared = A         ||    shared = B
atomic:              ||  if h > 0:
  shared = C         ||    atomic:
                     ||      shared = B
                    ...
```



If

- ✔ *shared* has the same initial value in both executions
- ✘ the two executions perform the “same” modifications
- (3) the modifications commute

then *shared* has the same final value in both executions

Basic Solution

```
shared = I
atomic:                 ||  atomic:
  shared = A            ||    shared = B
atomic:                 ||  if h > 0:
  shared = C            ||    atomic:
                        ||      shared = B
                       ...
```

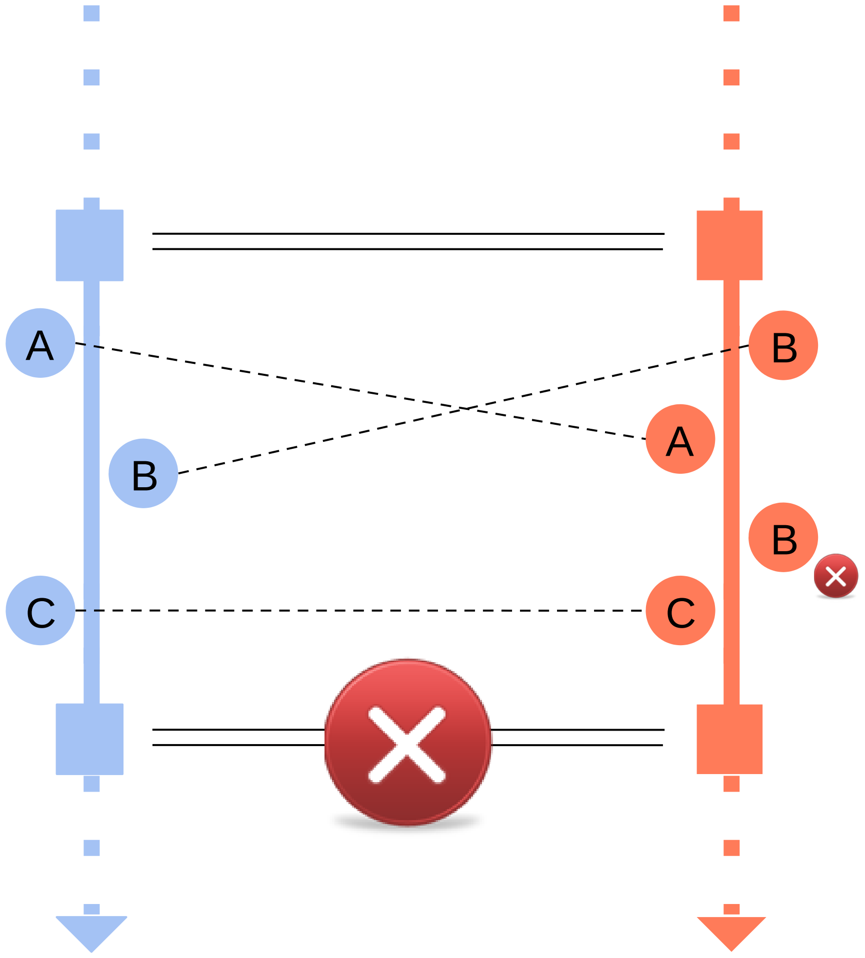
If

shared has the same initial value in both executions

the two executions perform the “same” modifications

(3) the modifications commute

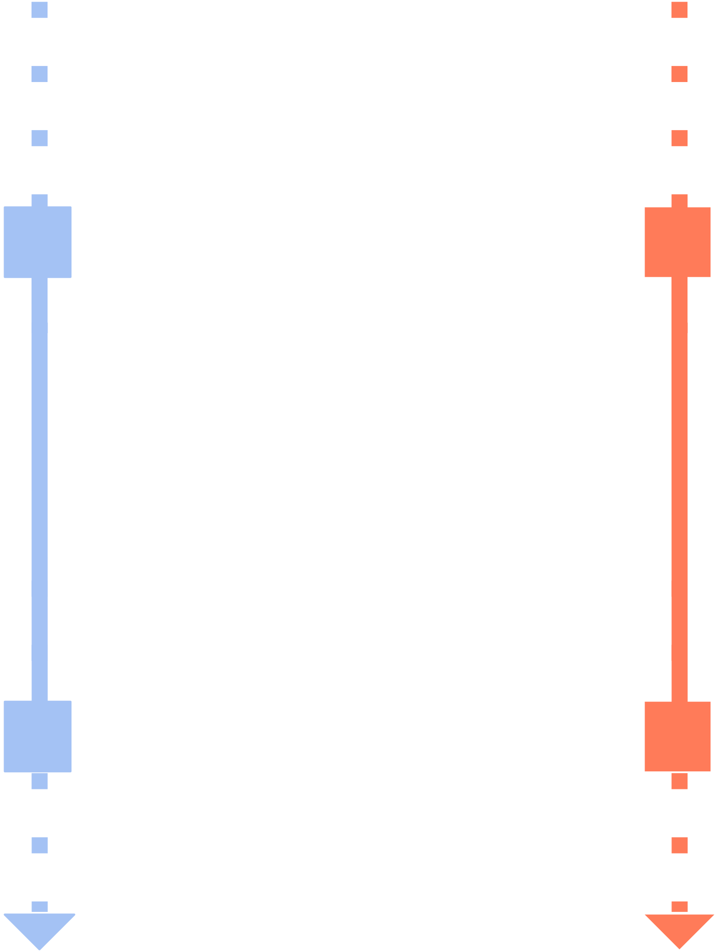
then *shared* has the same final value in both executions



Basic Solution

shared = I

```
atomic:                    ||  atomic:
  shared *= ...            ||    shared += ...
                           ||
atomic:                    ||
  shared += ...            ||
             ...
```



If

(1) *shared* has the same initial value in both executions

(2) the two executions perform the “same” modifications

(3) the modifications commute

then *shared* has the same final value in both executions

Basic Solution

shared = I

```
atomic:                    ||  atomic:
  shared *= ...            ||    shared += ...
                           ||
atomic:                    ||
  shared += ...            ||
                 ...
```

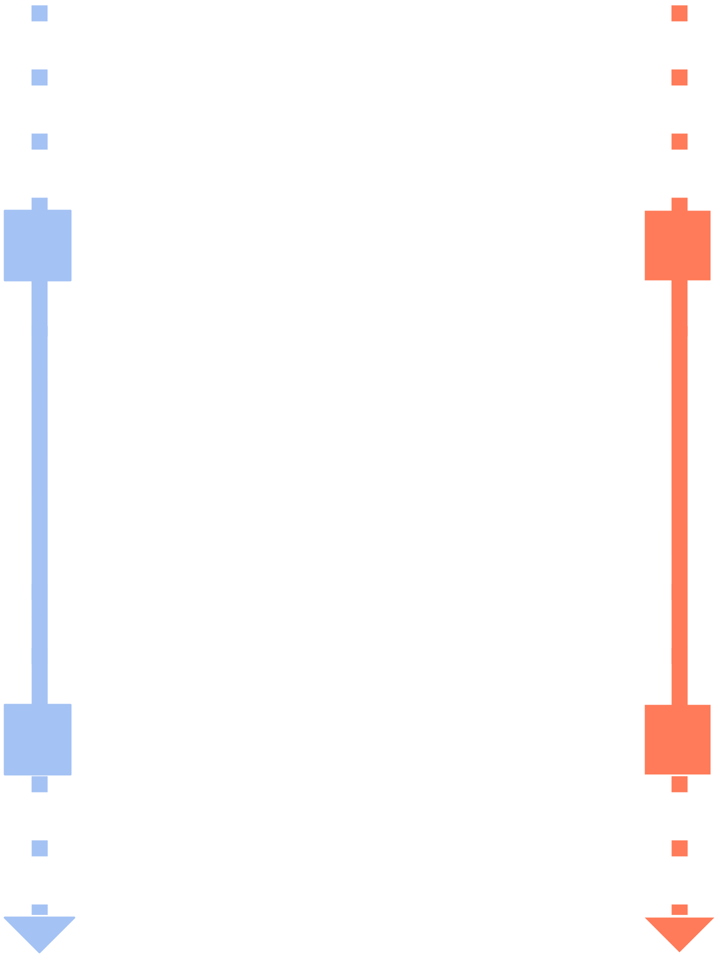
If

(1) *shared* has the same initial value in both executions

(2) the two executions perform the “same” modifications

(3) the modifications commute

then *shared* has the same final value in both executions



Basic Solution

```
           shared = I
atomic:                 || atomic:
  shared *= ...         ||   shared += ...
                        ||
atomic:                 ||
  shared += ...         ||
                   ...
```

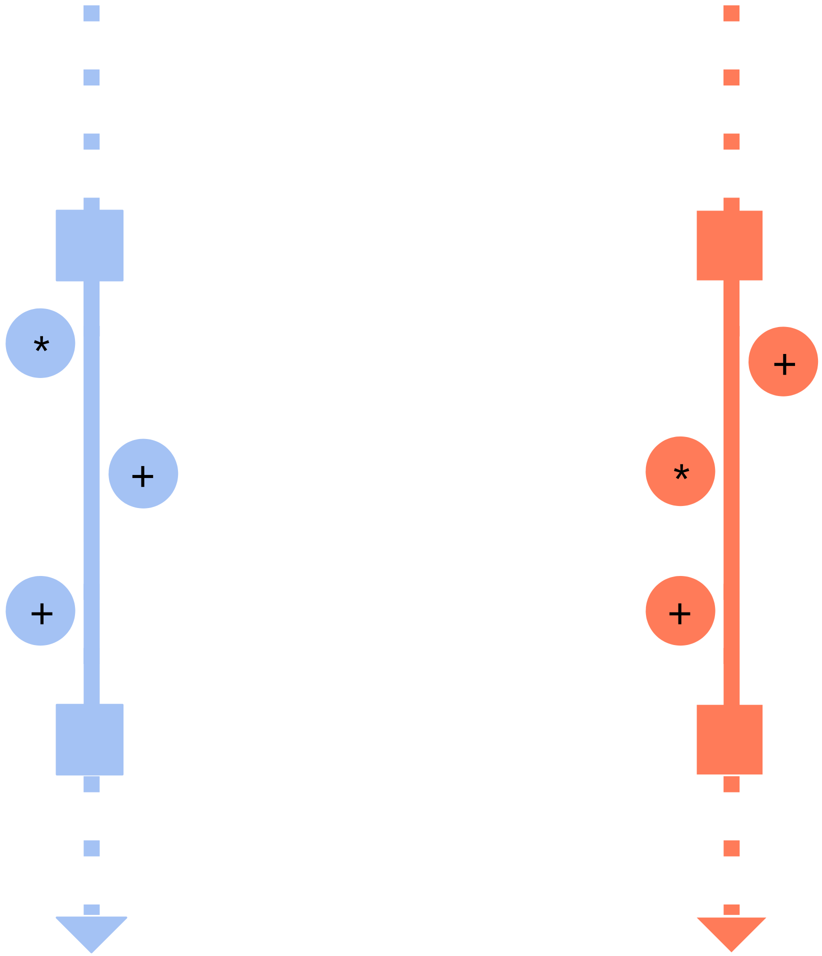
If

(1) *shared* has the same initial value in both executions

(2) the two executions perform the “same” modifications

(3) the modifications commute

then *shared* has the same final value in both executions



Basic Solution

```
shared = I
atomic:                   ||  atomic:
  shared *= ...           ||    shared += ...

atomic:                   ||
  shared += ...           ||
            ...
```

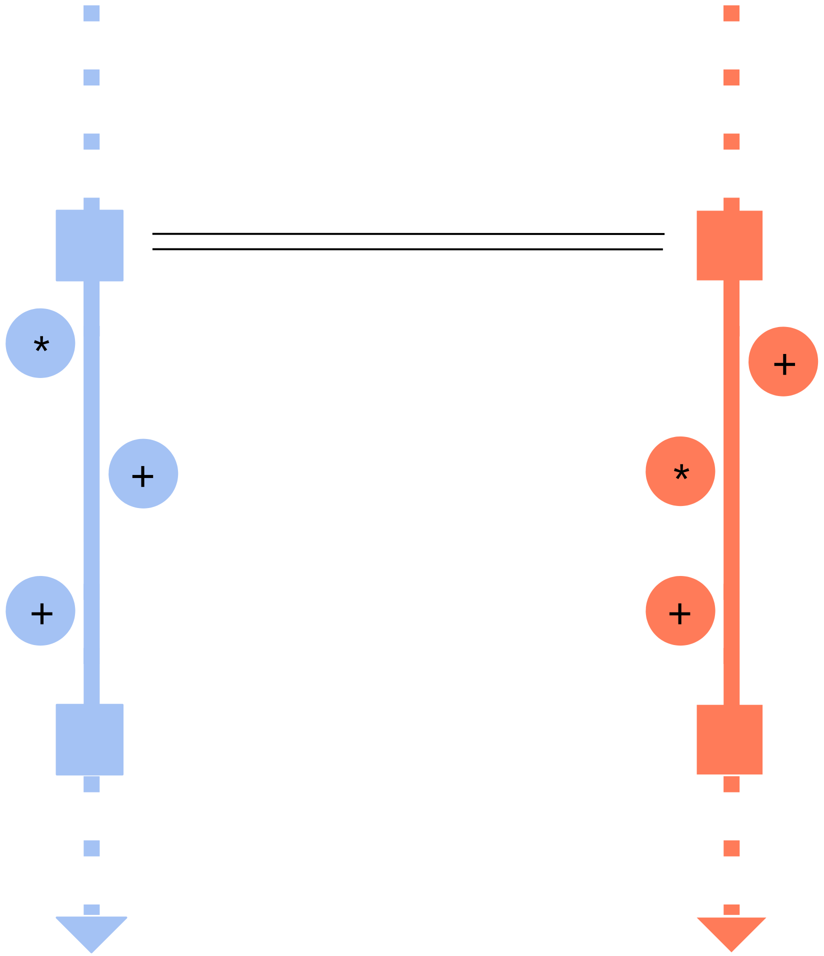
If

✔ *shared* has the same initial value in both executions

(2) the two executions perform the “same” modifications

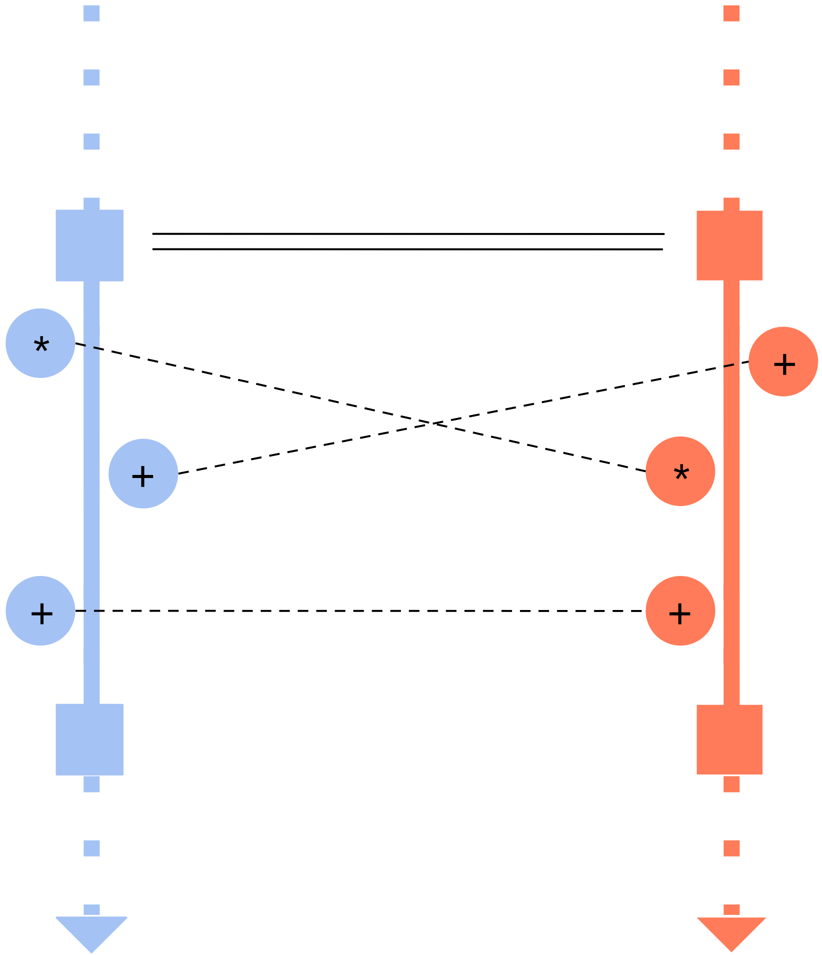
(3) the modifications commute

then *shared* has the same final value in both executions



Basic Solution

```
           shared = I
atomic:                  ||  atomic:
  shared *= ...          ||    shared += ...

atomic:
  shared += ...          ||
               ...
```



If

- *shared* has the same initial value in both executions
- the two executions perform the “same” modifications

(3) the modifications commute

then *shared* has the same final value in both executions

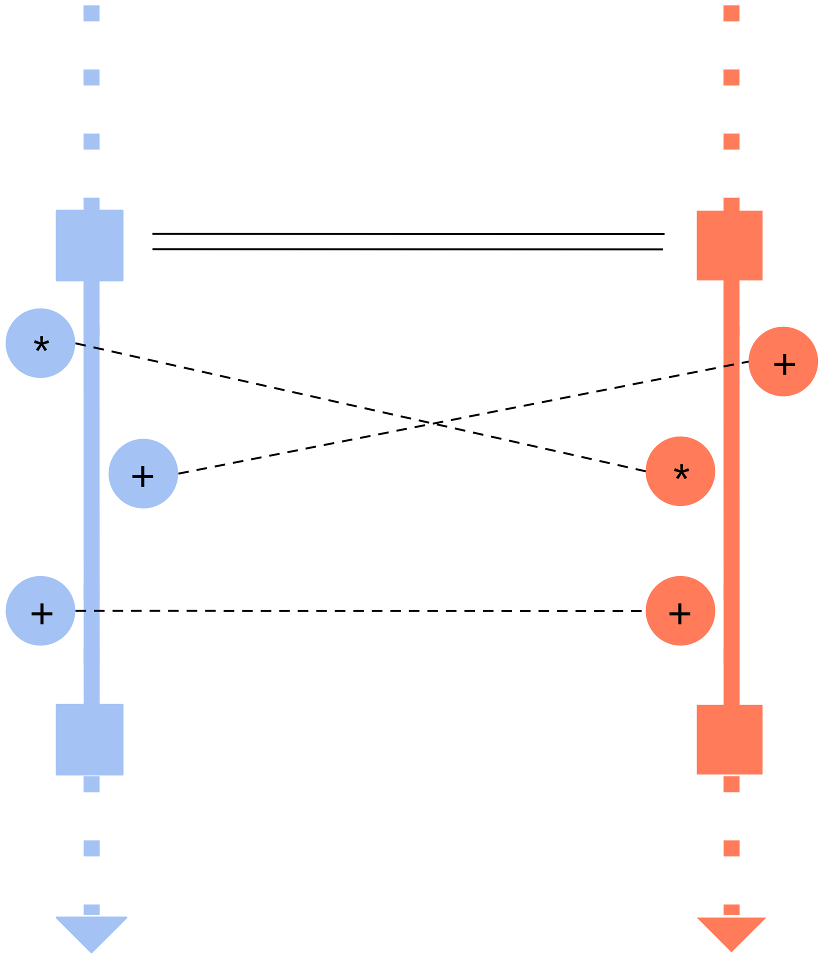
Basic Solution

```
            shared = I
atomic:                  ||  atomic:
   shared *= ...         ||     shared += ...

atomic:                  ||
   shared += ...         ||
                ...
```

If

- *shared* has the same initial value in both executions
- the two executions perform the “same” modifications
- the modifications commute

then *shared* has the same final value in both executions



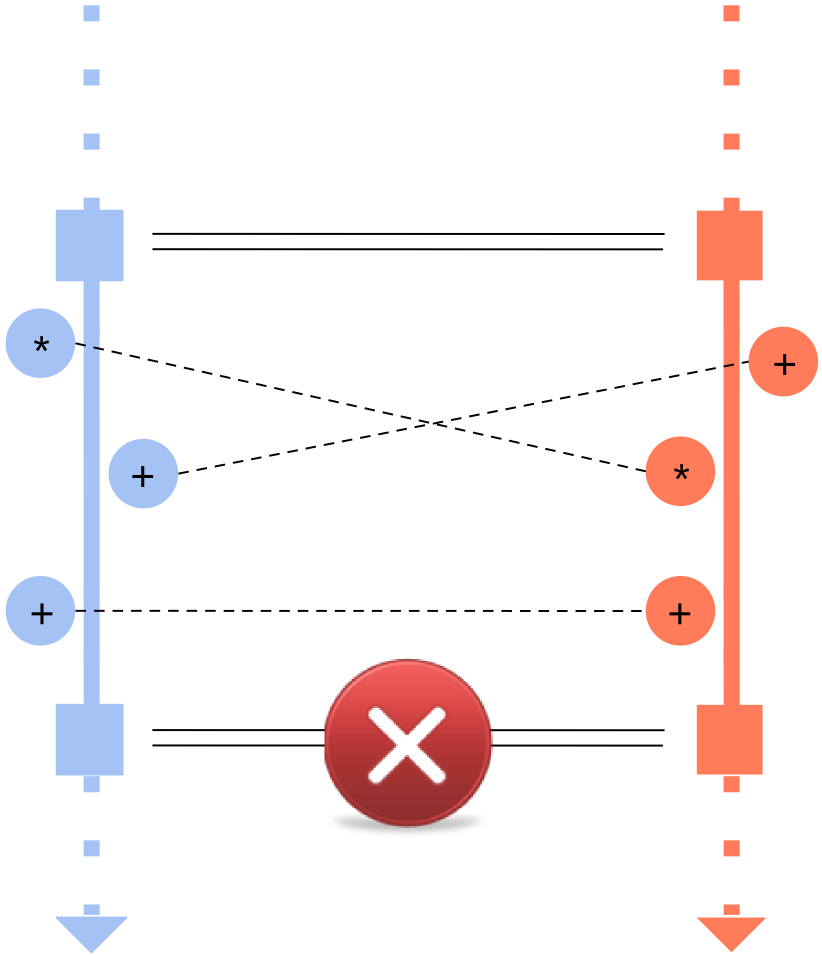
Basic Solution

```
            shared = I
atomic:                    ||  atomic:
   shared *= ...           ||     shared += ...
                           ||
atomic:                    ||
   shared += ...           ||
                          ...
```

If

- *shared* has the same initial value in both executions
- the two executions perform the “same” modifications
- the modifications commute

then *shared* has the same final value in both executions



Back to Program Verification

Verification Approach

Verification Approach

Based on **Concurrent Separation Logic** (CSL)

Verification Approach

Based on **Concurrent Separation Logic** (CSL)

- Extension of Hoare Logic to concurrent heap-manipulating programs

Verification Approach

Based on **Concurrent Separation Logic** (CSL)

- Extension of Hoare Logic to concurrent heap-manipulating programs
- Uses the notion of **resource ownership** (e.g., read/write permission)

Verification Approach

Based on **Concurrent Separation Logic** (CSL)

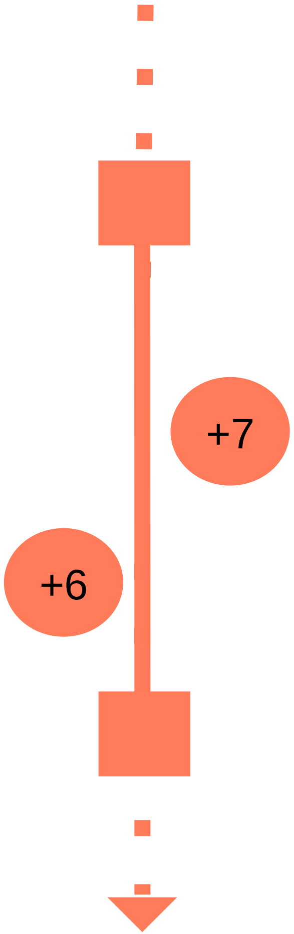
- Extension of Hoare Logic to concurrent heap-manipulating programs
- Uses the notion of **resource ownership** (e.g., read/write permission)
- Associates **resource invariants** with shared memory

Verification Approach

Based on **Concurrent Separation Logic** (CSL)

- Extension of Hoare Logic to concurrent heap-manipulating programs
- Uses the notion of **resource ownership** (e.g., read/write permission)
- Associates **resource invariants** with shared memory

```
                shared = l
                  share
while i < h:           ||  while j < 100:
  i += 1               ||    j += 1

atomic:                ||  atomic:
  shared += 6          ||    shared += 7
                 unshare
              return shared
```

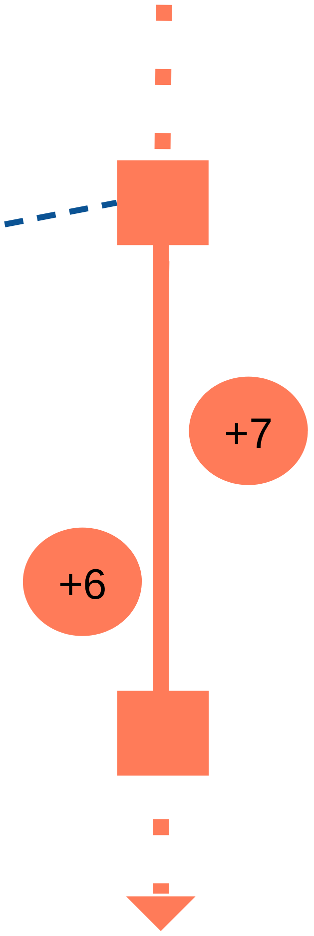


Verification Approach

Based on **Concurrent Separation Logic** (CSL)

- Extension of Hoare Logic to concurrent heap-manipulating programs
- Uses the notion of **resource ownership** (e.g., read/write permission)
- Associates **resource invariants** with shared memory

```
            shared = l
              share

while i < h:        ||  while j < 100:
  i += 1            ||    j += 1
                    ||
atomic:             ||  atomic:
  shared += 6       ||    shared += 7

             unshare
           return shared
```



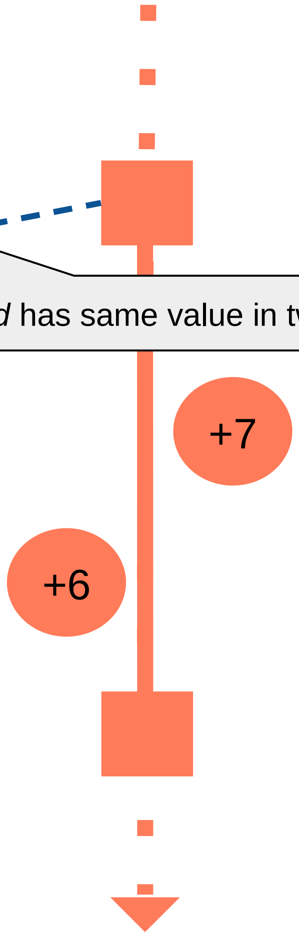
Verification Approach

Based on **Concurrent Separation Logic** (CSL)

- Extension of Hoare Logic to concurrent heap-manipulating programs
- Uses the notion of **resource ownership** (e.g., read/write permission)
- Associates **resource invariants** with shared memory

```
            shared = l
              share
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
                      ||
atomic:               ||  atomic:
  shared += 6         ||    shared += 7
             unshare
           return shared
```

Prove 1) *shared* has same value in two executions

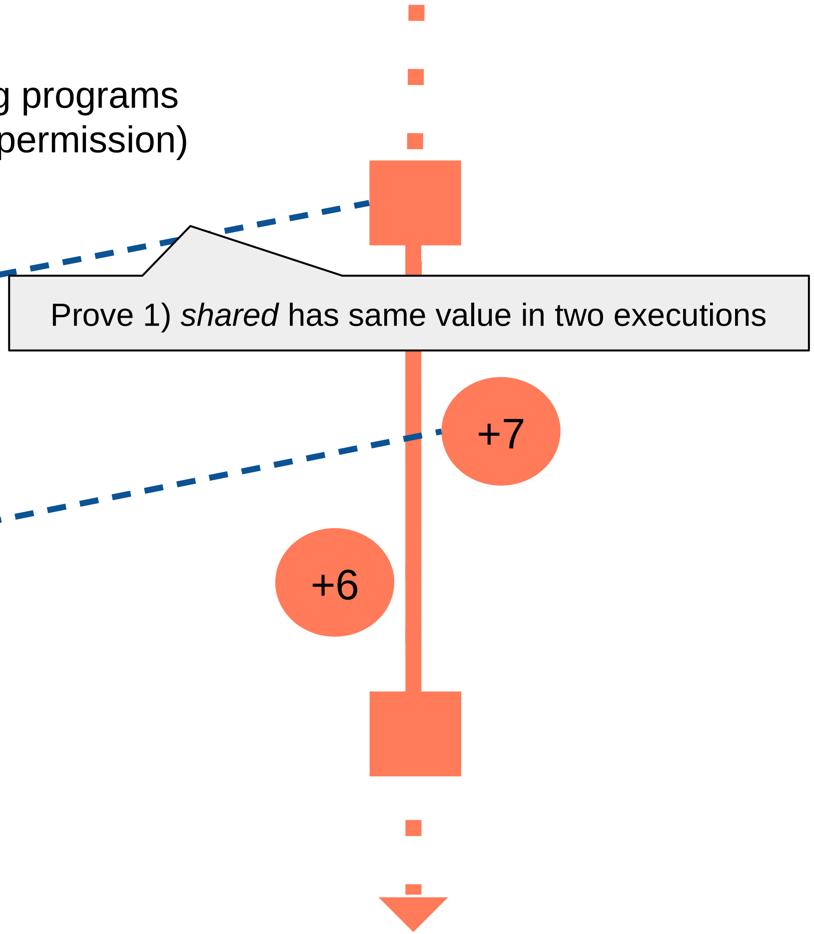


Verification Approach

Based on **Concurrent Separation Logic** (CSL)

- Extension of Hoare Logic to concurrent heap-manipulating programs
- Uses the notion of **resource ownership** (e.g., read/write permission)
- Associates **resource invariants** with shared memory

```
shared = l
share
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
                      ||
atomic:               ||  atomic:
  shared += 6         ||    shared += 7
unshare
return shared
```



Verification Approach

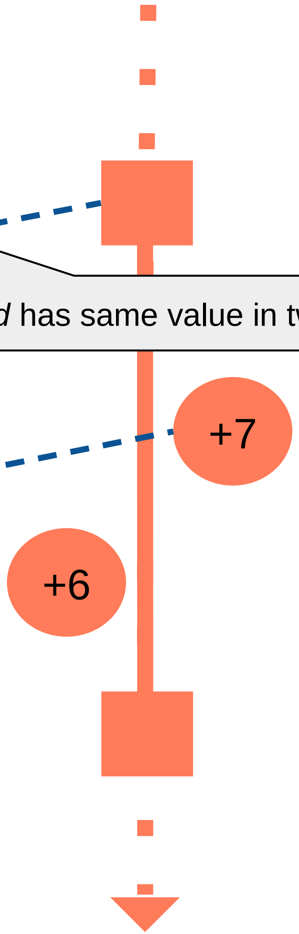
Based on **Concurrent Separation Logic** (CSL)

- Extension of Hoare Logic to concurrent heap-manipulating programs
- Uses the notion of **resource ownership** (e.g., read/write permission)
- Associates **resource invariants** with shared memory

```
shared = l
share

while i < h:            ||  while j < 100:
  i += 1                ||    j += 1
                        ||
atomic:                 ||  atomic:
  shared += 6           ||    shared += 7

unshare
return shared
```

Prove 1) *shared* has same value in two executions

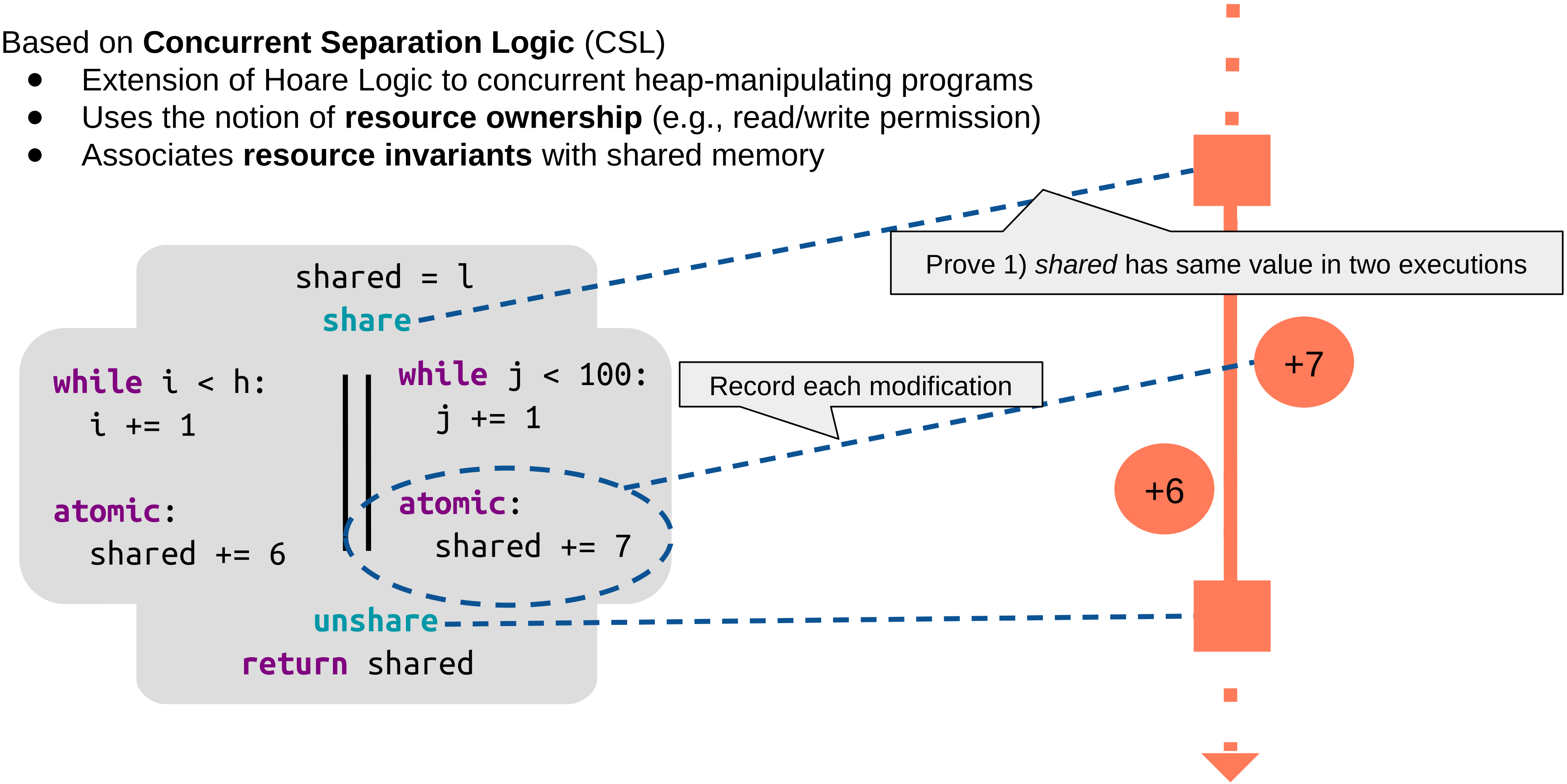
Record each modification



Verification Approach

Based on **Concurrent Separation Logic** (CSL)

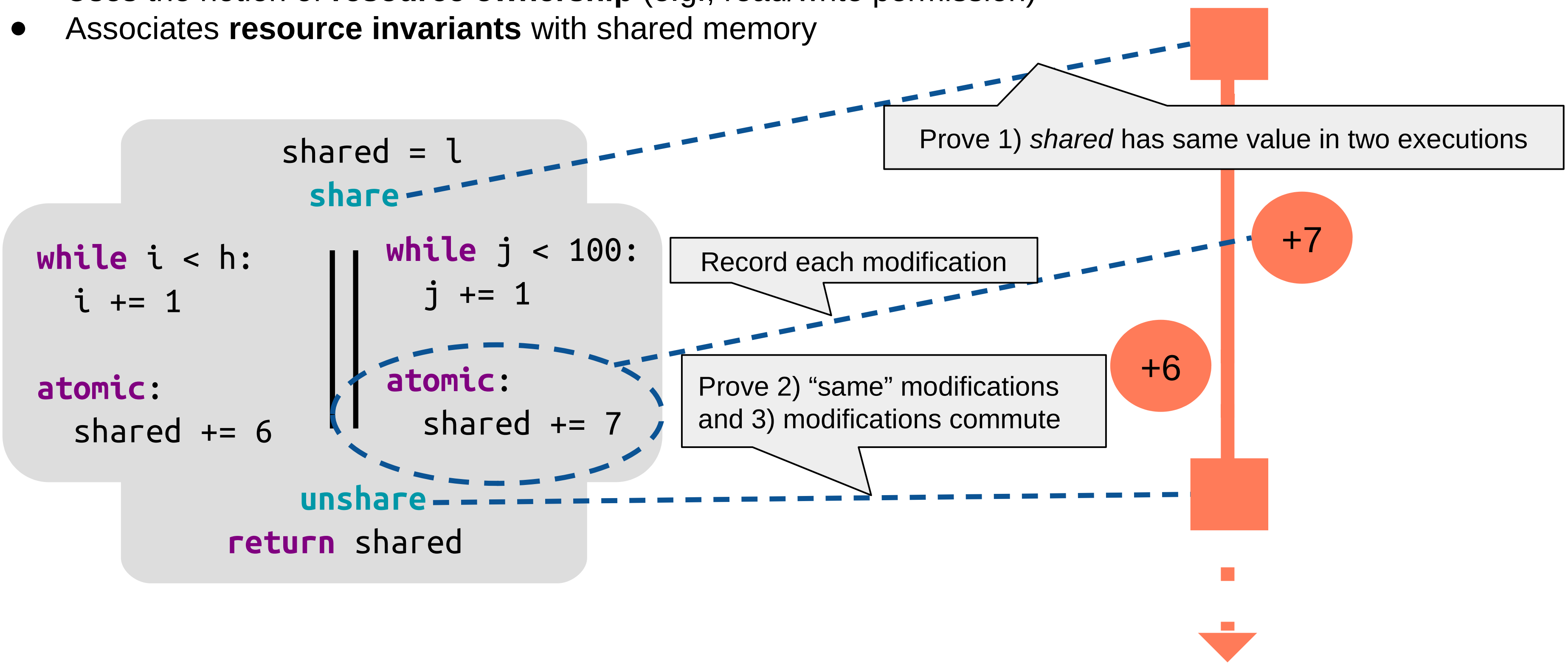
- Extension of Hoare Logic to concurrent heap-manipulating programs
- Uses the notion of **resource ownership** (e.g., read/write permission)
- Associates **resource invariants** with shared memory



Verification Approach

Based on **Concurrent Separation Logic** (CSL)

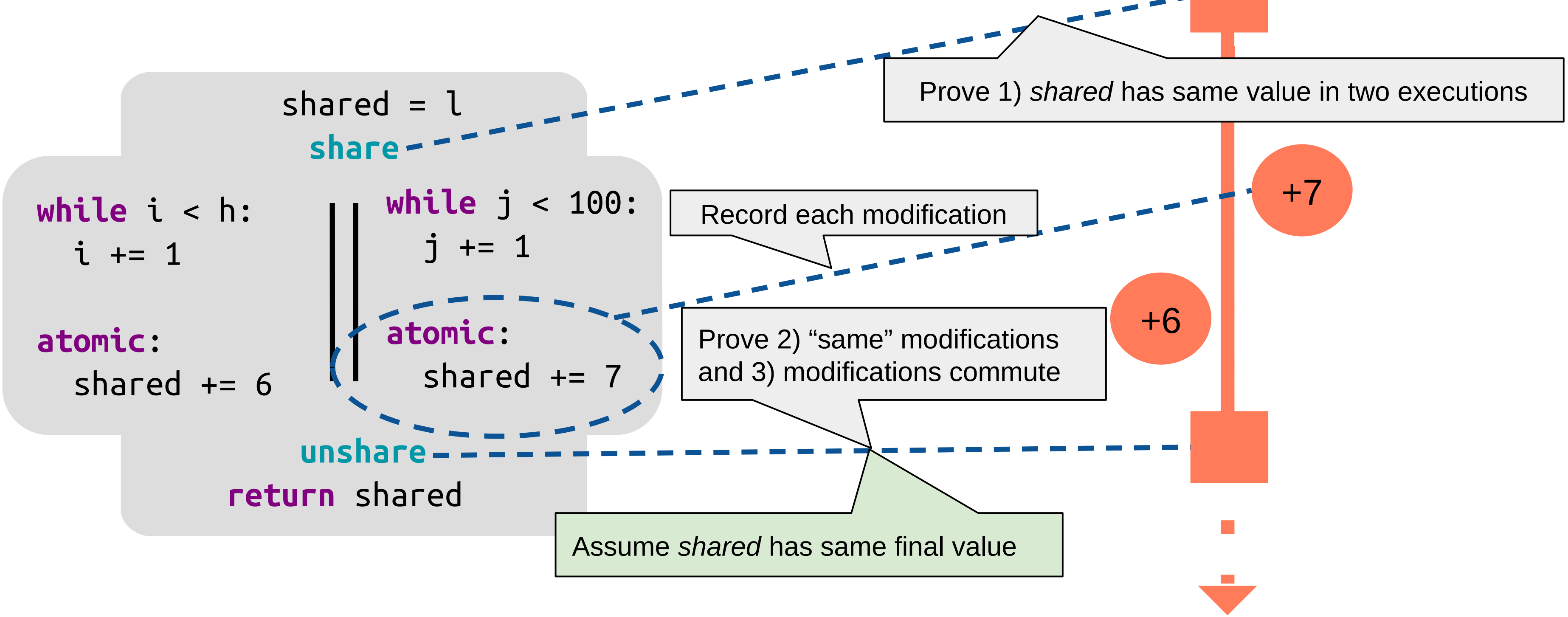
- Extension of Hoare Logic to concurrent heap-manipulating programs
- Uses the notion of **resource ownership** (e.g., read/write permission)
- Associates **resource invariants** with shared memory



Verification Approach

Based on **Concurrent Separation Logic** (CSL)

- Extension of Hoare Logic to concurrent heap-manipulating programs
- Uses the notion of **resource ownership** (e.g., read/write permission)
- Associates **resource invariants** with shared memory



Verification Approach

```
shared = l

atomic:           ||  atomic:
  shared += 6     ||    shared += 7

return shared
```

Verification Approach

l has the same value in the two executions

```
{low(l)}
shared = l

atomic:              ||   atomic:
  shared += 6        ||     shared += 7

return shared
```

Verification Approach

l has the same value in the two executions

```
{low(l)}
shared = l

atomic:              ||   atomic:
  shared += 6        ||     shared += 7

return shared
{low(result)}
```

Verification Approach

l has the same value in the two executions

shared has the same value in the two executions (1)

```
{low(l)}
shared = l
{low(shared)}

atomic:            ||  atomic:
  shared += 6      ||    shared += 7

return shared
{low(result)}
```

Verification Approach

l has the same value in the two executions

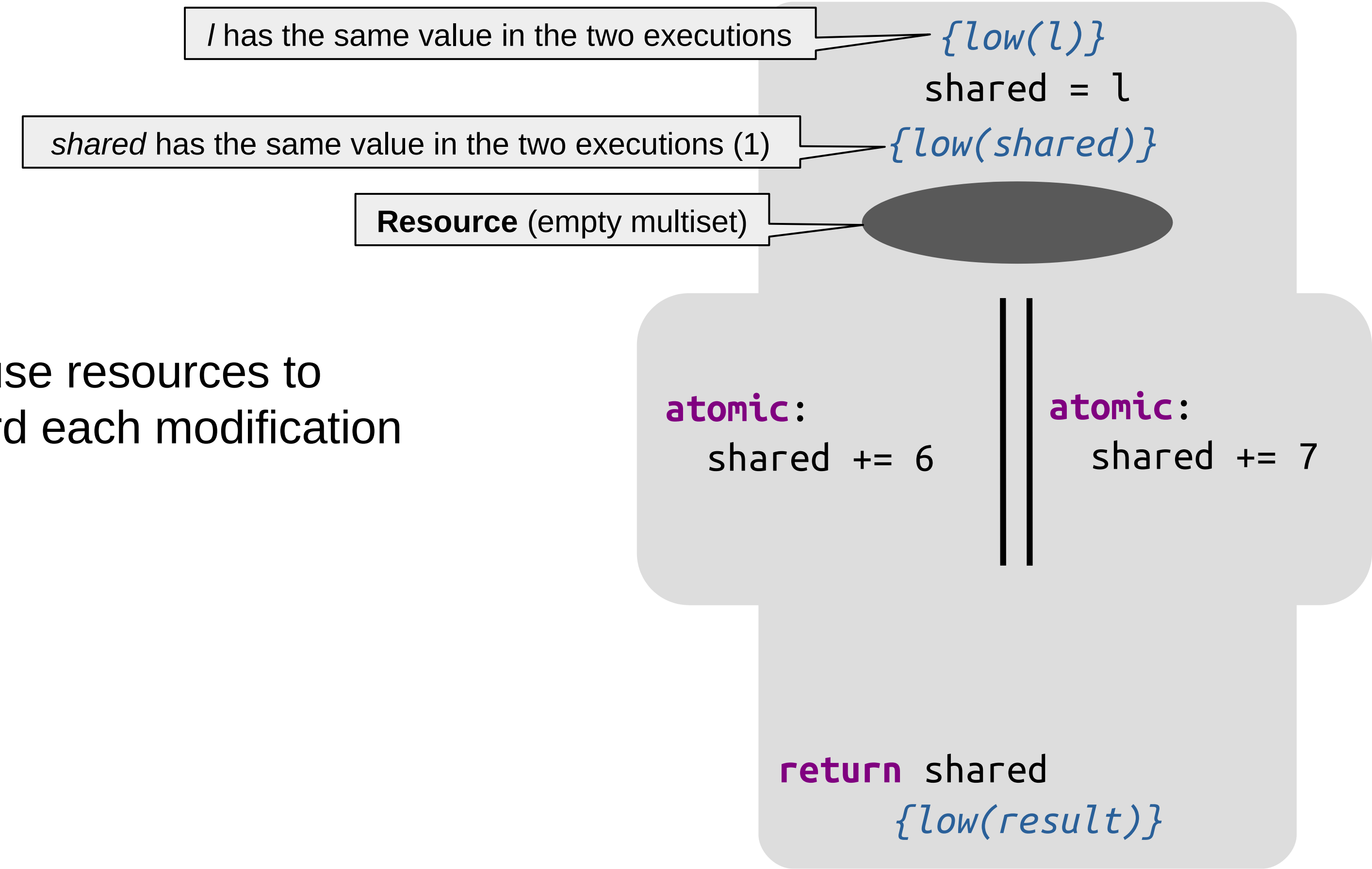
shared has the same value in the two executions (1)

We use resources to record each modification

```
{low(l)}
shared = l
{low(shared)}

atomic:              ||  atomic:
  shared += 6        ||    shared += 7

return shared
{low(result)}
```

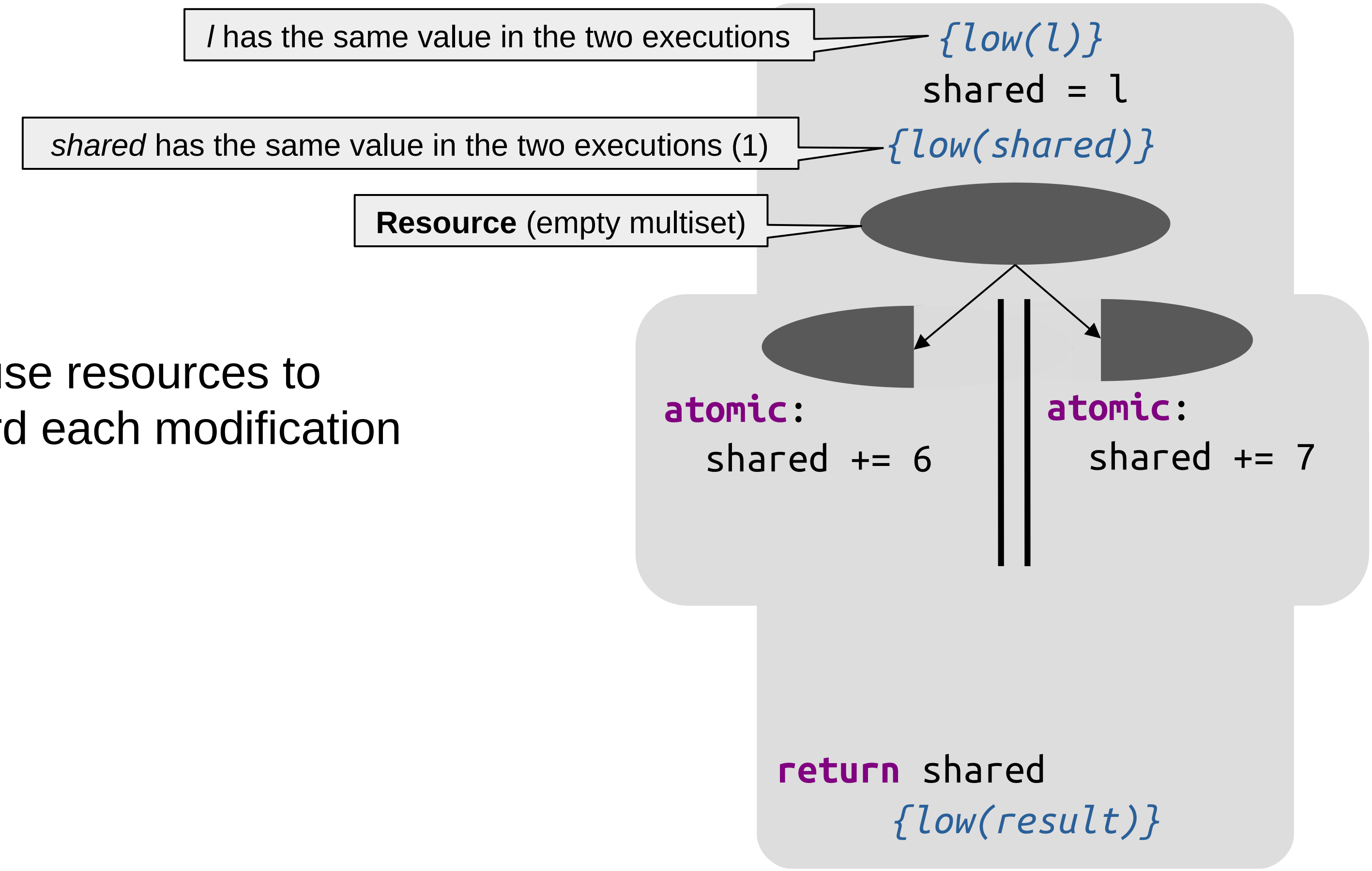
Verification Approach

We use resources to record each modification



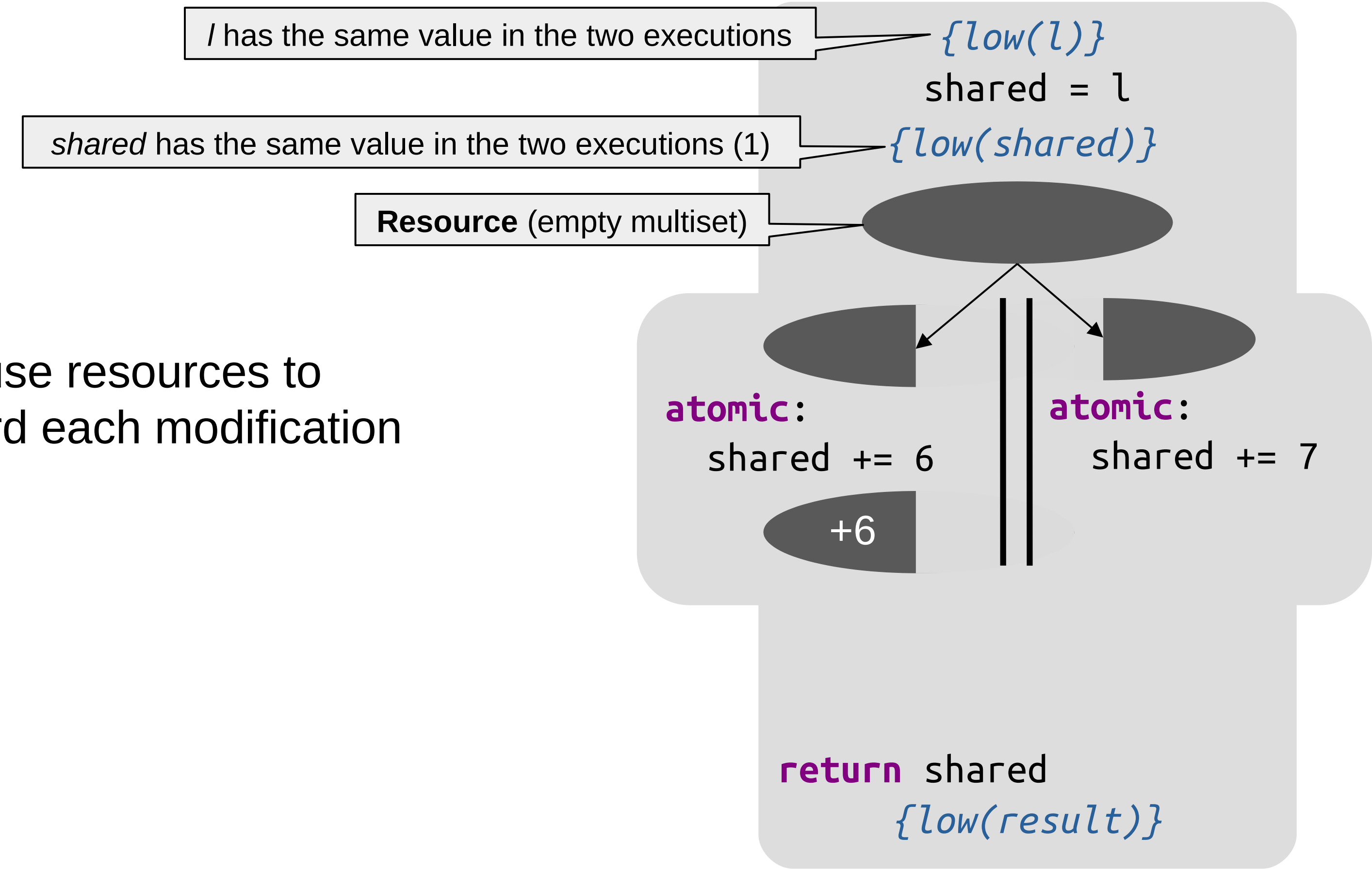
Verification Approach

We use resources to record each modification



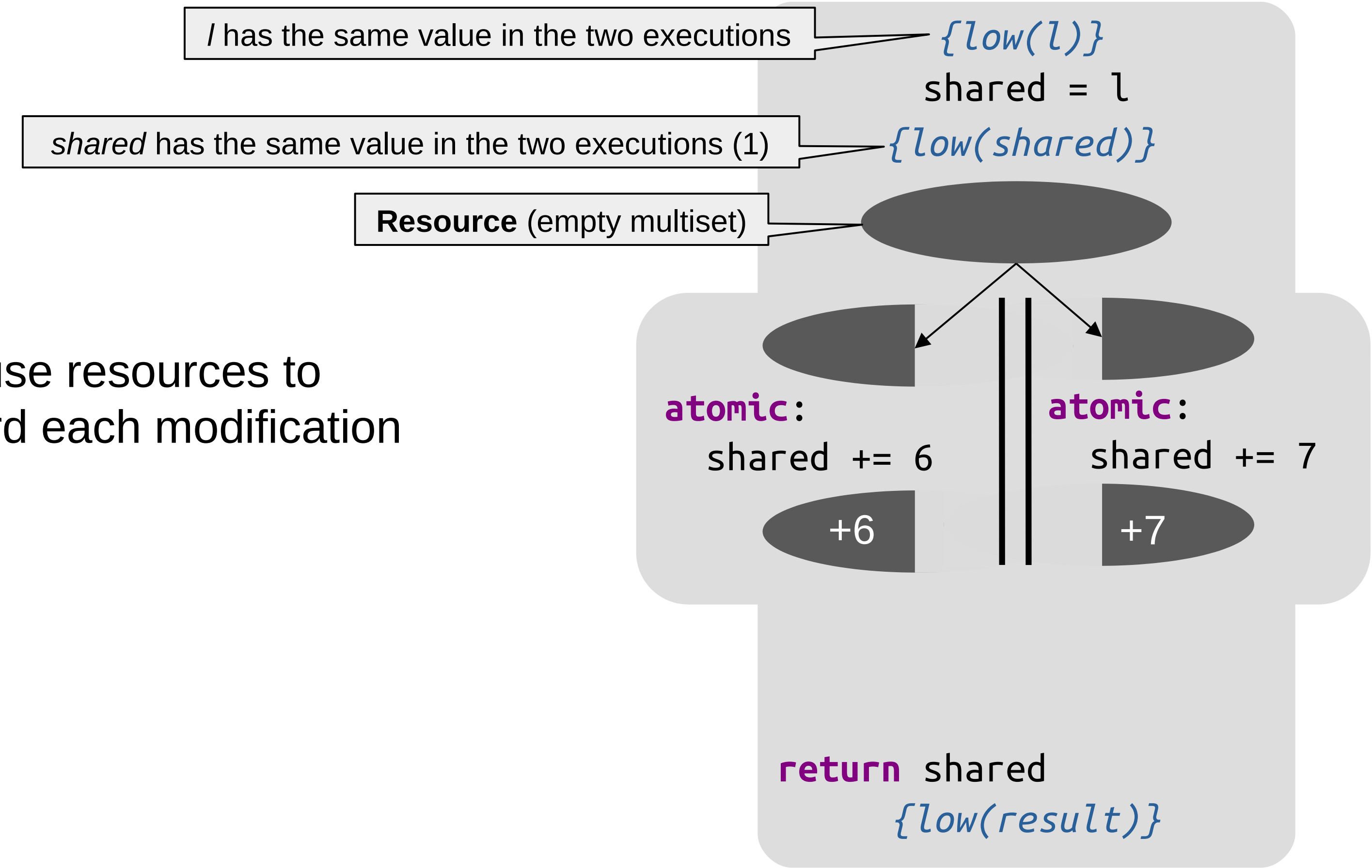
Verification Approach

We use resources to record each modification



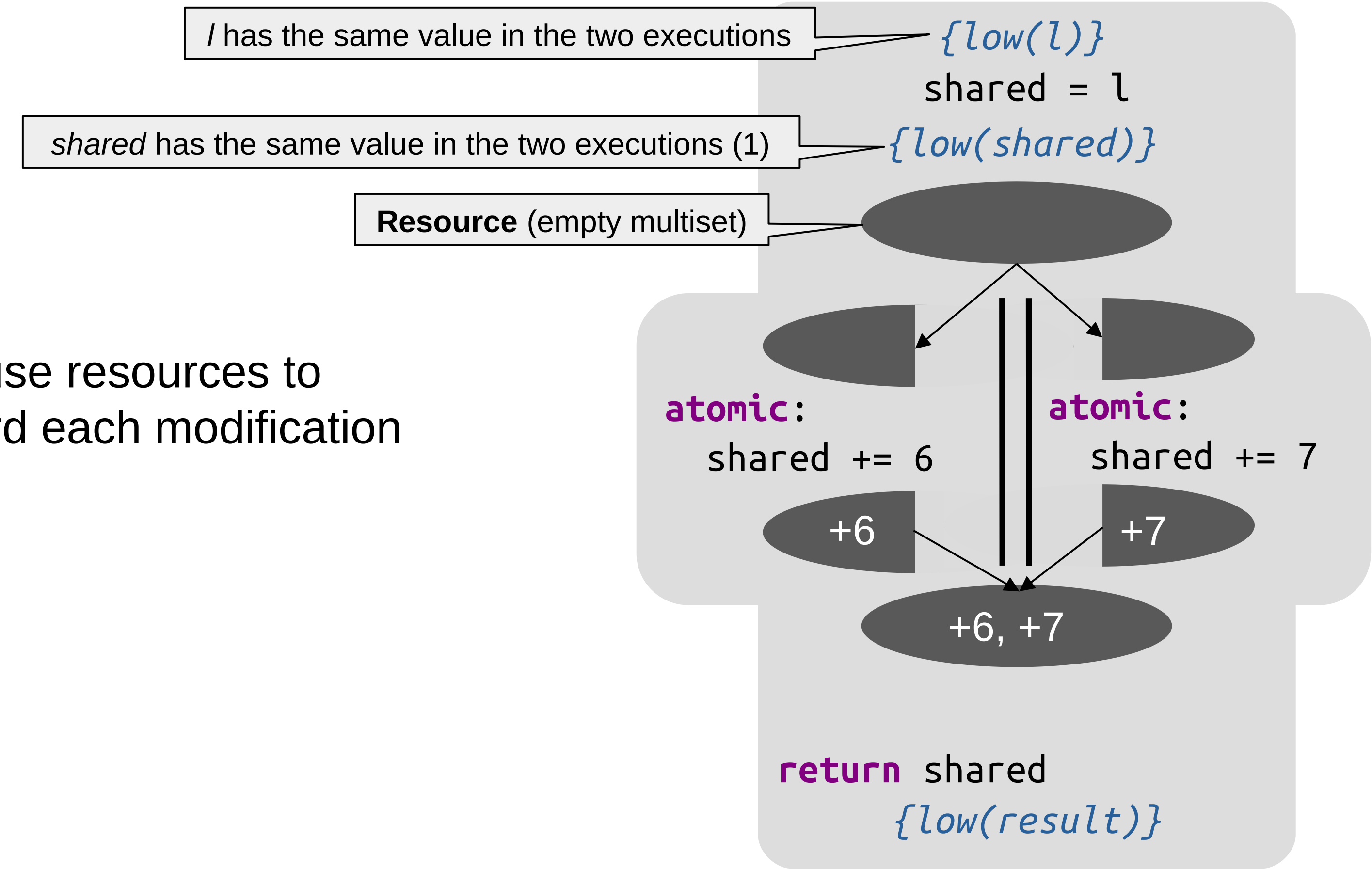
Verification Approach

We use resources to record each modification



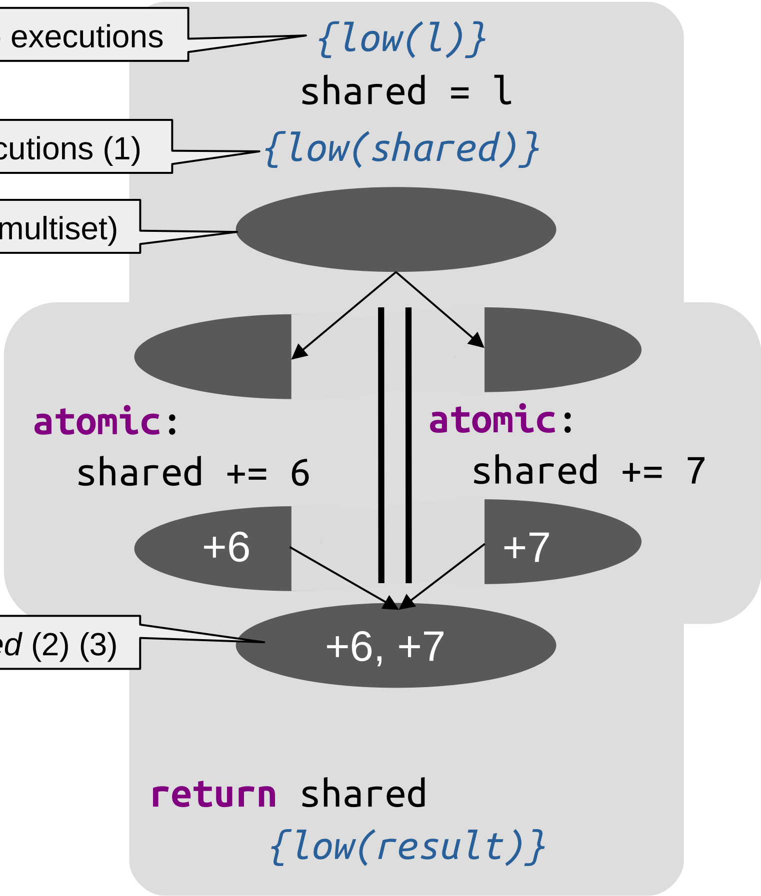
Verification Approach

We use resources to
record each modification



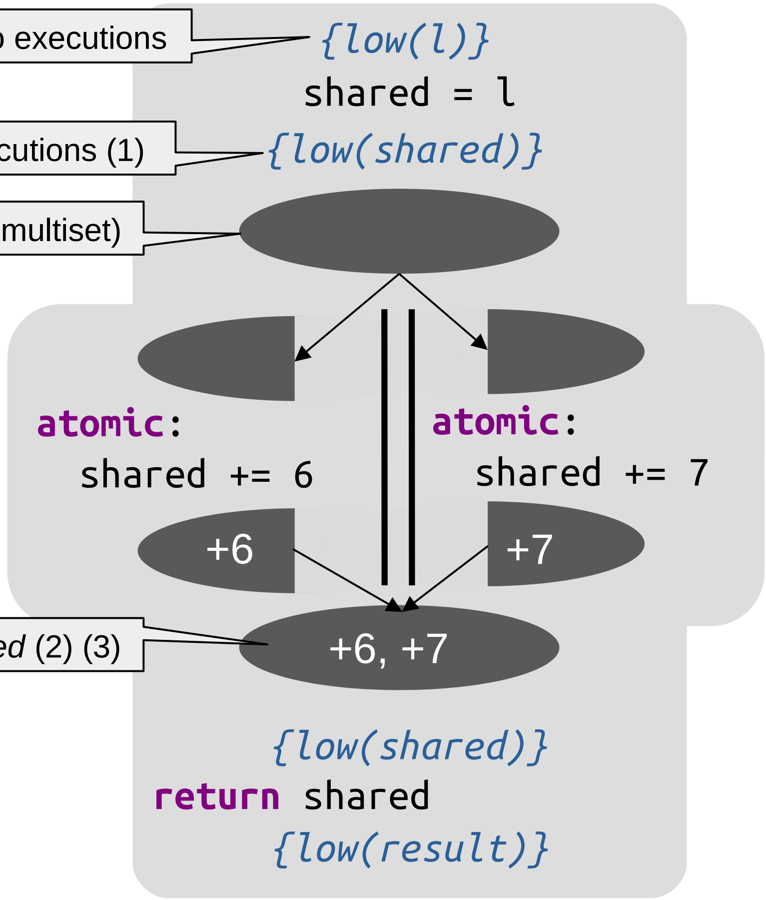
Verification Approach

We use resources to
record each modification



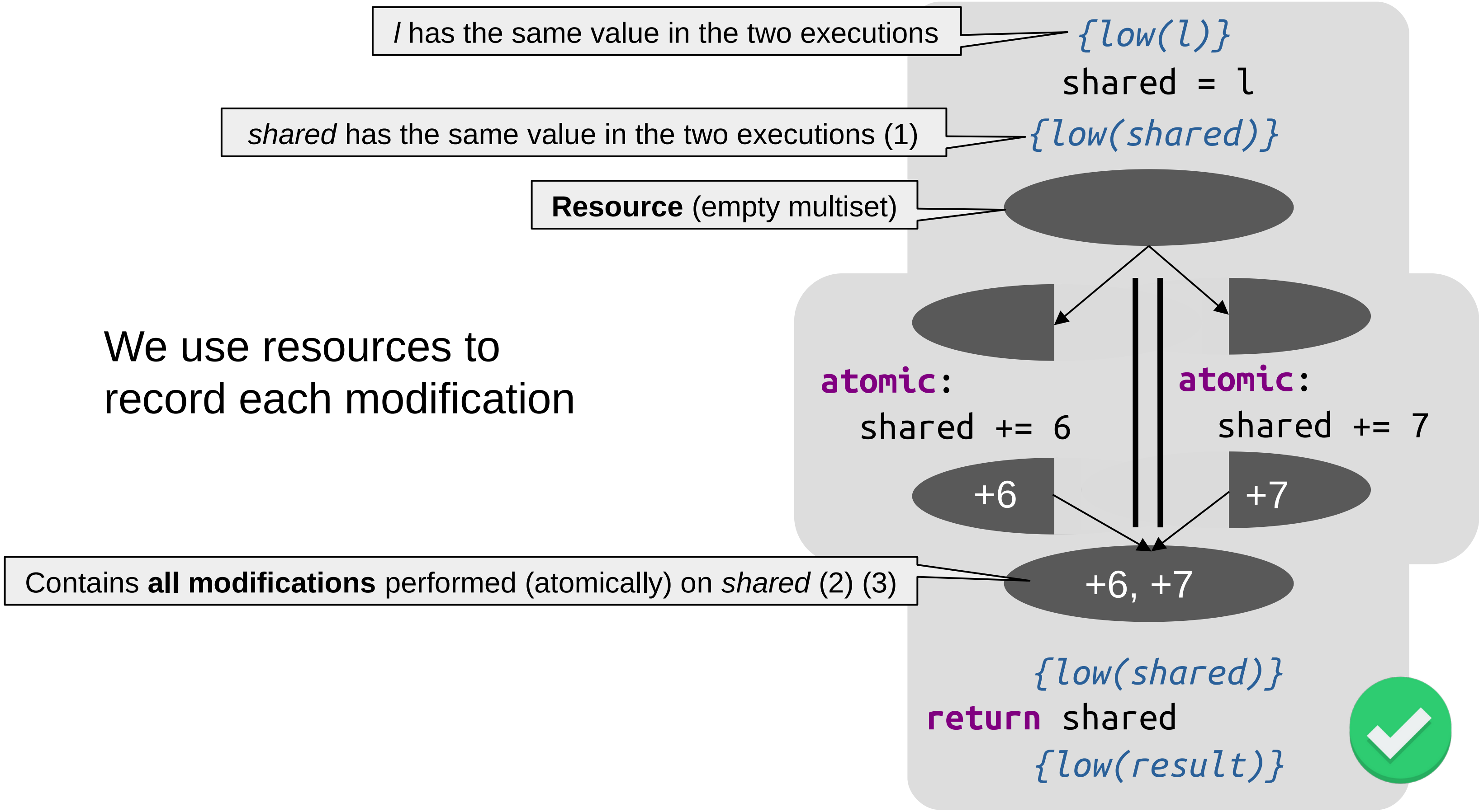
Verification Approach

We use resources to
record each modification



Verification Approach

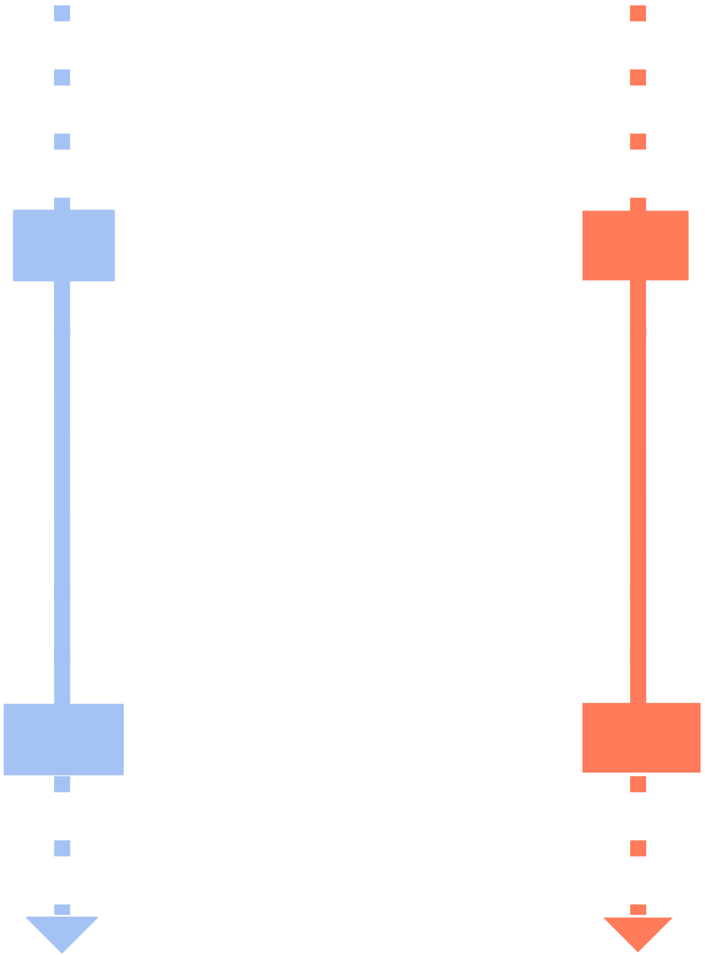
We use resources to record each modification



We can do better.

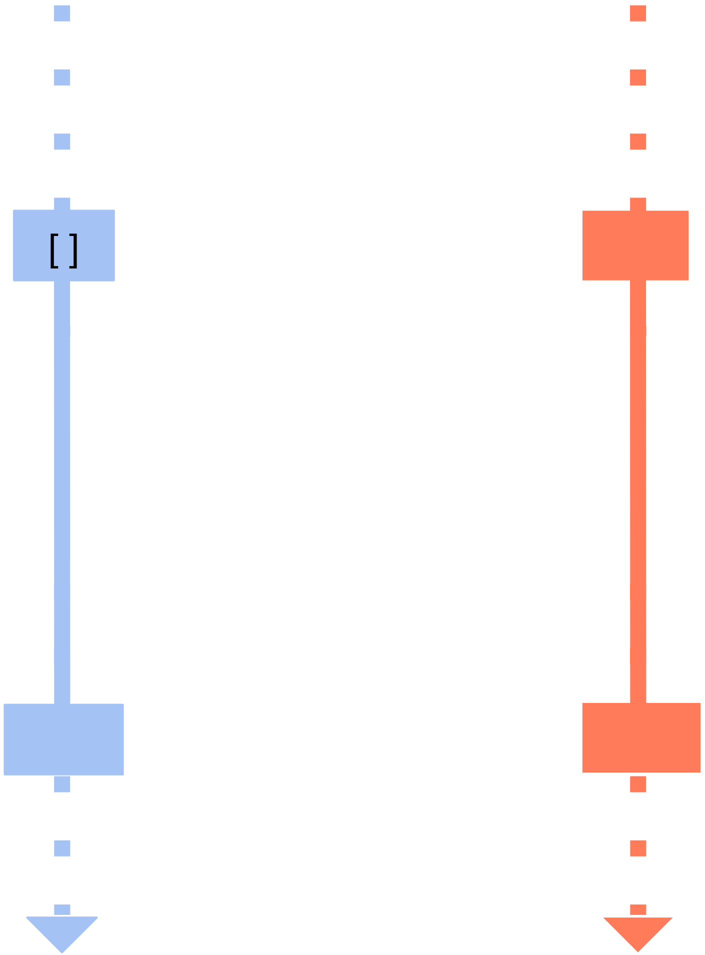
The Limits of Commutativity

```
shared = new List()

while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
atomic:               ||   atomic:
  shared.add(6)       ||     shared.add(7)

return sort(shared)
```



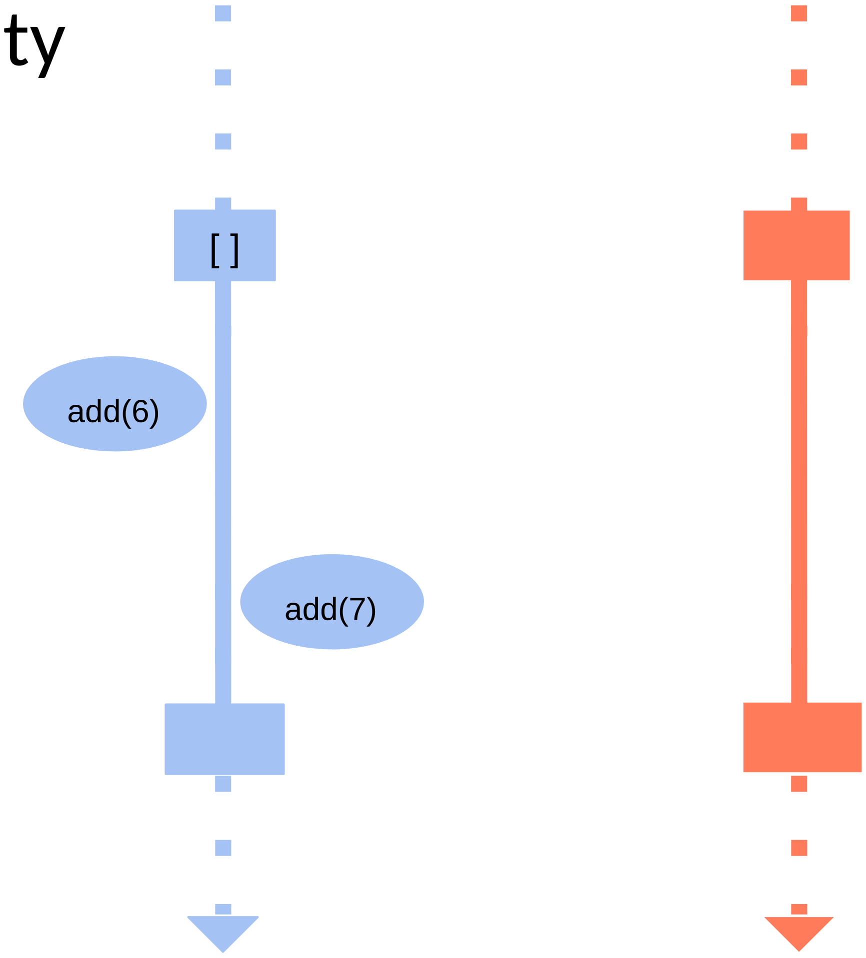
The Limits of Commutativity

```
shared = new List()

while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
atomic:               ||   atomic:
  shared.add(6)       ||     shared.add(7)

return sort(shared)
```



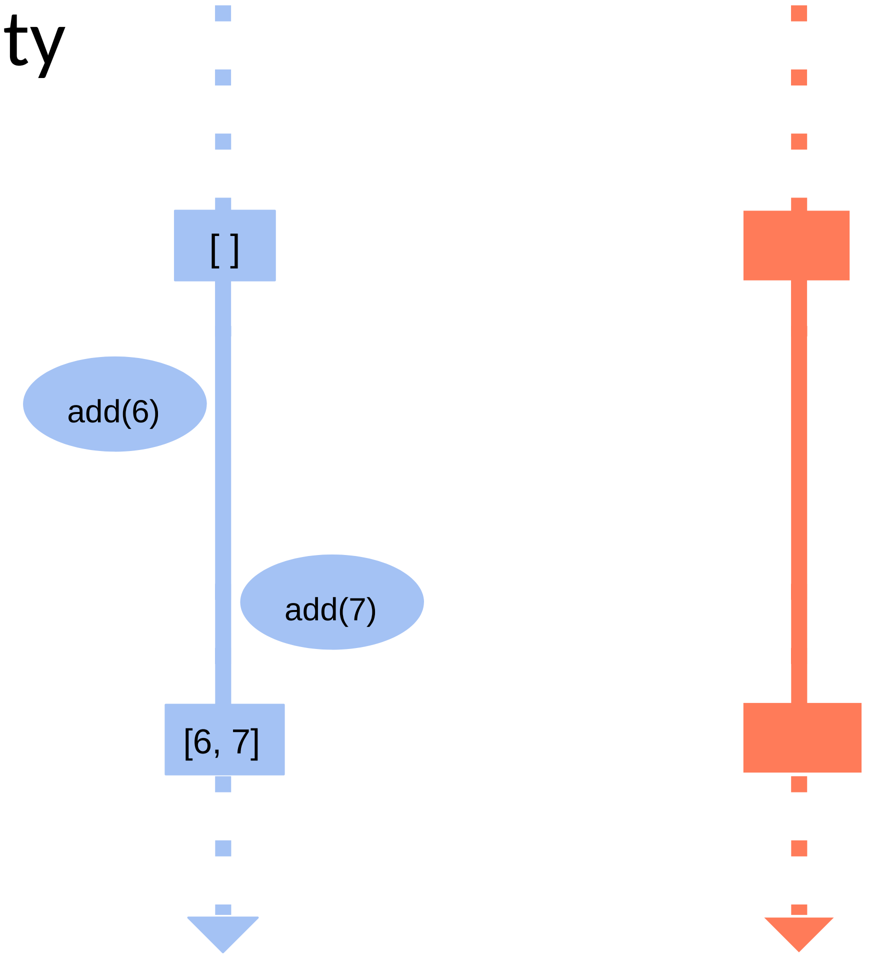
The Limits of Commutativity

```
shared = new List()

while i < h:            ||    while j < 100:
  i += 1                        j += 1
atomic:                       atomic:
  shared.add(6)                 shared.add(7)

return sort(shared)
```



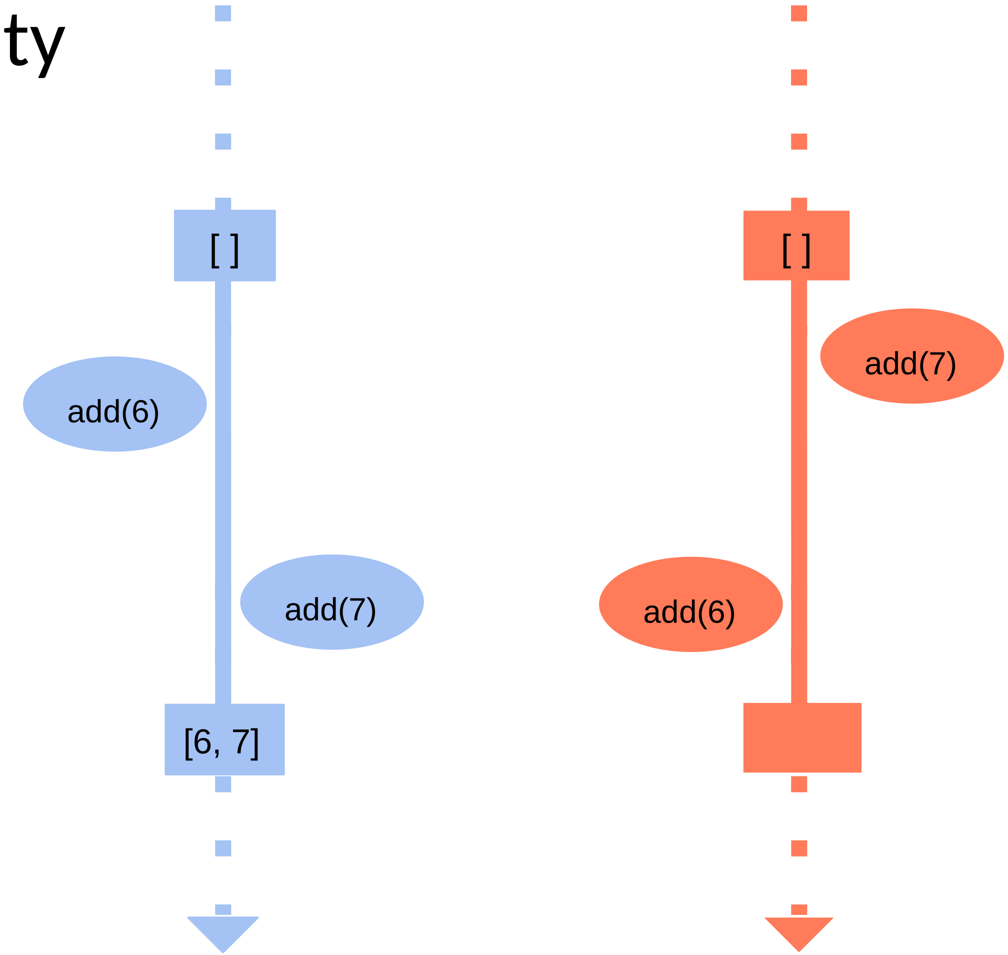
The Limits of Commutativity

```
shared = new List()

while i < h:          ||    while j < 100:
  i += 1              ||      j += 1
atomic:               ||    atomic:
  shared.add(6)       ||      shared.add(7)

return sort(shared)
```



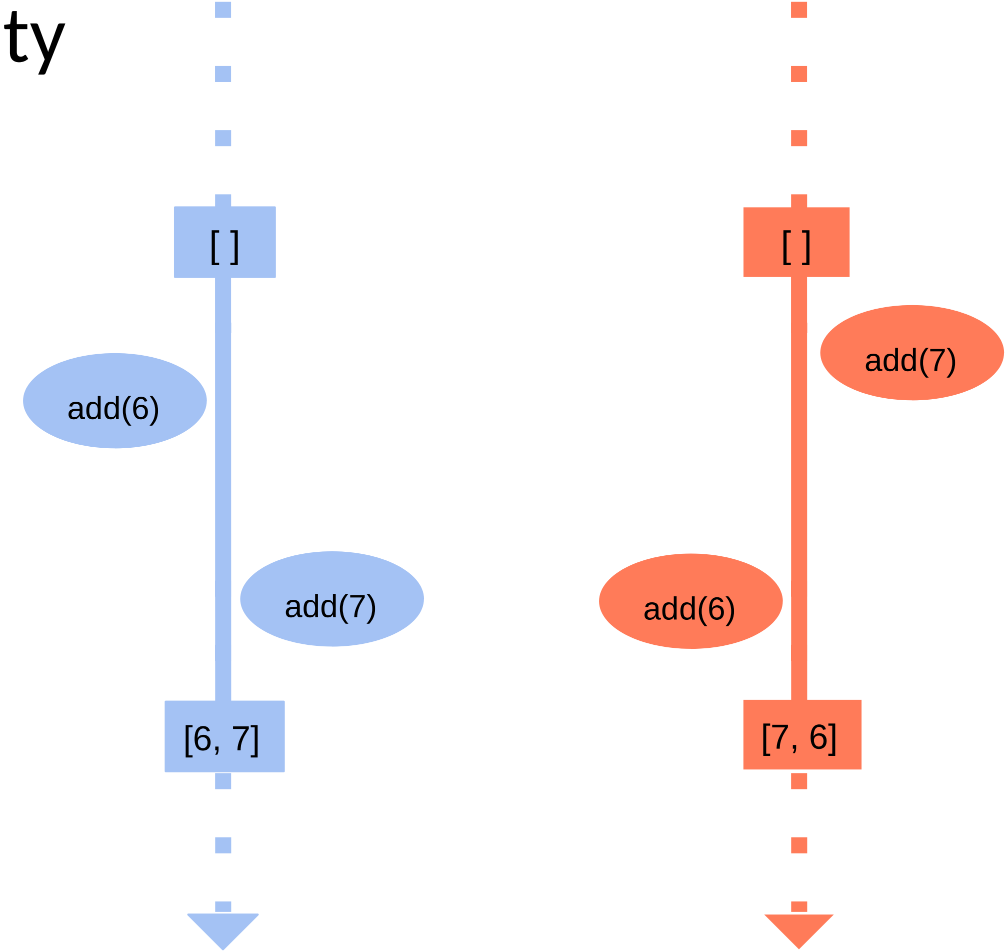
The Limits of Commutativity

```
          shared = new List()
while i < h:            ||  while j < 100:
  i += 1                ||    j += 1
atomic:                 ||  atomic:
  shared.add(6)         ||    shared.add(7)

        return sort(shared)
```



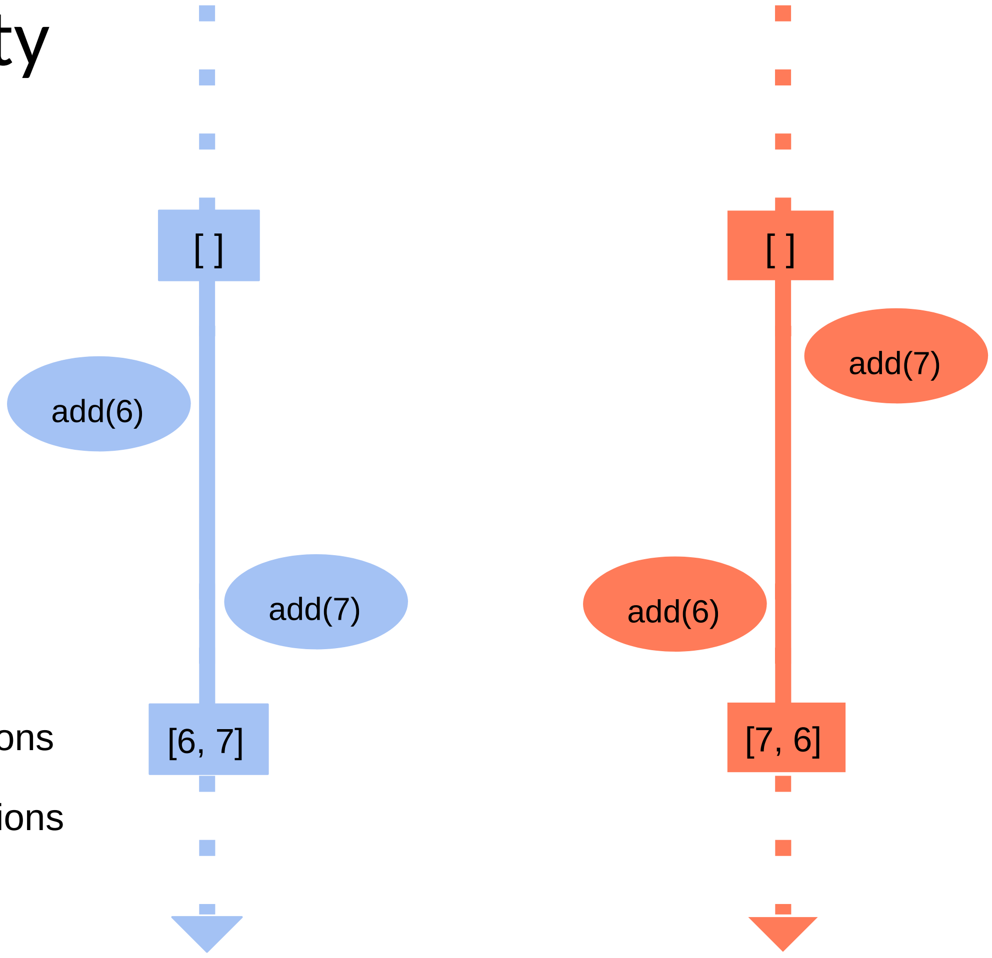
The Limits of Commutativity

```
shared = new List()

while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
atomic:               ||   atomic:
  shared.add(6)       ||     shared.add(7)

return sort(shared)
```



The Limits of Commutativity

```
           shared = new List()
while i < h:        ||  while j < 100:
  i += 1            ||    j += 1
atomic:             ||  atomic:
  shared.add(6)     ||    shared.add(7)
           return sort(shared)
```



If

(1) *shared* has the same initial value in both executions

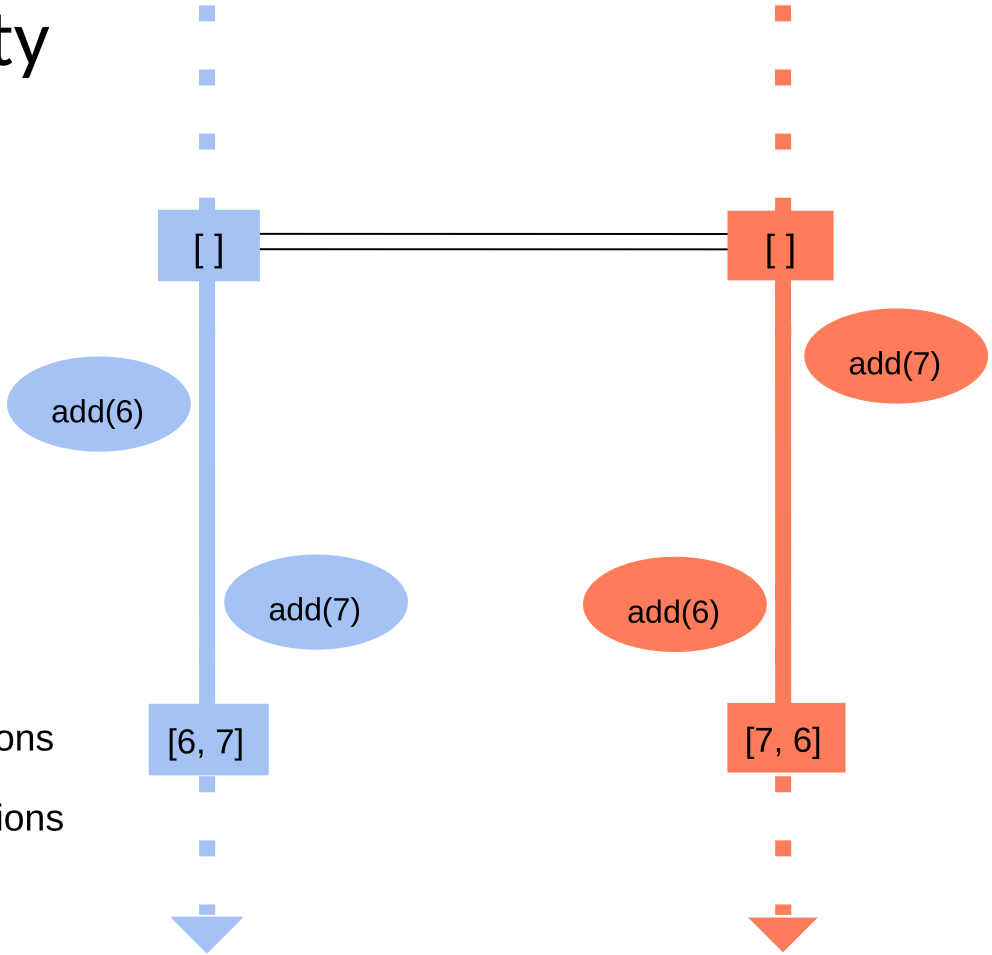
(2) the two executions perform the “same” modifications

(3) the modifications commute

then *shared* has the same final value in both executions

The Limits of Commutativity

```
          shared = new List()
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.add(6)       ||    shared.add(7)

          return sort(shared)
```



If

✅ *shared* has the same initial value in both executions

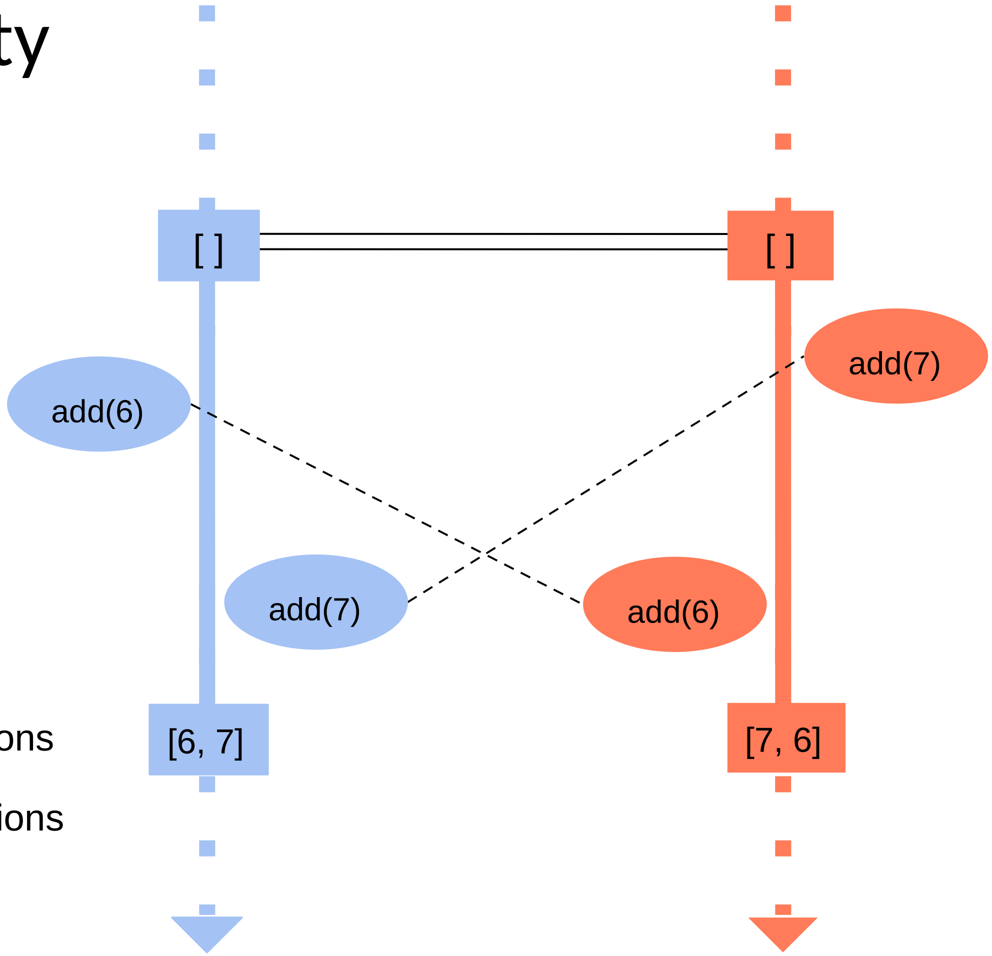
(2) the two executions perform the “same” modifications

(3) the modifications commute

then *shared* has the same final value in both executions

The Limits of Commutativity

```
            shared = new List()

while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.add(6)       ||    shared.add(7)

          return sort(shared)
```



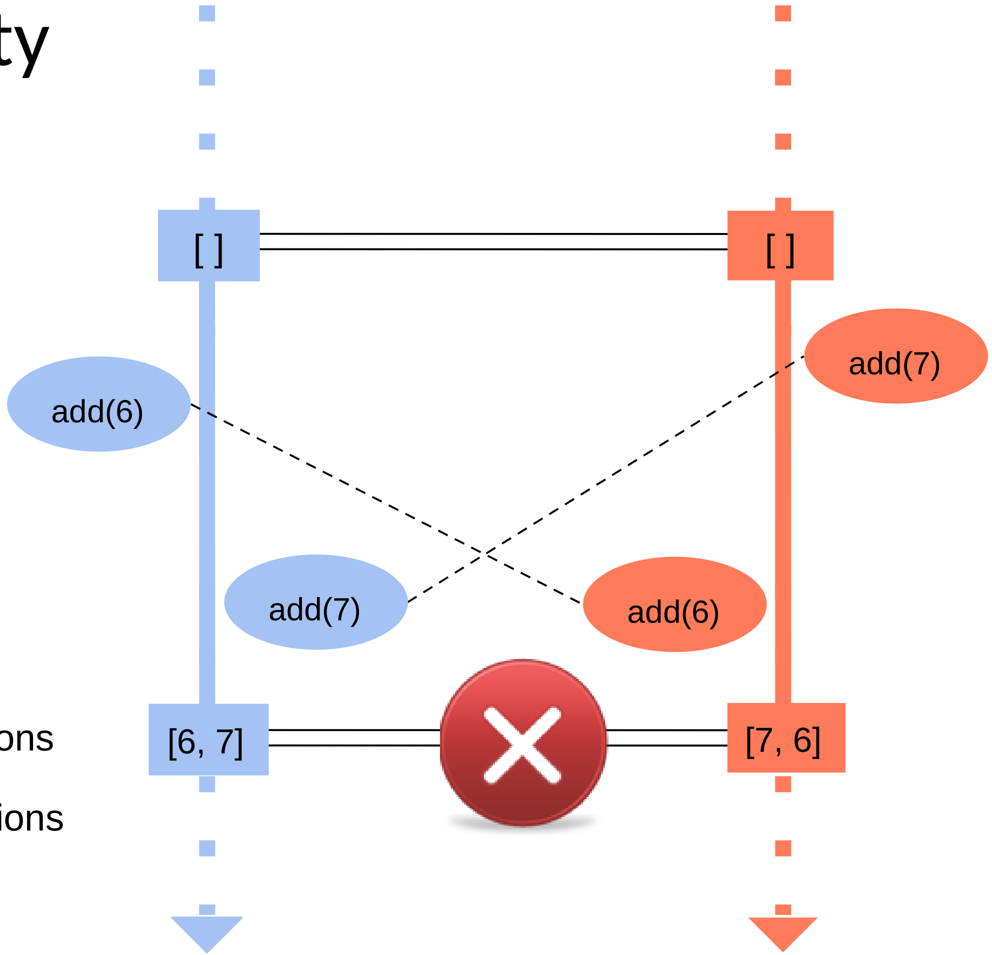
If

- ✔ *shared* has the same initial value in both executions
- ✔ the two executions perform the “same” modifications
- (3) the modifications commute

then *shared* has the same final value in both executions

The Limits of Commutativity

```
shared = new List()
while i < h:              ||   while j < 100:
  i += 1                  ||     j += 1
atomic:                   ||   atomic:
  shared.add(6)           ||     shared.add(7)
return sort(shared)
```



If

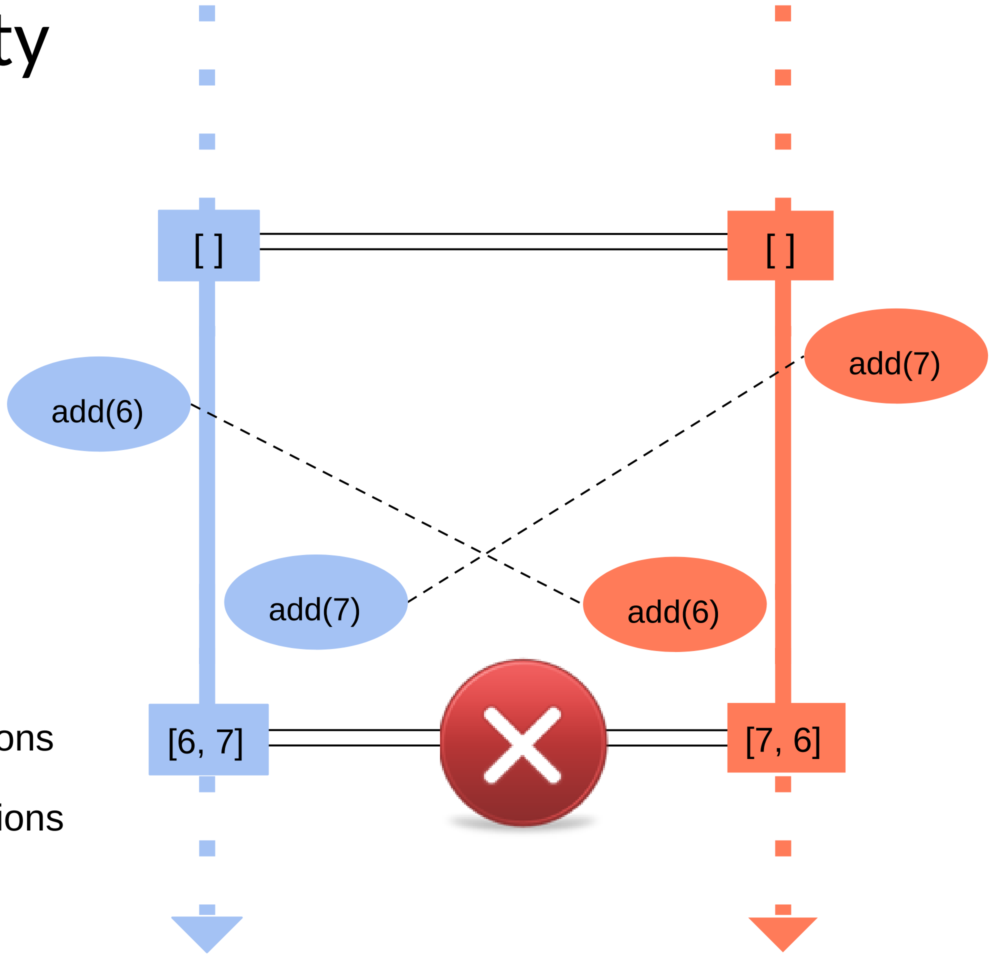
- *shared* has the same initial value in both executions
- the two executions perform the “same” modifications
- the modifications commute

then *shared* has the same final value in both executions

The Limits of Commutativity

```
shared = new List()

while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
atomic:               ||   atomic:
  shared.add(6)       ||     shared.add(7)

return sort(shared)
```

Secure



If

- *shared* has the same initial value in both executions
- the two executions perform the "same" modifications
- the modifications commute

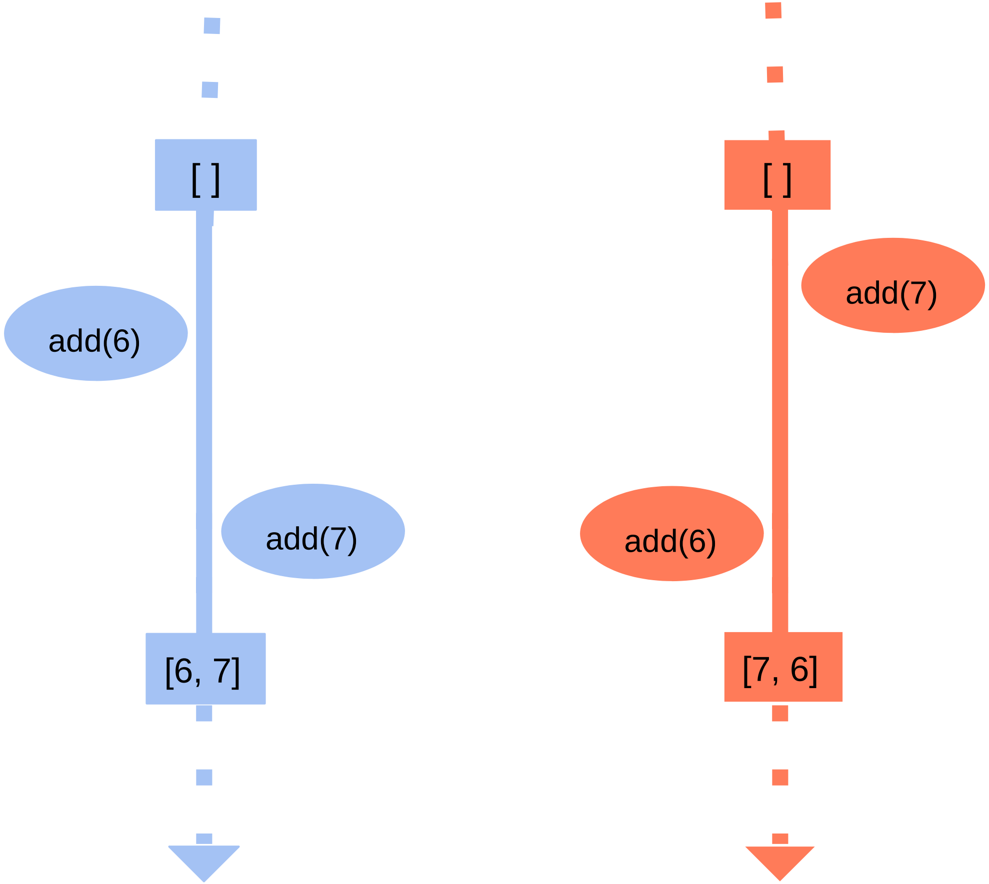
then *shared* has the same final value in both executions

Key Idea

Commutativity **modulo abstraction**

Commutativity Modulo Abstraction ("Abstract Commutativity")

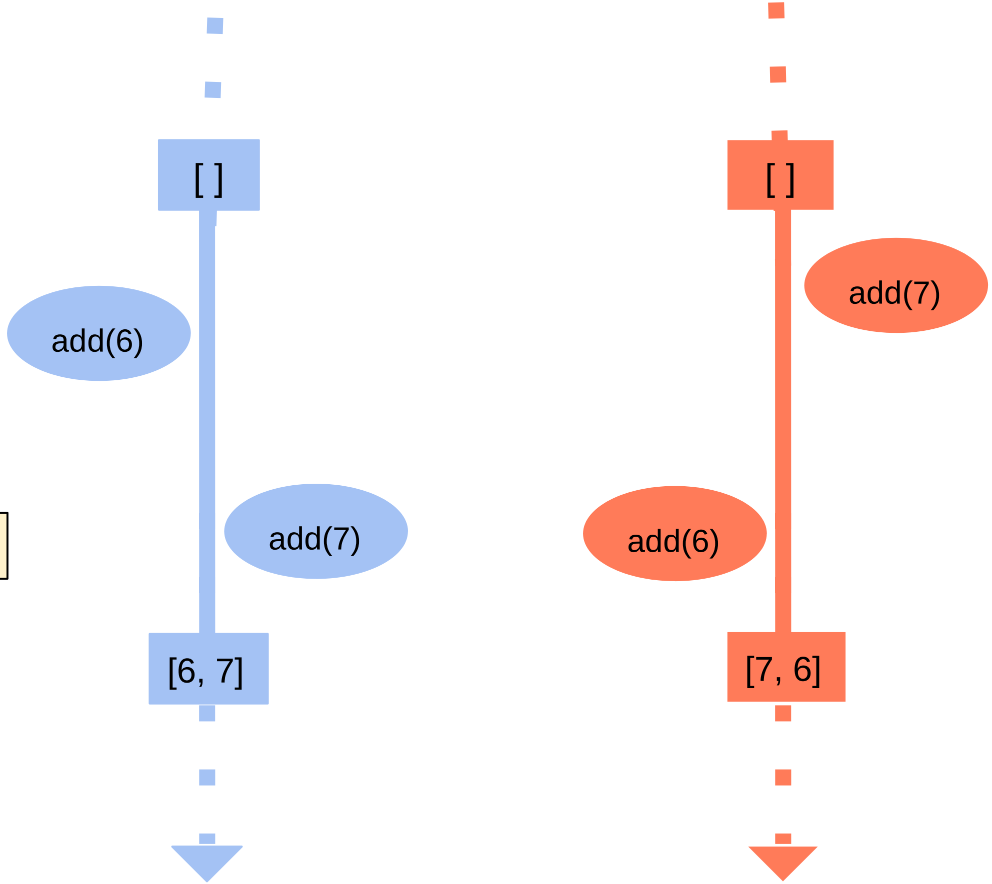
```
shared = new List()

while i < h:            ||    while j < 100:
  i += 1                ||      j += 1
atomic:                 ||    atomic:
  shared.add(6)         ||      shared.add(7)

return sort(shared)
```



Commutativity Modulo Abstraction ("Abstract Commutativity")

```
              shared = new List()
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.add(6)       ||    shared.add(7)
             return sort(shared)
```

Abstraction **α**: list → multiset of elements

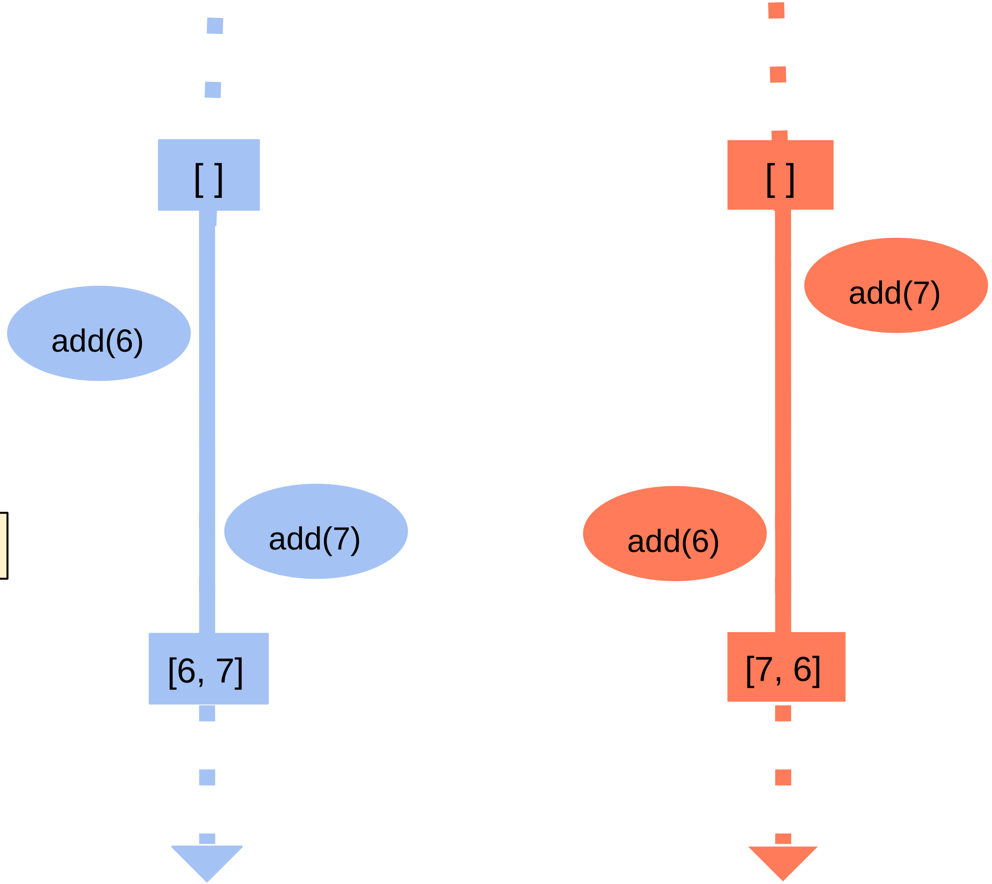


Commutativity Modulo Abstraction ("Abstract Commutativity")

```
              shared = new List()
while i < h:           ||  while j < 100:
  i += 1               ||    j += 1
atomic:                ||  atomic:
  shared.add(6)        ||    shared.add(7)

             return sort(shared)
```

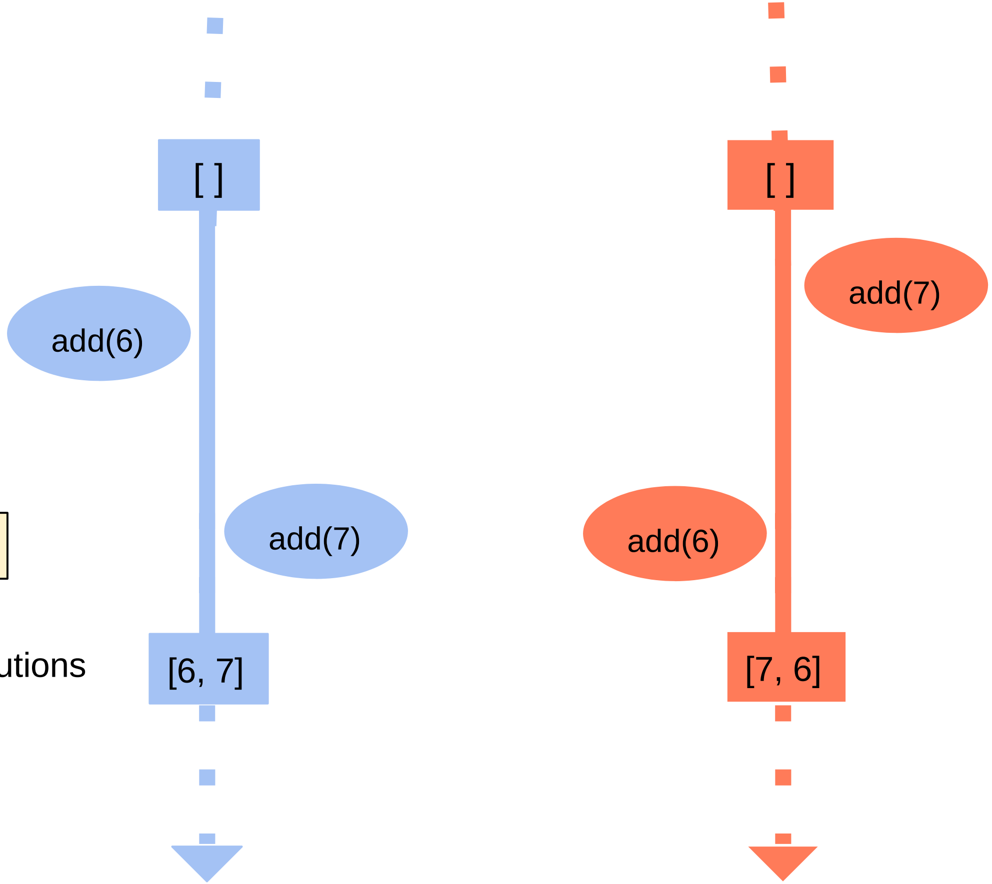
Abstraction **α**: list → multiset of elements

If



Commutativity Modulo Abstraction ("Abstract Commutativity")

```
        shared = new List()
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.add(6)       ||    shared.add(7)
        return sort(shared)
```



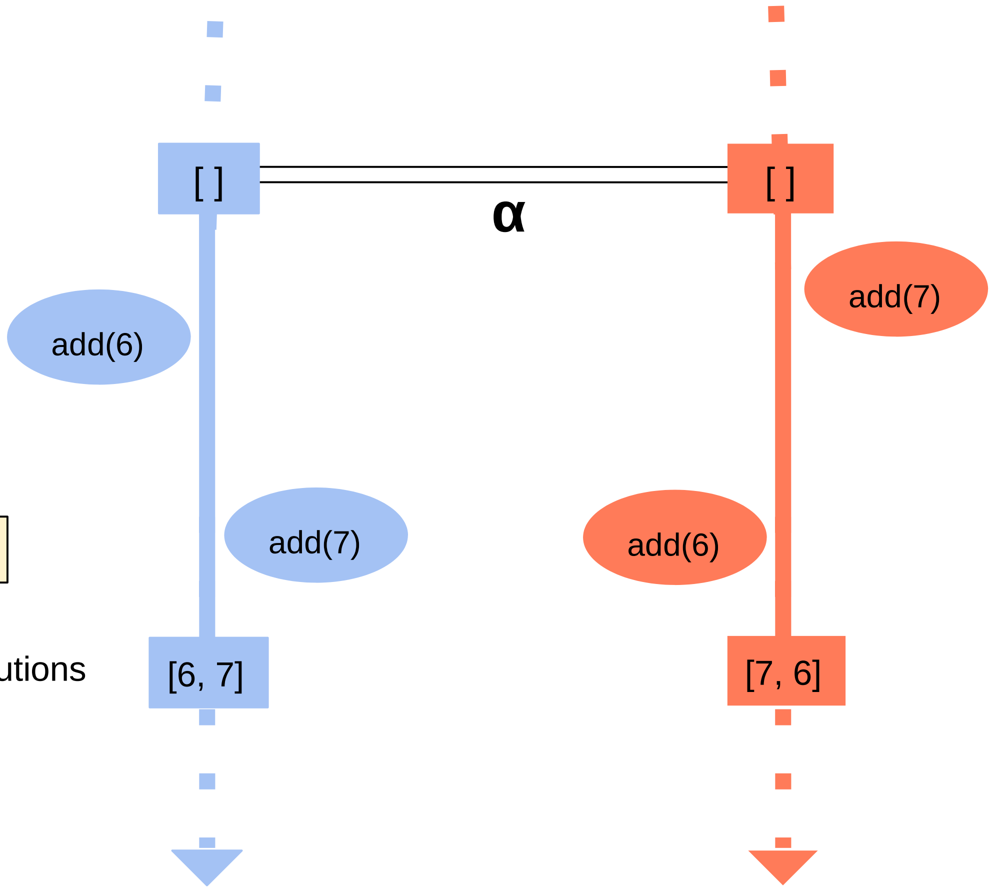
Abstraction **α**: list → multiset of elements

If

(1) *shared* has the same initial **abstraction** in both executions

Commutativity Modulo Abstraction (“Abstract Commutativity”)

```
          shared = new List()
while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
atomic:               ||   atomic:
  shared.add(6)       ||     shared.add(7)

          return sort(shared)
```



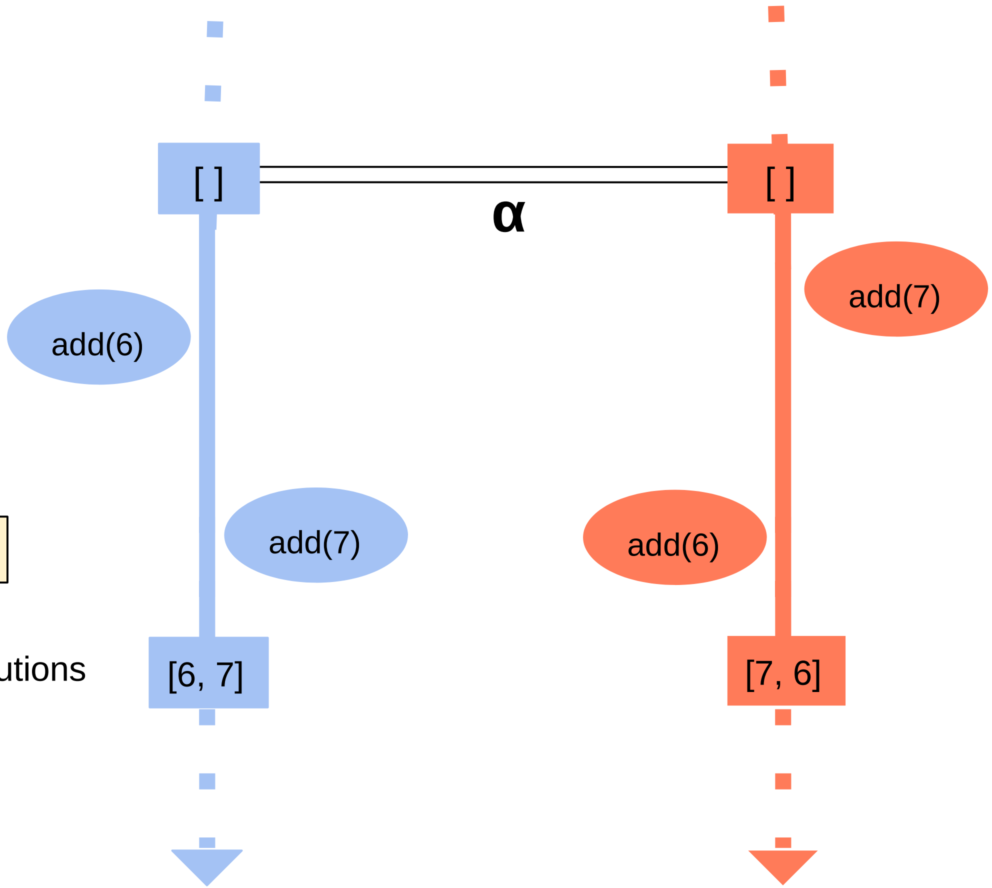
Abstraction **α**: list → multiset of elements

If

(1) *shared* has the same initial **abstraction** in both executions

Commutativity Modulo Abstraction (“Abstract Commutativity”)

```
          shared = new List()
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.add(6)       ||    shared.add(7)

          return sort(shared)
```



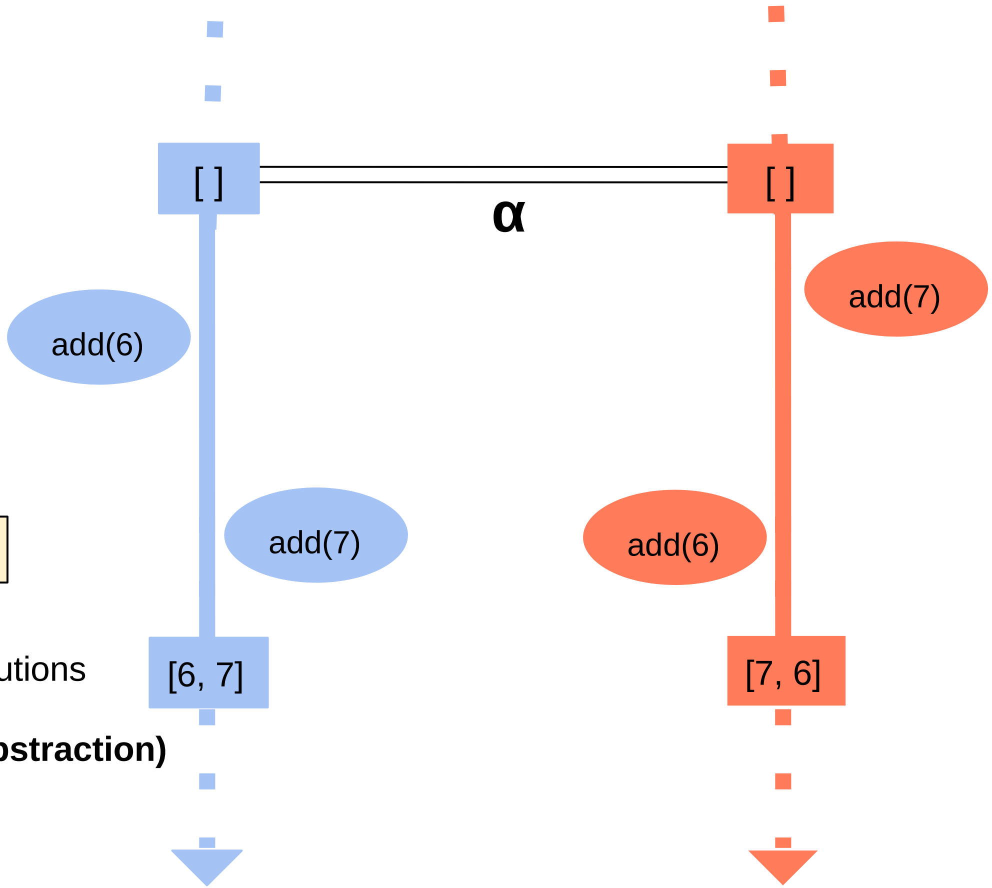
Abstraction **α**: list → multiset of elements

If

✅ *shared* has the same initial **abstraction** in both executions

Commutativity Modulo Abstraction ("Abstract Commutativity")

```
shared = new List()

while i < h:            ||   while j < 100:
  i += 1                ||     j += 1
atomic:                 ||   atomic:
  shared.add(6)         ||     shared.add(7)

return sort(shared)
```



Abstraction **α**: list → multiset of elements

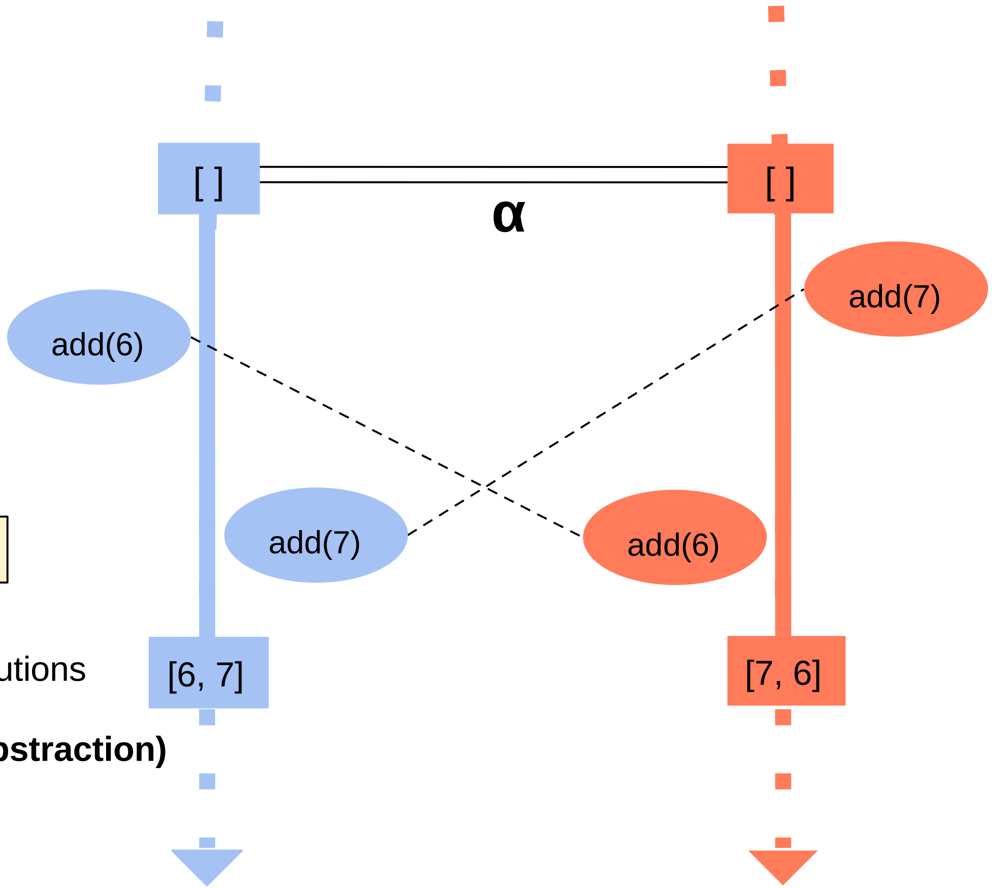
If

✅ *shared* has the same initial **abstraction** in both executions

(2) executions perform "same" modifications (**modulo abstraction**)

Commutativity Modulo Abstraction ("Abstract Commutativity")

```
              shared = new List()
while i < h:           ||   while j < 100:
  i += 1               ||     j += 1
atomic:                ||   atomic:
  shared.add(6)        ||     shared.add(7)

              return sort(shared)
```



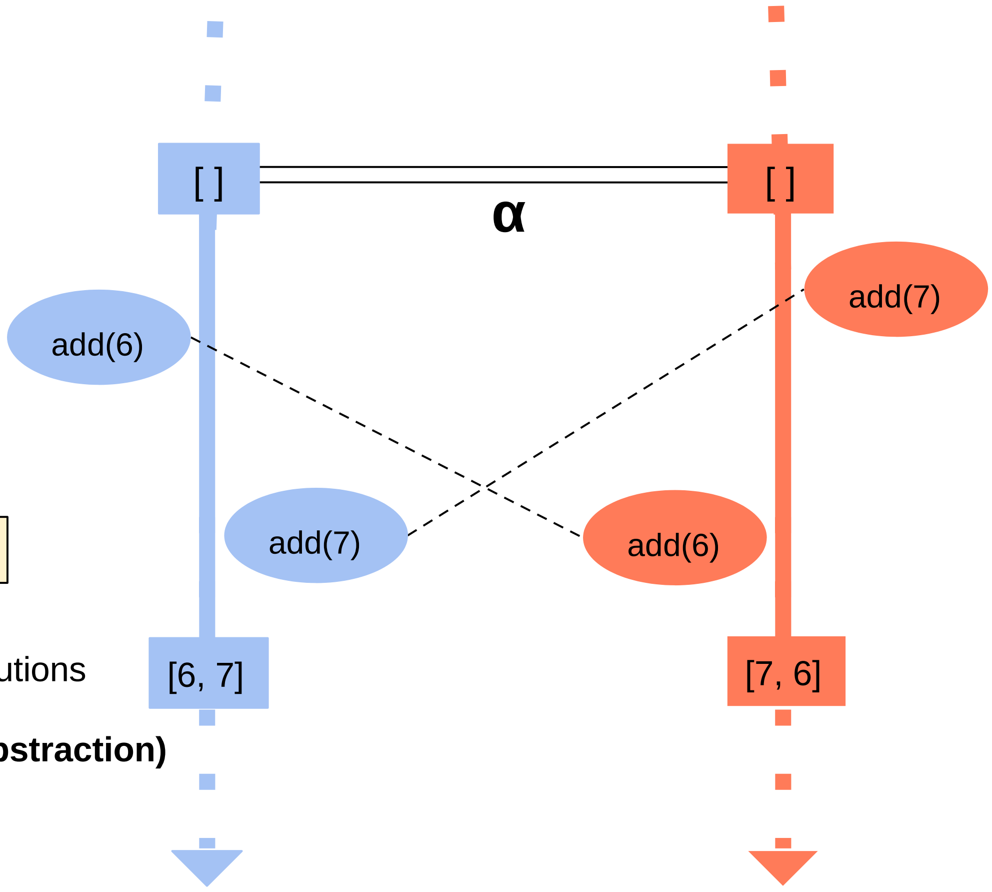
Abstraction **α**: list → multiset of elements

If

- *shared* has the same initial **abstraction** in both executions
- executions perform "same" modifications (**modulo abstraction)**

Commutativity Modulo Abstraction ("Abstract Commutativity")

```
          shared = new List()
while i < h:       ||  while j < 100:
  i += 1           ||    j += 1
atomic:            ||  atomic:
  shared.add(6)    ||    shared.add(7)
        return sort(shared)
```



Abstraction **α**: list → multiset of elements

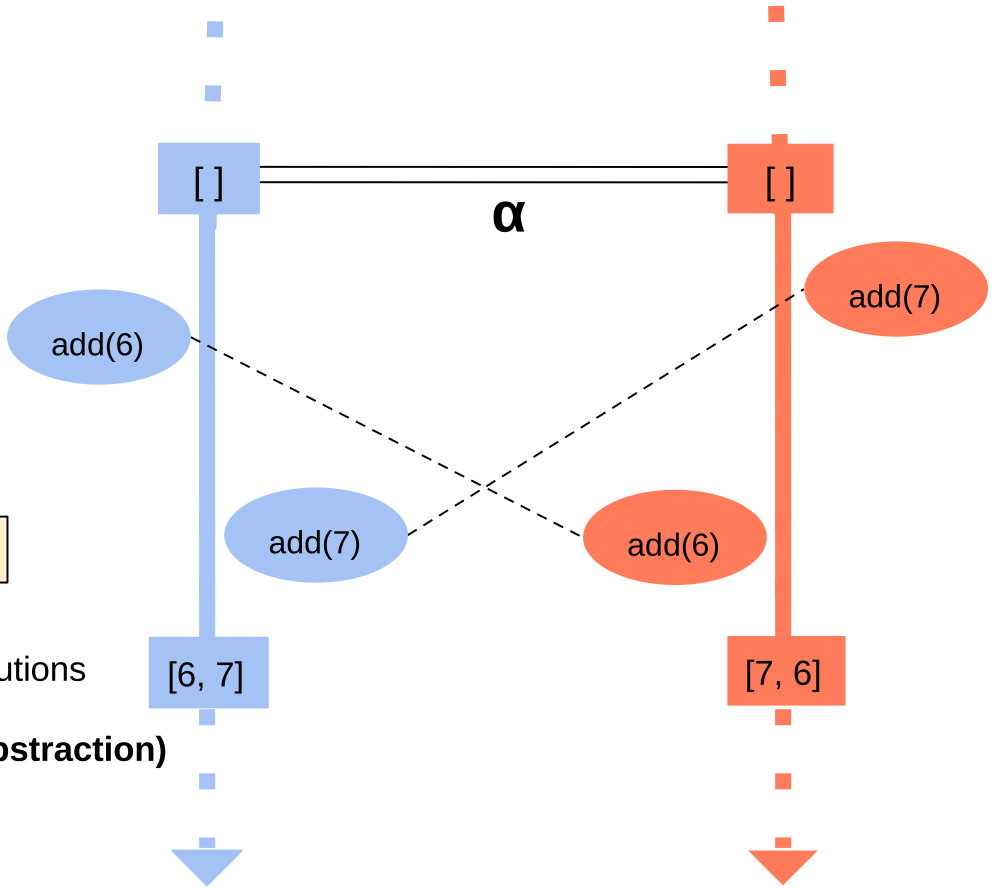
If

- *shared* has the same initial **abstraction** in both executions
- executions perform "same" modifications (**modulo abstraction)**

(3) the modifications commute **modulo abstraction**

Commutativity Modulo Abstraction ("Abstract Commutativity")

```
          shared = new List()
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.add(6)       ||    shared.add(7)
          return sort(shared)
```



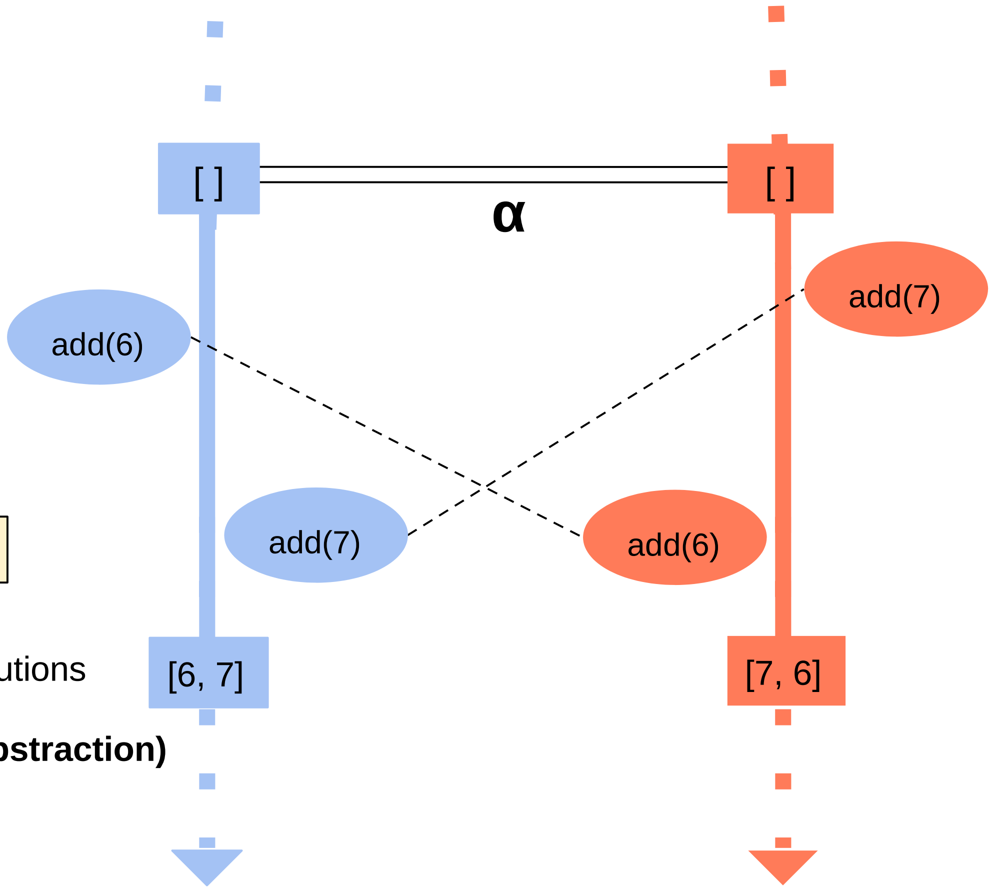
Abstraction **α**: list → multiset of elements

If

- *shared* has the same initial **abstraction** in both executions
- executions perform "same" modifications (**modulo abstraction)**
- the modifications commute **modulo abstraction**

Commutativity Modulo Abstraction (“Abstract Commutativity”)

```
             shared = new List()
while i < h:            ||  while j < 100:
  i += 1                ||    j += 1
atomic:                 ||  atomic:
  shared.add(6)         ||    shared.add(7)
          return sort(shared)
```



Abstraction **α**: list → multiset of elements

If

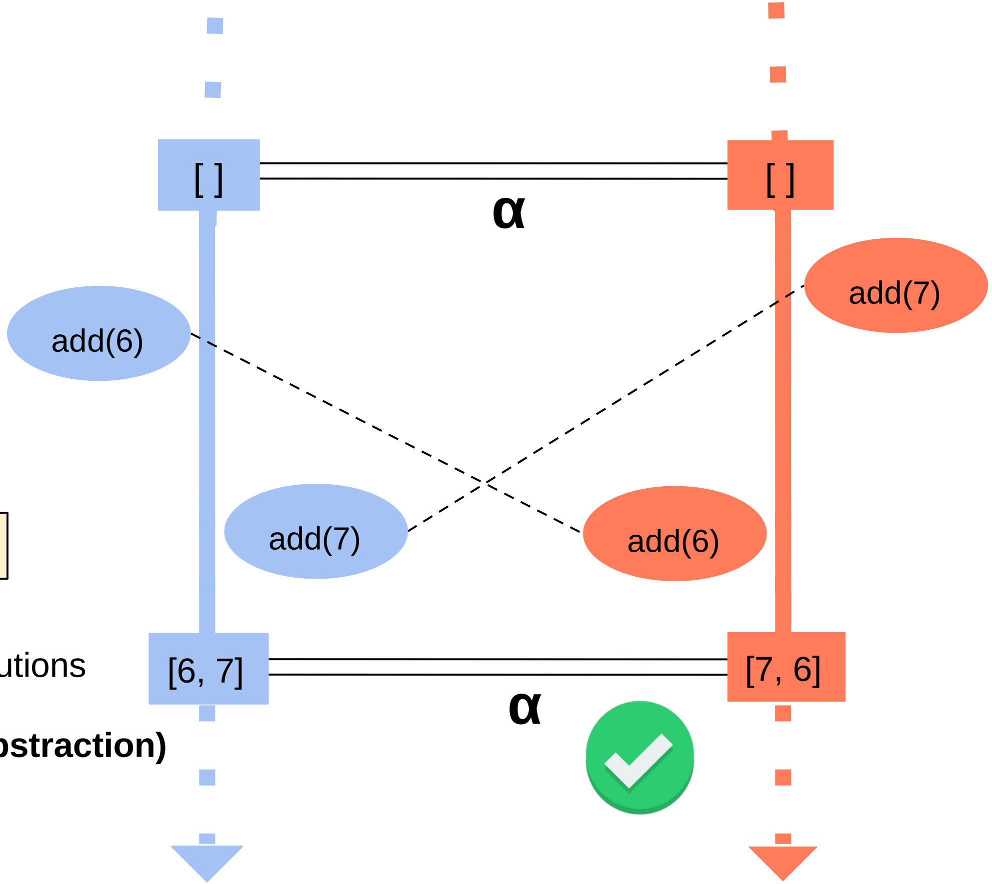
- ✅ *shared* has the same initial **abstraction** in both executions
- ✅ executions perform “same” modifications (**modulo abstraction)**
- ✅ the modifications commute **modulo abstraction**

then *shared* has the same final **abstraction** in both executions

Commutativity Modulo Abstraction (“Abstract Commutativity”)

```
        shared = new List()
while i < h:        ||  while j < 100:
  i += 1            ||    j += 1
atomic:             ||  atomic:
  shared.add(6)     ||    shared.add(7)
        return sort(shared)
```

Abstraction **α**: list → multiset of elements



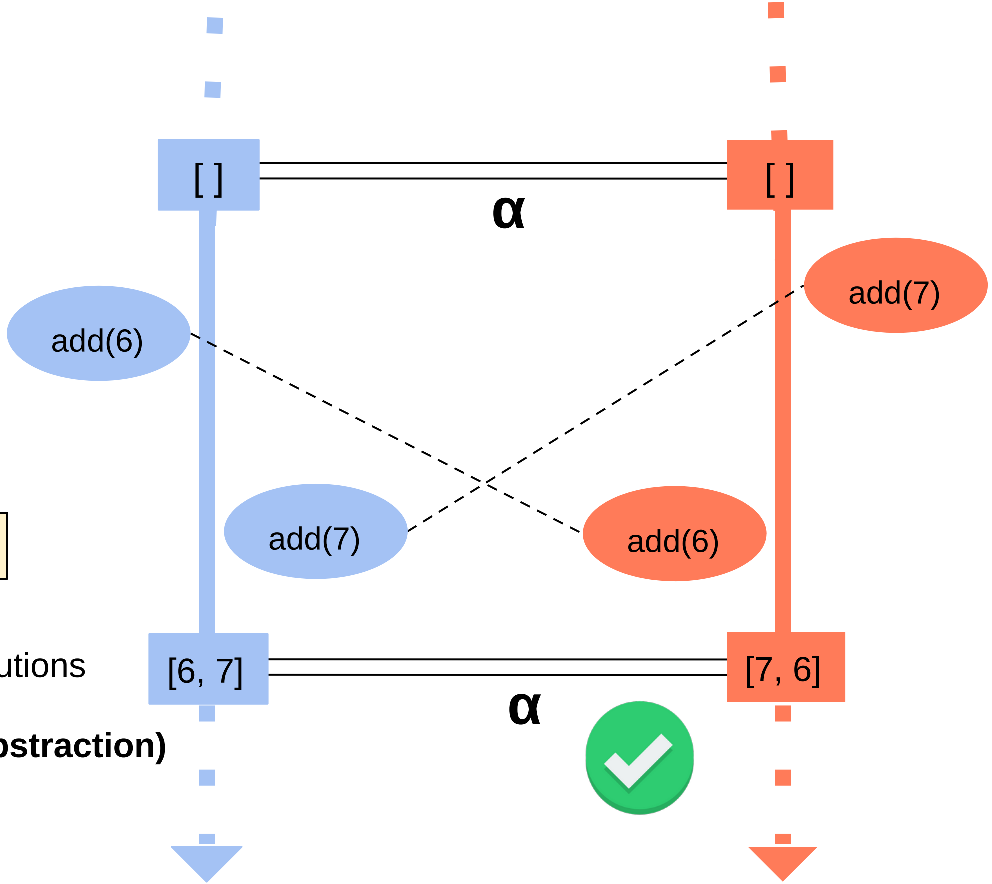
If

- *shared* has the same initial **abstraction** in both executions
- executions perform “same” modifications (**modulo abstraction)**
- the modifications commute **modulo abstraction**

then *shared* has the same final **abstraction** in both executions

Commutativity Modulo Abstraction ("Abstract Commutativity")

```
        shared = new List()
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.add(6)       ||    shared.add(7)
        return sort(shared)
```



Abstraction **α**: list → multiset of elements

If

- *shared* has the same initial **abstraction** in both executions
- executions perform "same" modifications (**modulo abstraction)**
- the modifications commute **modulo abstraction**

then *shared* has the same final **abstraction** in both executions

Commutativity Modulo Abstraction (“Abstract Commutativity”)

Abstraction **α**: list $\rightarrow$ multiset of elements

Commutativity Modulo Abstraction (“Abstract Commutativity”)

Abstraction **α**: list → multiset of elements

	Commutativity	**Commutativity modulo α**
f and *g* commute		
f and *g* are the “same”		

Commutativity Modulo Abstraction ("Abstract Commutativity")

Abstraction **α**: list → multiset of elements

	Commutativity	**Commutativity modulo α**
f and *g* commute	$f \circ g = g \circ f$	
f and *g* are the "same"		

Commutativity Modulo Abstraction (“Abstract Commutativity”)

Abstraction **α**: list → multiset of elements

	Commutativity	**Commutativity modulo α**
f and *g* commute	$f \circ g = g \circ f$	$\forall v, v'.\, \alpha(v) = \alpha(v') \Rightarrow \alpha(f(g(v))) = \alpha(g(f(v')))$
f and *g* are the “same”		

Commutativity Modulo Abstraction (“Abstract Commutativity”)

Abstraction **α**: list → multiset of elements

	Commutativity	**Commutativity modulo** α
f and *g* commute	$f \circ g = g \circ f$	lists $\forall v, v'.\, \alpha(v) = \alpha(v')$ $\Rightarrow \quad \alpha(f(g(v)) = \alpha(g(f(v'))$
f and *g* are the “same”		

Commutativity Modulo Abstraction (“Abstract Commutativity”)

Abstraction **α**: list → multiset of elements

	Commutativity	**Commutativity modulo α**
f and *g* commute	$f \circ g = g \circ f$	$\forall \overset{\text{lists}}{v, v'} .\, \alpha(v) \overset{\text{contain same elements}}{=} \alpha(v') \Rightarrow \alpha(f(g(v))) = \alpha(g(f(v')))$
f and *g* are the “same”		

Commutativity Modulo Abstraction ("Abstract Commutativity")

Abstraction **α**: list → multiset of elements

	Commutativity	**Commutativity modulo α**
f and *g* commute	$f \circ g = g \circ f$	lists contain same elements $\forall v, v' .\, \alpha(v) = \alpha(v')$ $\Rightarrow \quad \alpha(f(g(v)) = \alpha(g(f(v'))$ add(6) add(7)
f and *g* are the "same"		

Commutativity Modulo Abstraction (“Abstract Commutativity”)

Abstraction **α**: list → multiset of elements

	Commutativity	**Commutativity modulo α**
f and *g* commute	$f \circ g = g \circ f$	$\forall \overbrace{v, v'}^{\text{lists}} .\, \alpha(v) \overset{\text{contain same elements}}{=} \alpha(v')$ $\Rightarrow \alpha(\underbrace{f}_{\text{add(6)}}(\underbrace{g}_{\text{add(7)}}(v)) = \alpha(g(f(v'))$
f and *g* are the “same”	$f = g$	

Commutativity Modulo Abstraction (“Abstract Commutativity”)

Abstraction **α**: list → multiset of elements

	Commutativity	**Commutativity modulo α**
f and *g* commute	$f \circ g = g \circ f$	lists contain same elements $\forall v, v' .\, \alpha(v) = \alpha(v')$ $\Rightarrow \quad \alpha(f(g(v))) = \alpha(g(f(v')))$ add(6) add(7)
f and *g* are the “same”	$f = g$	$\forall v, v' .\, \alpha(v) = \alpha(v')$ $\Rightarrow \quad \alpha(f(v)) = \alpha(g(v'))$

Abstractions

```
shared = new List()

while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.add(6)       ||    shared.add(7)

return sort(shared)
```

Abstraction **α**: list → multiset of elements

Abstractions

```
shared = new List()

while i < h:              ||   while j < 100:
  i += 1                  ||     j += 1
atomic:                   ||   atomic:
  shared.add(6)           ||     shared.add(7)

return sort(shared)
```

Abstraction **α**: list → multiset of elements

Abstraction **α**: list → mean

Abstraction **α**: list → sum

Abstractions

```
shared = new List()

while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.add(6)       ||    shared.add(7)

return sort(shared)
```

Abstraction **α**: list → multiset of elements

Abstraction **α**: list → mean

Abstraction **α**: list → sum

Abstraction **α**: list → length

Abstractions

```
shared = new List()

while i < h:            ||    while j < 100:
  i += 1                ||      j += 1
atomic:                 ||    atomic:
  shared.add(6)         ||      shared.add(7)

return sort(shared)
```

Abstraction **α**: list → multiset of elements

Abstraction **α**: list → mean

Abstraction **α**: list → sum

Abstraction **α**: list → length

…

Abstractions

```
shared = new List()

while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
atomic:               ||   atomic:
  shared.add(6)       ||     shared.add(7)

return sort(shared)
```

```
shared = new Map()

while i < h:          ||   while j < 100:
  i += 1              ||     j += 1
atomic:               ||   atomic:
  shared.put(1,8)     ||     shared.put(1,h)

return shared.keySet()
```

Abstraction **α**: list → multiset of elements

Abstraction **α**: list → mean

Abstraction **α**: list → sum

Abstraction **α**: list → length

…

Abstractions

```
shared = new List()

while i < h:              ||   while j < 100:
  i += 1                  ||     j += 1
atomic:                   ||   atomic:
  shared.add(6)           ||     shared.add(7)

return sort(shared)
```

```
shared = new Map()

while i < h:              ||   while j < 100:
  i += 1                  ||     j += 1
atomic:                   ||   atomic:
  shared.put(1,8)         ||     shared.put(1,h)

return shared.keySet()
```

Abstraction **α**: list → multiset of elements

Abstraction **α**: map → set of keys

Abstraction **α**: list → mean

Abstraction **α**: list → sum

Abstraction **α**: list → length

...

Abstractions

```
shared = new List()
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.add(6)       ||    shared.add(7)
return sort(shared)
```

Abstraction **α**: list → multiset of elements

Abstraction **α**: list → mean

Abstraction **α**: list → sum

Abstraction **α**: list → length

...

```
shared = new Map()
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.put(1,8)     ||    shared.put(1,h)
return shared.keySet()
```

Abstraction **α**: map → set of keys

Abstractions

```
shared = new List()

while i < h:            ||  while j < 100:
  i += 1                ||    j += 1
atomic:                 ||  atomic:
  shared.add(6)         ||    shared.add(7)

return sort(shared)
```

Abstraction **α**: list → multiset of elements

Abstraction **α**: list → mean

Abstraction **α**: list → sum

Abstraction **α**: list → length

...

```
shared = new Map()

while i < h:            ||  while j < 100:
  i += 1                ||    j += 1
atomic:                 ||  atomic:
  shared.put(1,8)       ||    shared.put(1,h)

return shared.keySet()
```

Abstraction **α**: map → set of keys

```
shared = new Map()

if h > 0:               ||  if h <= 0:
 atomic:                ||   atomic:
   shared.put(1,8)      ||    shared.put(1,h)

return shared.keySet()
```

Abstractions

```
shared = new List()
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.add(6)       ||    shared.add(7)
return sort(shared)
```

Abstraction **α**: list → multiset of elements

Abstraction **α**: list → mean

Abstraction **α**: list → sum

Abstraction **α**: list → length

...

```
shared = new Map()
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.put(1,8)     ||    shared.put(1,h)
return shared.keySet()
```

Abstraction **α**: map → set of keys

“Same” modulo α

```
shared = new Map()
if h > 0:             ||  if h <= 0:
  atomic:             ||    atomic:
    shared.put(1,8)   ||      shared.put(1,h)
return shared.keySet()
```

Abstractions

```
        shared = new List()
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.add(6)       ||    shared.add(7)
        return sort(shared)
```

Abstraction **α**: list → multiset of elements

Abstraction **α**: list → mean

Abstraction **α**: list → sum

Abstraction **α**: list → length

...

```
        shared = new Map()
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.put(1,8)     ||    shared.put(1,h)
        return shared.keySet()
```

Abstraction **α**: map → set of keys

"Same" modulo α

```
        shared = new Map()
if h > 0:             ||  if h <= 0:
 atomic:              ||   atomic:
  shared.put(1,8)     ||    shared.put(1,h)
        return shared.keySet()
```

CommCSL

CommCSL

$$\Gamma \vdash \{P\}C\{Q\}$$

CommCSL

$$\Gamma \vdash \{P\}C\{Q\}$$

Program

CommCSL

$$\Gamma \vdash \{\underbrace{P}_{\text{Precondition}}\}\underbrace{C}_{\text{Program}}\{\underbrace{Q}_{\text{Postcondition}}\}$$

CommCSL

Precondition

Postcondition

$$\Gamma \vdash \{P\}C\{Q\}$$

Invariant

Program

CommCSL

Precondition

Postcondition

$$\Gamma \vdash \{P\} C \{Q\}$$

Invariant

Program

$$\frac{\Gamma_\perp = \Gamma \Rightarrow x \notin fv(\Gamma)}{\Gamma_\perp \vdash \{P[e/x]\}x{:=}e\{P\}}\ (\textsc{Assign}) \qquad \frac{x \notin fv(e) \quad \Gamma_\perp = \Gamma \Rightarrow x \notin fv(\Gamma)}{\Gamma_\perp \vdash \{\mathtt{emp}\}x{:=}\mathtt{alloc}(e)\{x \mapsto^1 e\}}\ (\textsc{New})$$

$$\frac{x \notin fv(e_1, e_2) \quad \Gamma_\perp = \Gamma \Rightarrow x \notin fv(\Gamma)}{\Gamma_\perp \vdash \{e_1 \mapsto^r e_2\}x{:=}[e]\{e_1 \mapsto^r e_2 * x = e_2\}}\ (\textsc{Read}) \qquad \frac{}{\Gamma_\perp \vdash \{e_1 \mapsto^1 \_\}[e_1]{:=}e_2\{e_1 \mapsto^1 e_2\}}\ (\textsc{Write})$$

$$\frac{\Gamma_\perp \vdash \{P \land b\}c_1\{Q\} \quad \Gamma_\perp \vdash \{P \land \neg b\}c_2\{Q\}}{\Gamma_\perp \vdash \{P \land \mathrm{Low}(b)\}\mathtt{if}\ (b)\ \mathtt{then}\ \{c_1\}\ \mathtt{else}\ \{c_2\}\{Q\}}\ (\textsc{If1})$$

$$\frac{\Gamma_\perp \vdash \{P \land b\}c_1\{Q\} \quad \Gamma_\perp \vdash \{P \land \neg b\}c_2\{Q\} \quad \mathit{unary}\ Q}{\Gamma_\perp \vdash \{P\}\mathtt{if}\ (b)\ \mathtt{then}\ \{c_1\}\ \mathtt{else}\ \{c_2\}\{Q\}}\ (\textsc{If2})$$

$$\frac{\Gamma_\perp \vdash \{P \land b\}c_I\{P \land \mathrm{Low}(b)\}}{\Gamma_\perp \vdash \{P \land \mathrm{Low}(b)\}\mathtt{while}\ (b)\ \mathtt{do}\ \{c_I\}\{P \land \neg b\}}\ (\textsc{While1}) \qquad \frac{\Gamma_\perp \vdash \{P \land b\}c_I\{P\} \quad \mathit{unary}\ P}{\Gamma_\perp \vdash \{P\}\mathtt{while}\ (b)\ \mathtt{do}\ \{c_I\}\{P \land \neg b\}}\ (\textsc{While2})$$

$$\frac{\Gamma_\perp \vdash \{P\}c_1\{R\} \quad \Gamma_\perp \vdash \{R\}c_2\{Q\}}{\Gamma_\perp \vdash \{P\}c_1;c_2\{Q\}}\ (\textsc{Seq}) \qquad \frac{}{\Gamma_\perp \vdash \{P\}\mathtt{skip}\{P\}}\ (\textsc{Skip})$$

$$\frac{\begin{array}{c}\Gamma_\perp \vdash \{P_1\}c_1\{Q_1\} \quad \Gamma_\perp \vdash \{P_2\}c_2\{Q_2\} \quad fv(P_1, c_1, Q_1) \cap mod(c_2) = \emptyset \quad fv(P_2, c_2, Q_2) \cap mod(c_1) = \emptyset \\ \Gamma_\perp = \Gamma \Rightarrow fv(\Gamma) \cap mod(c_1, c_2) = \emptyset \quad P_1 \text{ is precise or } P_2 \text{ is precise}\end{array}}{}\ (\textsc{Par})$$

$$\frac{\ldots Q'\} \quad Q' \Rightarrow Q}{\Gamma_\perp \vdash \{P\}c\{Q\}}\ (\textsc{Cons}) \qquad \frac{\begin{array}{c}fv(R) \cap mod(c) = \emptyset \quad \Gamma_\perp \vdash \{P\}c\{Q\} \\ P \text{ is precise or } R \text{ is precise}\end{array}}{\Gamma_\perp \vdash \{P * R\}c\{Q * R\}}\ (\textsc{Frame})$$

$$\frac{x \notin fv(c) \quad \Gamma_\perp = \Gamma \Rightarrow x \notin fv(\Gamma) \quad \Gamma_\perp \vdash \{P\}c\{Q\}}{\Gamma_\perp \vdash \{\exists x.\, P\}c\{\exists x.\, Q\}}\ (\textsc{Exists})$$

$$\frac{\begin{array}{c}\Gamma = \langle \alpha, f_{a_s}, f_{a_u}, I(x)\rangle \quad \Gamma \text{ is valid} \quad I(x) \text{ is unary and precise} \\ \Gamma \vdash \{P * \mathit{sguard}(1, \emptyset^\#) * \mathit{uguard}([])\}c\{Q * \mathit{sguard}(1, x_s) * \mathit{PRE}_s(x_s) * \mathit{uguard}(x_u) * \mathit{PRE}_u(x_u)\}\end{array}}{\perp \vdash \{I(x) * \mathrm{Low}(\alpha(x)) * P\}c\{\exists x'.\, I(x') * \mathrm{Low}(\alpha(x')) * Q\}}\ (\textsc{Share})$$

$$\frac{\begin{array}{c}\Gamma = \langle \alpha, f_{a_s}, f_{a_u}, I(x)\rangle \quad I(x_v) \text{ is unary and precise} \\ x_v \notin fv(P, Q) \quad x_s, x_a, x_v \notin mod(c) \quad \mathit{noguard}(P) \quad \mathit{noguard}(Q) \\ \perp \vdash \{P * I(x_v)\}c\{Q * I(f_{a_s}(x_v, x_a))\}\end{array}}{\Gamma \vdash \{P * \mathit{sguard}(r, x_s)\}\mathtt{atomic}\ c\{Q * \mathit{sguard}(r, x_s \cup^\# \{x_a\}^\#)\}}\ (\textsc{AtomicShr})$$

$$\frac{\begin{array}{c}\Gamma = \langle \alpha, f_{a_s}, f_{a_u}, I(x)\rangle \quad I(x_v) \text{ is unary and precise} \\ x_v \notin fv(P, Q) \quad x_s, x_a, x_v \notin mod(c) \quad \mathit{noguard}(P) \quad \mathit{noguard}(Q) \\ \perp \vdash \{P * I(x_v)\}c\{Q * I(f_{a_u}(x_v, x_a))\}\end{array}}{\Gamma \vdash \{P * \mathit{uguard}(x_s)\}\mathtt{atomic}\ c\{Q * \mathit{uguard}(x_s ++ [x_a])\}}\ (\textsc{AtomicUnq})$$

CommCSL

CommCSL

- Relational concurrent separation logic
- Support for (abstract) commutativity-based information flow reasoning
- Thread-modular reasoning, mutable heaps

- Other features:
  - Low events, standard output…
  - More complete support for non-symmetric concurrency

- Formalized and proved sound in Isabelle/HOL
  - Challenging soundness argument distinct from existing logics
  - Available on the Archive of Formal Proofs

CommCSL

- Relational concurrent separation logic
- Support for (abstract) commutativity-based information flow reasoning
- Thread-modular reasoning, mutable heaps

- Other features:
  - Low events, standard output…
  - More complete support for non-symmetric concurrency

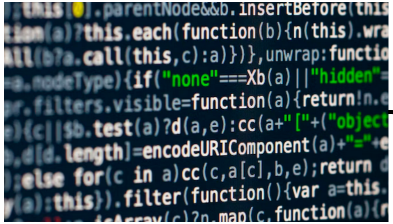
Non-interference theorem

- Formalized and proved sound in Isabelle/HOL
  - Challenging soundness argument distinct from existing logics
  - Available on the Archive of Formal Proofs

Implementation

HyperViper

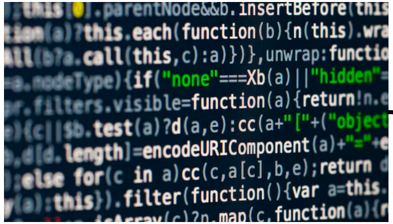
Implementation



Source code

HyperViper

Implementation

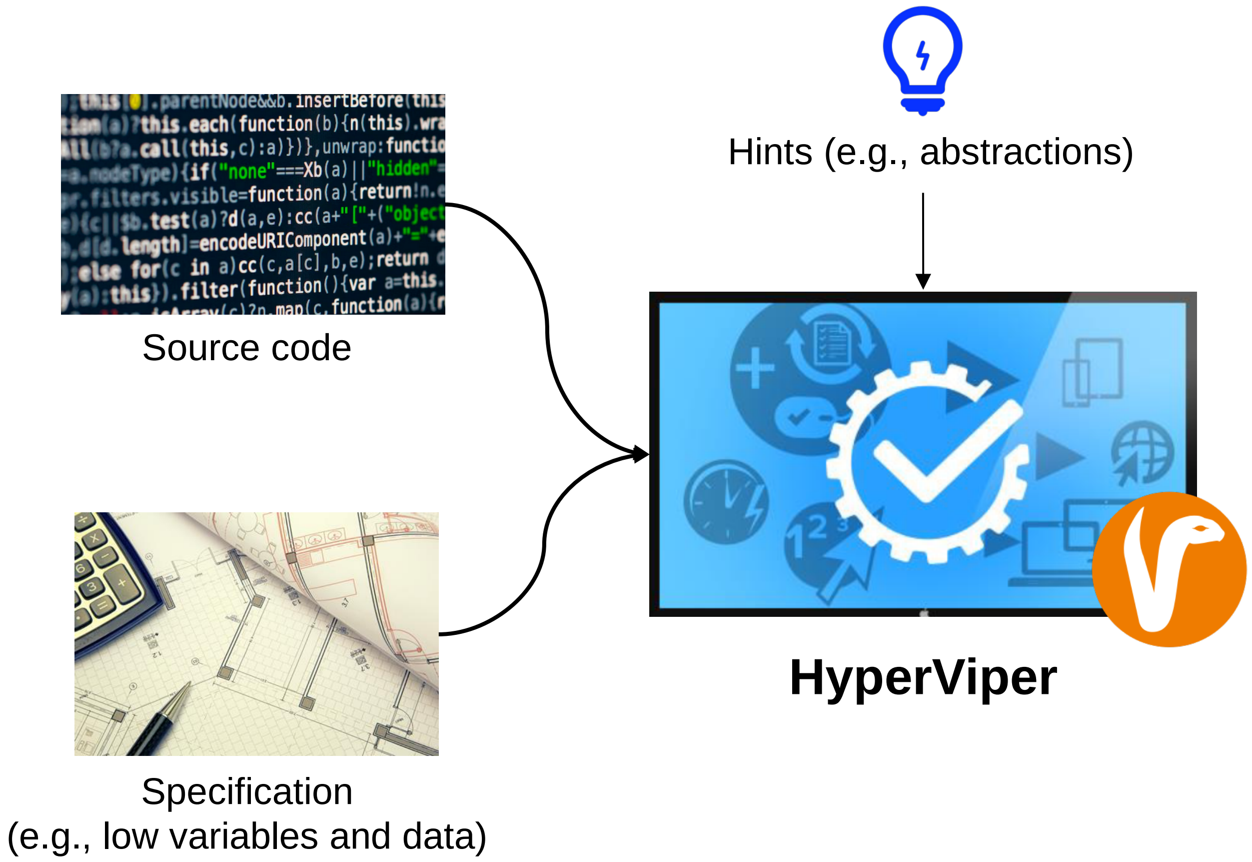


Source code

Specification
(e.g., low variables and data)

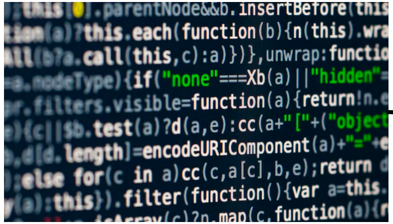
HyperViper

Implementation



Implementation

Hints (e.g., abstractions)



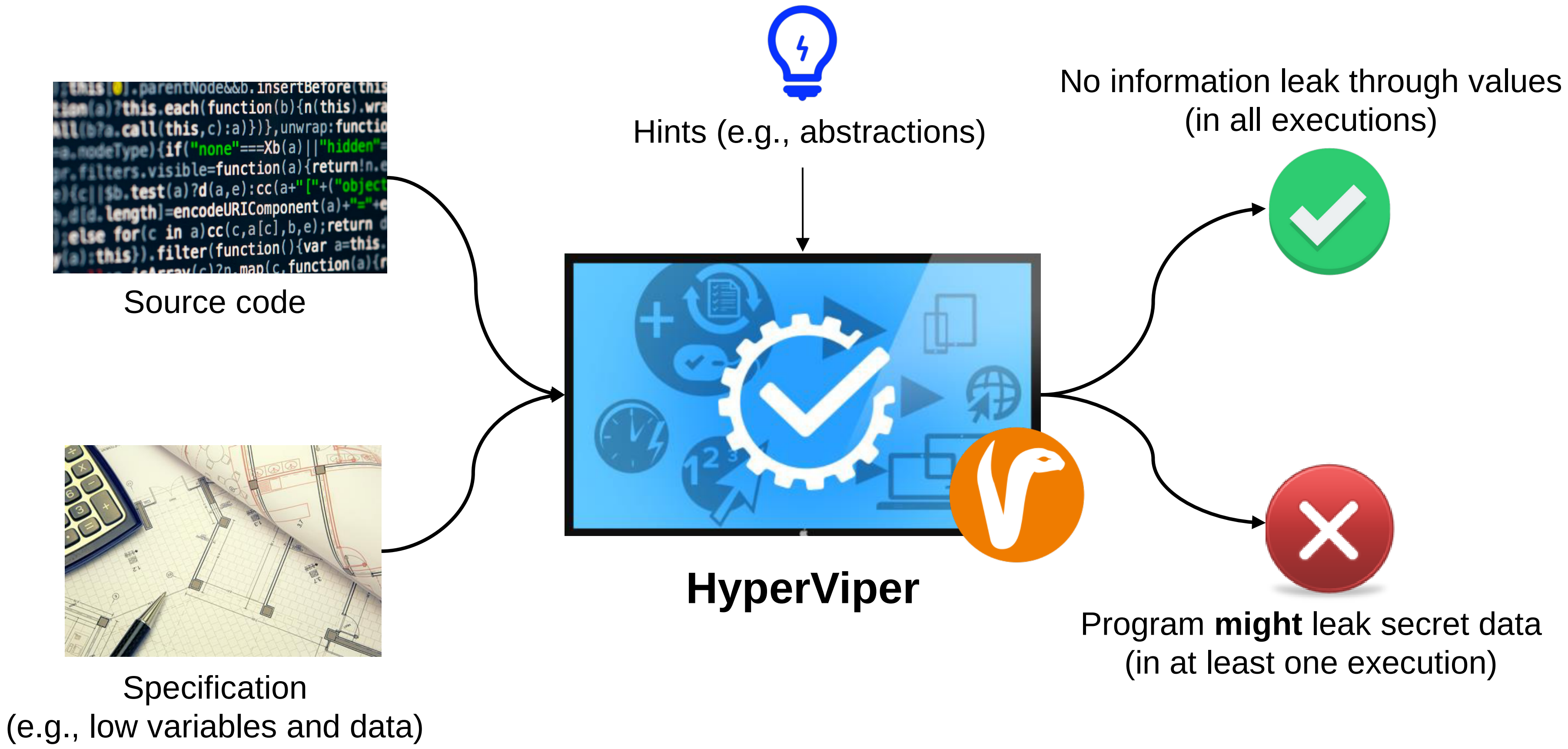
Source code

Specification
(e.g., low variables and data)

HyperViper

No information leak through values
(in all executions)

Implementation



HyperViper

- Automated, SMT-based verifier
  - Based on Viper verification infrastructure and Z3
  - Relational reasoning using Modular Product Programs
- User provides abstractions, pre- and postconditions, invariants…
- Supports dynamic thread creation, multiple shared resources, …
- https://github.com/viperproject/hyperviper

HyperViper

- Automated, SMT-based verifier
  - Based on Viper verification infrastructure and Z3
  - Relational reasoning using Modular Product Programs
- User provides abstractions, pre- and postconditions, invariants…
- Supports dynamic thread creation, multiple shared resources, …
- https://github.com/viperproject/hyperviper

```
lockType IntLock {
 type Int
 invariant(l, v) = [l.lockInt |-> ?cp && [cp.val |-> v]]
 alpha(v): Int = 0  // we abstract to a constant, so everything commutes
 actions = [(SetValue, Int, duplicable)]
 action SetValue(v, arg)
   requires true
 { arg }
 noLabels = 2
}

...

method worker(l: Lock, lbl: Int)
 requires lowEvent && sguard[IntLock,SetValue](l, Set(lbl))
 requires sguardArgs[IntLock,SetValue](l, Set(lbl)) == Multiset[Int]()
 ensures sguard[IntLock,SetValue](l, Set(lbl))
 ensures allPre[IntLock, SetValue](sguardArgs[IntLock,SetValue](l, Set(lbl)))
{

   var v: Int
   v := lbl
   with[IntLock] l performing SetValue(v) at lbl {
       l.lockInt.val := v
   }

}

method print(i: Int)
 requires lowEvent && low(i)
```

HyperViper

- Automated, SMT-based verifier
  - Based on Viper verification infrastructure and Z3
  - Relational reasoning using Modular Product Programs
- User provides abstractions, pre- and postconditions, invariants…
- Supports dynamic thread creation, multiple shared resources, …
- https://github.com/viperproject/hyperviper

```
lockType IntLock {
 type Int
 invariant(l, v) = [l.lockInt |-> ?cp && [cp.val |-> v]]
 alpha(v): Int = 0  // we abstract to a constant, so everything commutes
 actions = [(SetValue, Int, duplicable)]
 action SetValue(v, arg)
   requires true
 { arg }
 noLabels = 2
}

...

method worker(l: Lock, lbl: Int)
 requires lowEvent && sguard[IntLock,SetValue](l, Set(lbl))
 requires sguardArgs[IntLock,SetValue](l, Set(lbl)) == Multiset[Int]()
 ensures sguard[IntLock,SetValue](l, Set(lbl))
 ensures allPre[IntLock, SetValue](sguardArgs[IntLock,SetValue](l, Set(lbl)))
{

   var v: Int
   v := lbl
   with[IntLock] l performing SetValue(v) at lbl {
       l.lockInt.val := v
   }

}

method print(i: Int)
 requires lowEvent && low(i)
```

HyperViper

- Automated, SMT-based verifier
  - Based on Viper verification infrastructure and Z3
  - Relational reasoning using Modular Product Programs
- User provides abstractions, pre- and postconditions, invariants…
- Supports dynamic thread creation, multiple shared resources, …
- https://github.com/viperproject/hyperviper

```
lockType IntLock {
 type Int
 invariant(l, v) = [l.lockInt |-> ?cp && [cp.val |-> v]]
 alpha(v): Int = 0  // we abstract to a constant, so everything commutes
 actions = [(SetValue, Int, duplicable)]
 action SetValue(v, arg)
   requires true
 { arg }
 noLabels = 2
}

...

method worker(l: Lock, lbl: Int)
 requires lowEvent && sguard[IntLock,SetValue](l, Set(lbl))
 requires sguardArgs[IntLock,SetValue](l, Set(lbl)) == Multiset[Int]()
 ensures sguard[IntLock,SetValue](l, Set(lbl))
 ensures allPre[IntLock, SetValue](sguardArgs[IntLock,SetValue](l, Set(lbl)))
{

   var v: Int
   v := lbl
   with[IntLock] l performing SetValue(v) at lbl {
       l.lockInt.val := v
   }

}

method print(i: Int)
 requires lowEvent && low(i)
```

HyperViper

- Automated, SMT-based verifier
  - Based on Viper verification infrastructure and Z3
  - Relational reasoning using Modular Product Programs
- User provides abstractions, pre- and postconditions, invariants…
- Supports dynamic thread creation, multiple shared resources, …
- https://github.com/viperproject/hyperviper

```
lockType IntLock {
 type Int
 invariant(l, v) = [l.lockInt |-> ?cp && [cp.val |-> v]]
 alpha(v): Int = 0  // we abstract to a constant, so everything commutes
 actions = [(SetValue, Int, duplicable)]
 action SetValue(v, arg)
   requires true
 { arg }
 noLabels = 2
}

...

method worker(l: Lock, lbl: Int)
 requires lowEvent && sguard[IntLock,SetValue](l, Set(lbl))
 requires sguardArgs[IntLock,SetValue](l, Set(lbl)) == Multiset[Int]()
 ensures sguard[IntLock,SetValue](l, Set(lbl))
 ensures allPre[IntLock, SetValue](sguardArgs[IntLock,SetValue](l, Set(lbl)))
{

   var v: Int
   v := lbl
   with[IntLock] l performing SetValue(v) at lbl {
       l.lockInt.val := v
   }

}

method print(i: Int)
 requires lowEvent && low(i)
```

Evaluation

Example	Data structure	Abstraction	LOC	Ann.	T
Count-Vaccinated	Counter, increment	None	44	46	10.15
Figure 2	Integer, add	None	129	95	10.90
Count-Sick-Days	Integer, add	None	52	45	13.67
Figure 1	Integer, arbitrary	Constant	29	20	1.52
Mean-Salary	List, append	Mean	80	84	14.10
Email-Metadata	List, append	Multiset	82	75	16.70
Patient-Statistic	List, append	Length	73	70	4.92
Debt-Sum	List, append	Sum	76	81	14.45
Sick-Employee-Names	Treeset, add	None	105	113	28.43
Website-Visitor-IPs	Listset, add	None	74	69	6.20
Figure 3	HashMap, put	Key set	129	96	10.37
Sales-By-Region	HashMap, disjoint put	None	129	104	12.37
Salary-Histogram	HashMap, increment value	None	135	109	13.78
Count-Purchases	HashMap, add value	None	137	109	11.73
Most-Valuable-Purchase	HashMap, conditional put	None	140	118	17.87
1-Producer-1-Consumer	Queue	Consumed sequence	82	88	3.23
Pipeline	Two queues	Consumed sequences	122	100	3.66
2-Producers-2-Consumers	Queue	Produced multiset	130	134	8.45

Evaluation

Example	Data structure	Abstraction	LOC	Ann.	T
Count-Vaccinated	Counter, increment	None	44	46	10.15
Figure 2	Integer, add	None	129	95	10.90
Count-Sick-Days	Integer, add	None	52	45	13.67
Figure 1	Integer, arbitrary	Constant	29	20	1.52
Mean-Salary	List, append	Mean	80	84	14.10
Email-Metadata	List, append	Multiset	82	75	16.70
Patient-Statistic	List, append	Length	73	70	4.92
Debt-Sum	List, append	Sum	76	81	14.45
Sick-Employee-Names	Treeset, add	None	105	113	28.43
Website-Visitor-IPs	Listset, add	None	74	69	6.20
Figure 3	HashMap, put	Key set	129	96	10.37
Sales-By-Region	HashMap, disjoint put	None	129	104	12.37
Salary-Histogram	HashMap, increment value	None	135	109	13.78
Count-Purchases	HashMap, add value	None	137	109	11.73
Most-Valuable-Purchase	HashMap, conditional put	None	140	118	17.87
1-Producer-1-Consumer	Queue	Consumed sequence	82	88	3.23
Pipeline	Two queues	Consumed sequences	122	100	3.66
2-Producers-2-Consumers	Queue	Produced multiset	130	134	8.45

Evaluation

Example	Data structure	Abstraction	LOC	Ann.	T
Count-Vaccinated	Counter, increment	None	44	46	10.15
Figure 2	Integer, add	None	129	95	10.90
Count-Sick-Days	Integer, add	None	52	45	13.67
Figure 1	Integer, arbitrary	Constant	29	20	1.52
Mean-Salary	List, append	Mean	80	84	14.10
Email-Metadata	List, append	Multiset	82	75	16.70
Patient-Statistic	List, append	Length	73	70	4.92
Debt-Sum	List, append	Sum	76	81	14.45
Sick-Employee-Names	Treeset, add	None	105	113	28.43
Website-Visitor-IPs	Listset, add	None	74	69	6.20
Figure 3	HashMap, put	Key set	129	96	10.37
Sales-By-Region	HashMap, disjoint put	None	129	104	12.37
Salary-Histogram	HashMap, increment value	None	135	109	13.78
Count-Purchases	HashMap, add value	None	137	109	11.73
Most-Valuable-Purchase	HashMap, conditional put	None	140	118	17.87
1-Producer-1-Consumer	Queue	Consumed sequence	82	88	3.23
Pipeline	Two queues	Consumed sequences	122	100	3.66
2-Producers-2-Consumers	Queue	Produced multiset	130	134	8.45

Evaluation

Example	Data structure	Abstraction	LOC	Ann.	T
Count-Vaccinated	Counter, increment	None	44	46	10.15
Figure 2	Integer, add	None	129	95	10.90
Count-Sick-Days	Integer, add	None	52	45	13.67
Figure 1	Integer, arbitrary	Constant	29	20	1.52
Mean-Salary	List, append	Mean	80	84	14.10
Email-Metadata	List, append	Multiset	82	75	16.70
Patient-Statistic	List, append	Length	73	70	4.92
Debt-Sum	List, append	Sum	76	81	14.45
Sick-Employee-Names	Treeset, add	None	105	113	28.43
Website-Visitor-IPs	Listset, add	None	74	69	6.20
Figure 3	HashMap, put	Key set	129	96	10.37
Sales-By-Region	HashMap, disjoint put	None	129	104	12.37
Salary-Histogram	HashMap, increment value	None	135	109	13.78
Count-Purchases	HashMap, add value	None	137	109	11.73
Most-Valu				8	17.87
1-Produce				8	3.23
Pipeline	Two queues	Consumed sequences	122	100	3.66
2-Producers-2-Consumers	Queue	Produced multiset	130	134	8.45

Secret data influences which thread performs which modification

- Modular reasoning about value sensitivity for concurrent programs
  - Independently of timing
  - Sound on real hardware
- Key idea is to exploit commutativity modulo abstraction
- Proved sound in Isabelle/HOL, automated in prototype verifier
- Will be presented at PLDI'23 by Marco

- Modular reasoning about value sensitivity for concurrent programs
  - Independently of timing
  - Sound on real hardware
- Key idea is to exploit commutativity modulo abstraction
- Proved sound in Isabelle/HOL, automated in prototype verifier
- Will be presented at PLDI’23 by Marco

- Modular reasoning about value sensitivity for concurrent programs
    - Independently of timing
    - Sound on real hardware
- Key idea is to exploit commutativity modulo abstraction
- Proved sound in Isabelle/HOL, automated in prototype verifier
- Will be presented at PLDI’23 by Marco

- Modular reasoning about value sensitivity for concurrent programs
  - Independently of timing
  - Sound on real hardware
- Key idea is to exploit commutativity modulo abstraction
- Proved sound in Isabelle/HOL, automated in prototype verifier
- Will be presented at PLDI’23 by Marco

Thank you for your attention!

- Modular reasoning about value sensitivity for concurrent programs
  - Independently of timing
  - Sound on real hardware
- Key idea is to exploit commutativity modulo abstraction
- Proved sound in Isabelle/HOL, automated in prototype verifier
- Will be presented at PLDI'23 by Marco