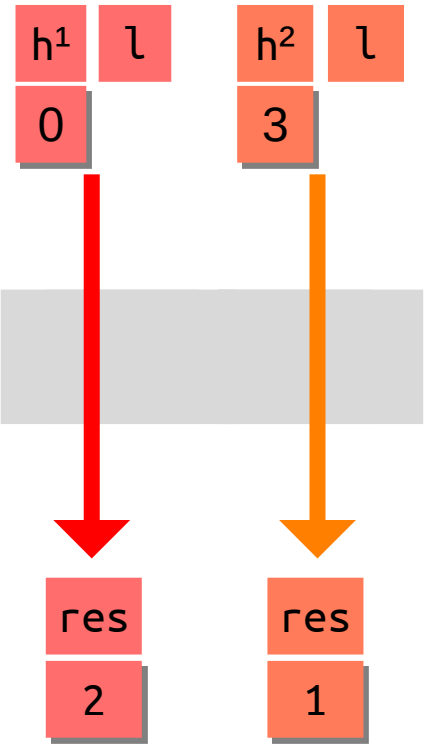
CommCSL: Proving Information Flow Security for Concurrent Programs Using Abstract Commutativity

Marco Eilers
Thibault Dardinier
Peter Müller

ETH zürich

Value Leaks: Noninterference

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



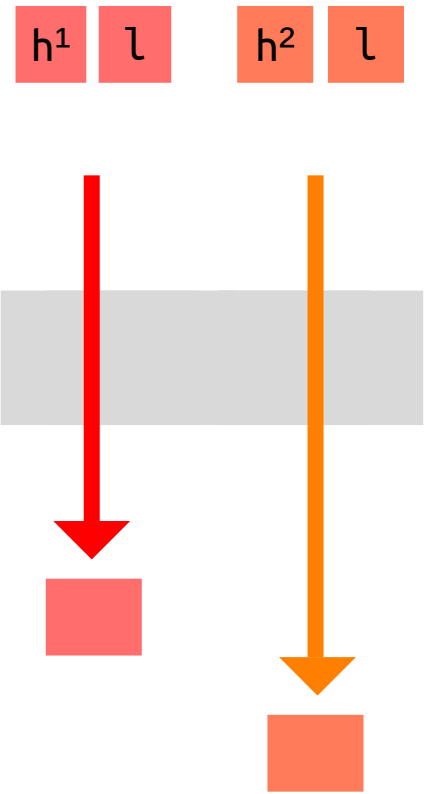
Timing side channels

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

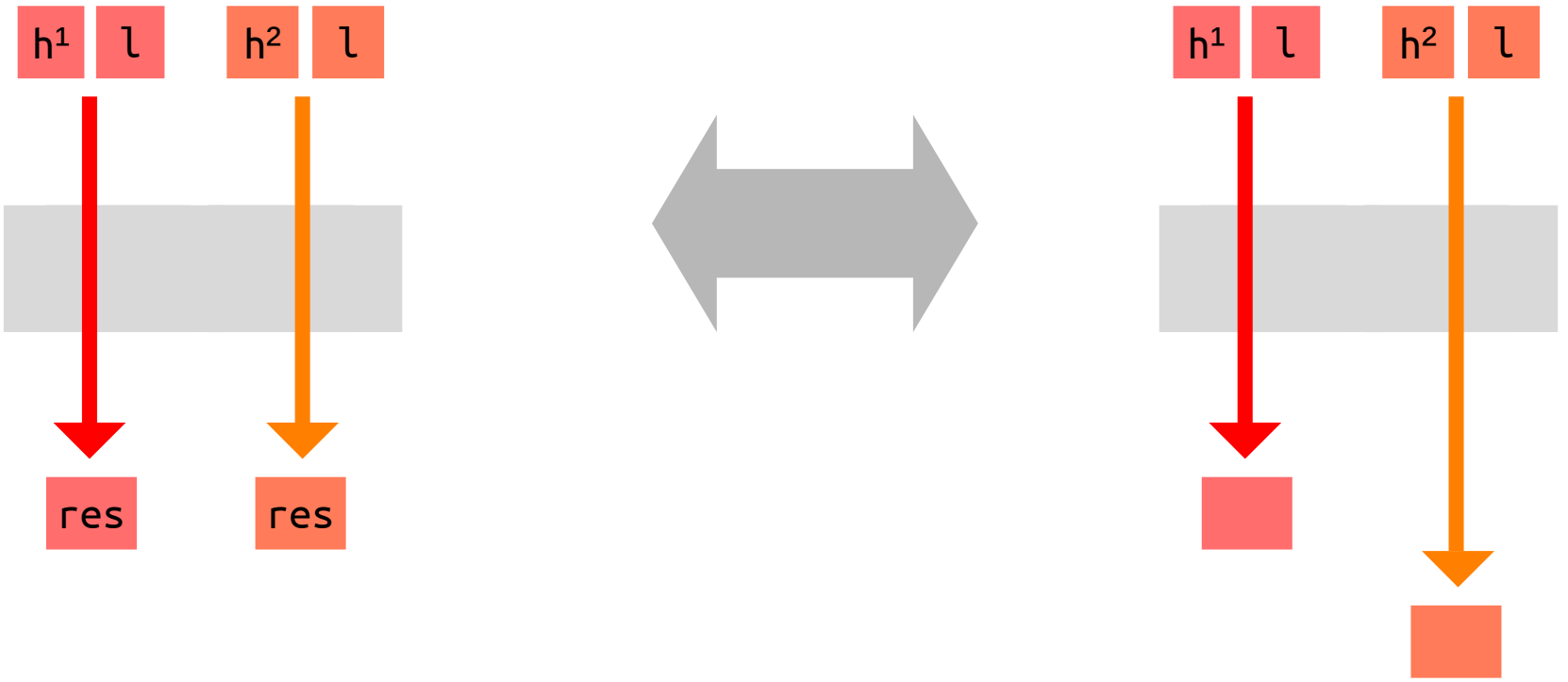
Timing side channels

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

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

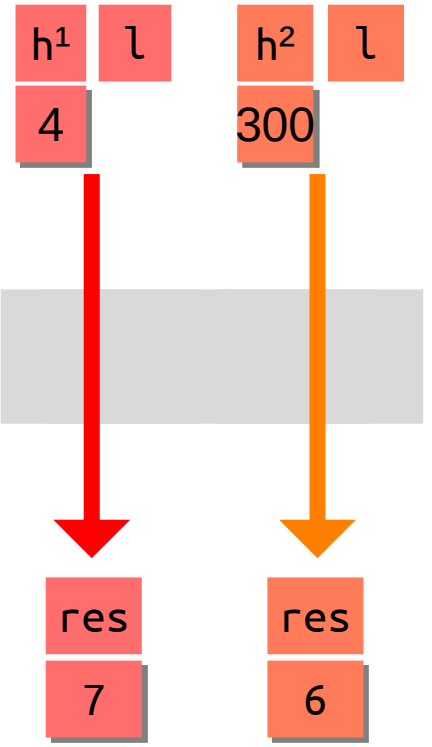


Values vs. timing



Shared-Memory Concurrency Ruins Everything

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

          return shared
```



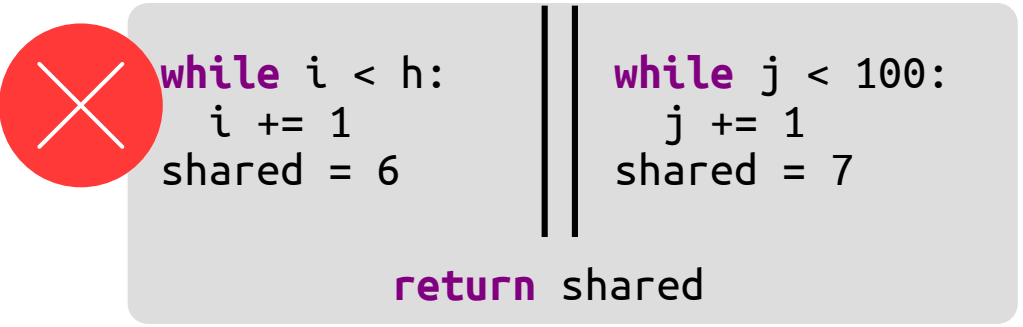
Timing channel + Concurrency =
Value Channel

Shared-Memory Concurrency Ruins Everything

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

return shared
```

Secret-dependent timing
influences
order of modifications of shared data,
which influences
the final result value

Existing (Modular) Solutions



~~Secret-dependent timing~~
influences
order of modifications of shared data,
which influences
the final result value

Existing (Modular) Solutions

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

~~Secret-dependent timing~~
influences
order of modifications of shared data,
which influences
the final result value

Goal:
Reason about ***values***
in concurrent programs
without reasoning about ***timing***

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

Our Solution

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

Secret-dependent timing
influences
order of modifications of shared data,
which influences
the final result value

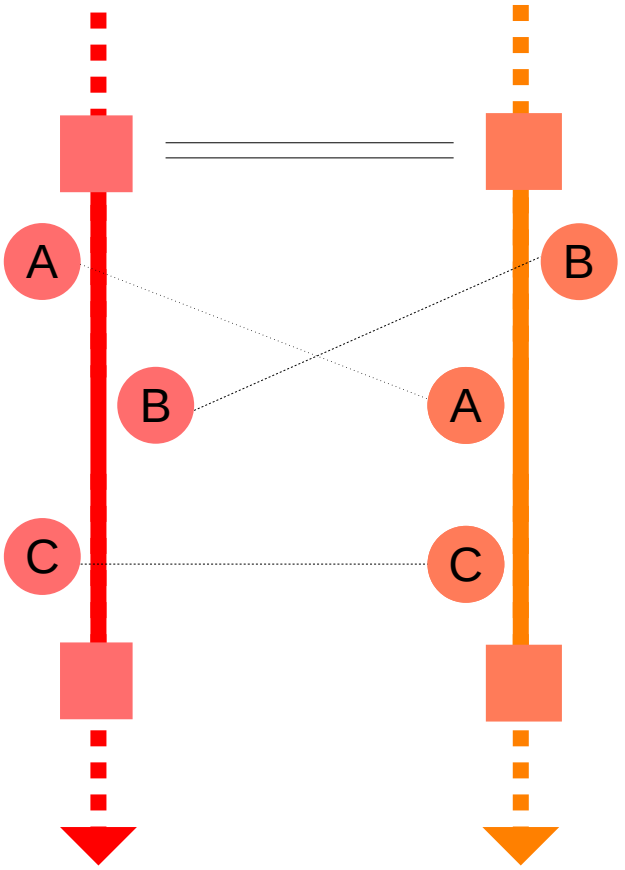
Key Idea:
Order does not influence result if modifications
commute

Our Solution

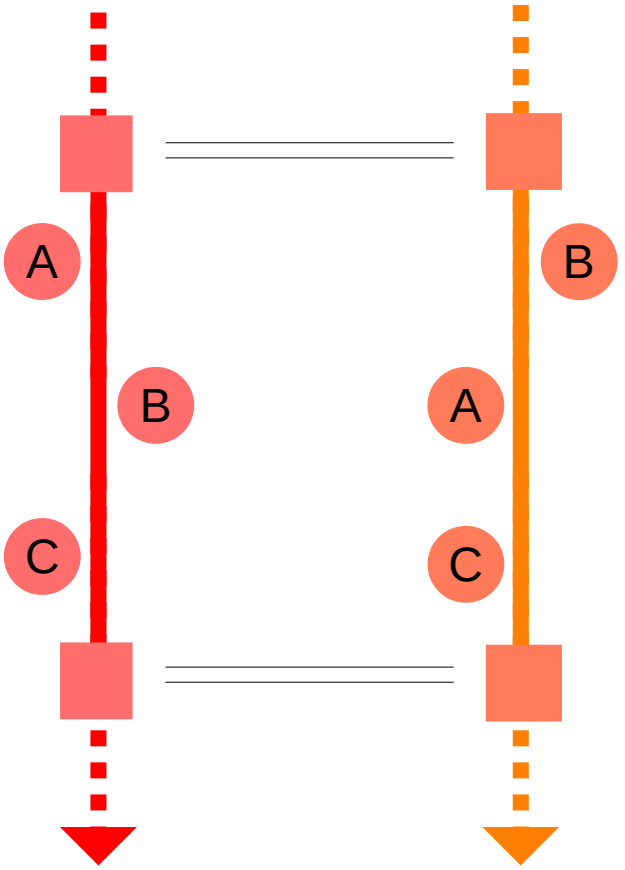
```
          shared = l
while i < h:      ||  while j < 100:
  i += 1          ||    j += 1
atomic:           ||  atomic:
  shared += 6     ||    shared += 7
          return shared
```

Secret-dependent timing
influences
order of modifications of shared data,
which influences
the final result value

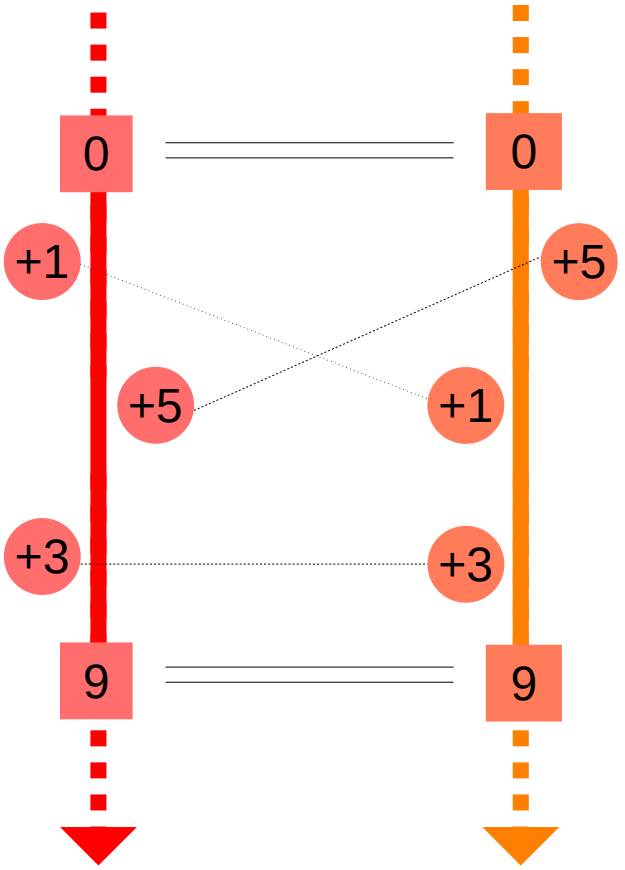
Basic Solution



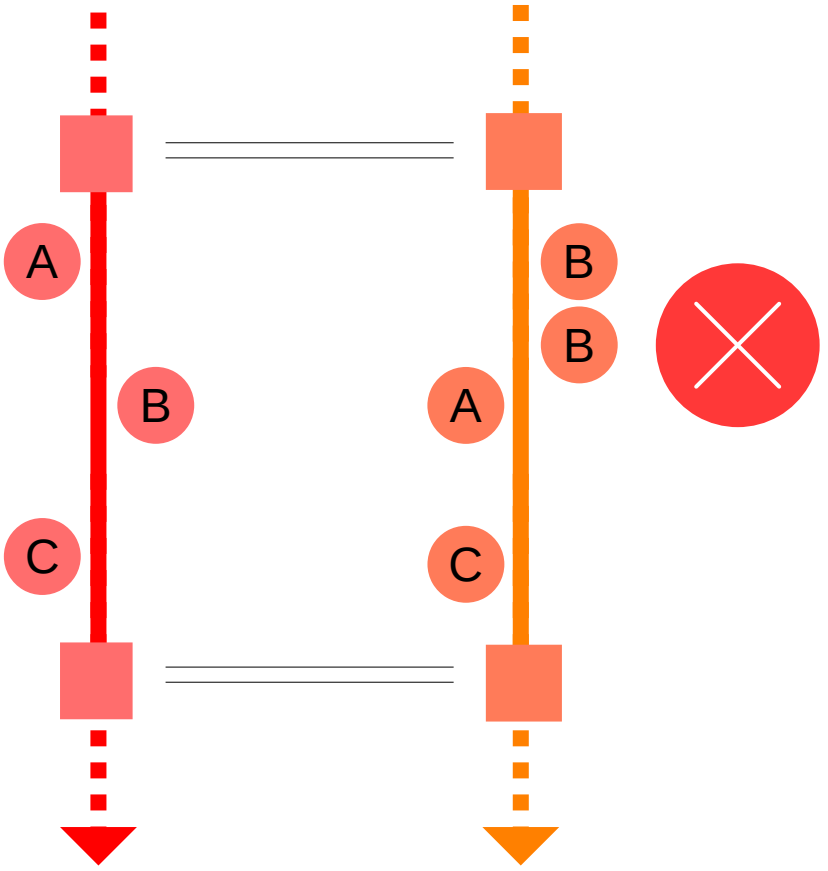
Basic Solution



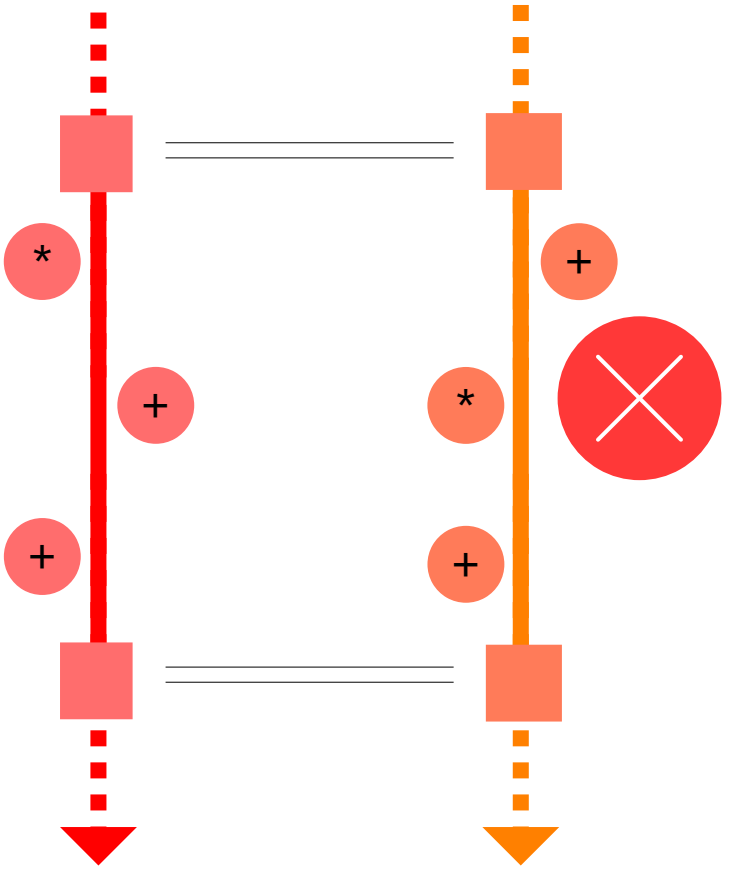
Basic Solution



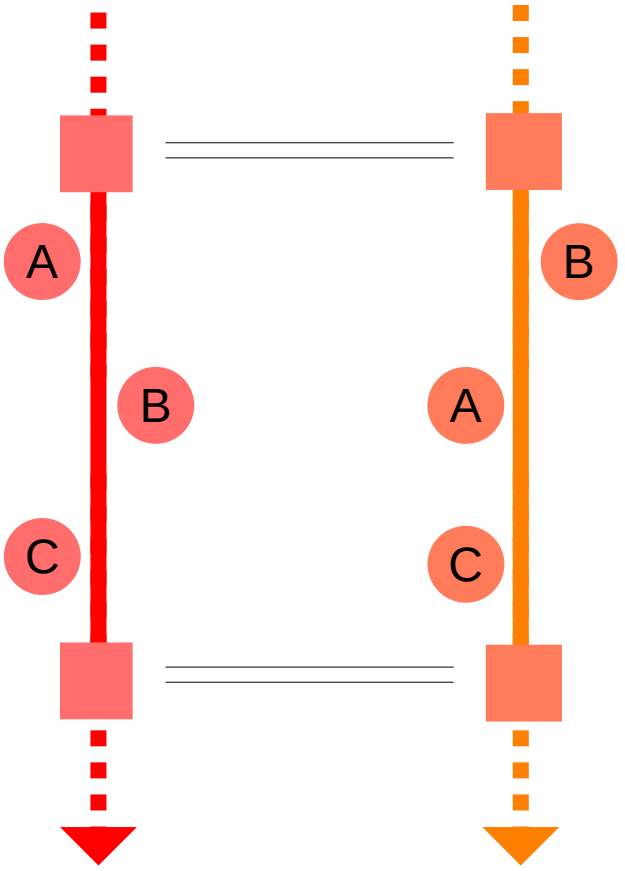
Basic Solution



Basic Solution

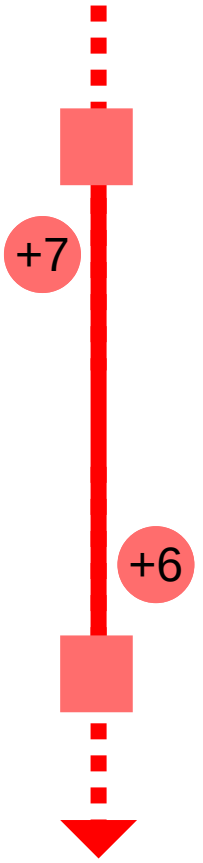


Basic Solution



Concurrent Separation Logic

```
                shared = l

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


              return shared
```



Concurrent Separation Logic

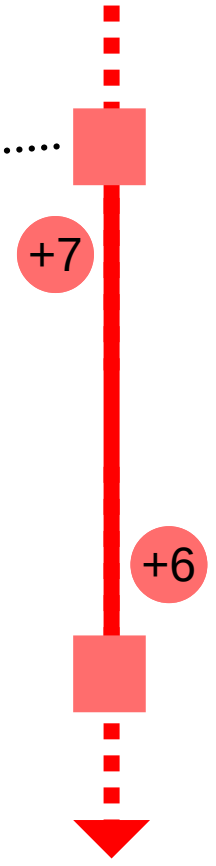
```
shared = l
share
```

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

atomic:
  shared += 6
```

||

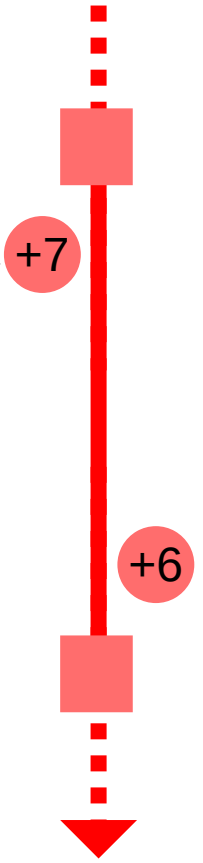
```
while j < 100:
  j += 1

atomic:
  shared += 7
```

```
return shared
```



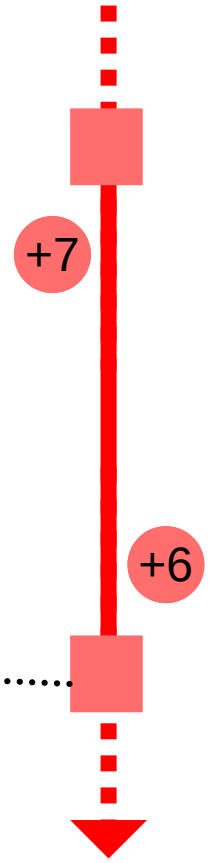
Concurrent Separation Logic

```
shared = l
share

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

return shared
```

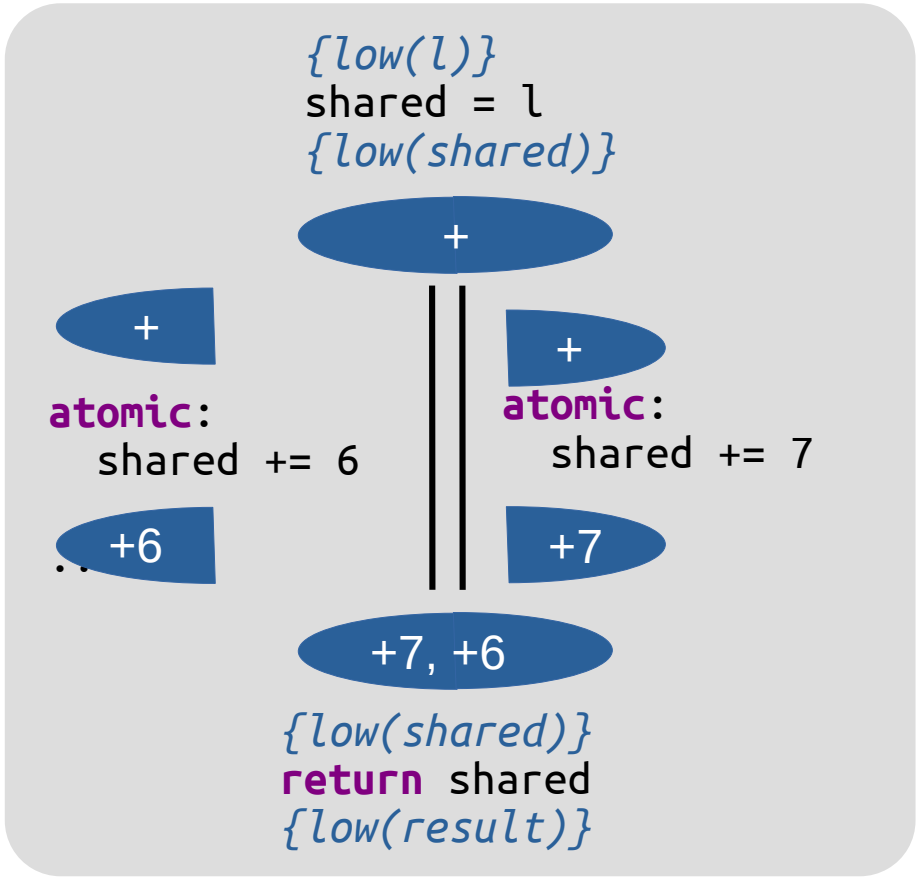


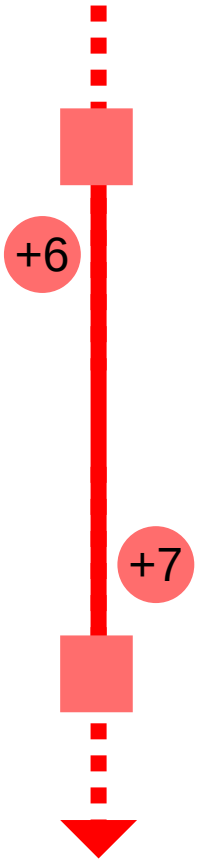
Concurrent Separation Logic

```
shared = l
share

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

unshare
return shared
```



CommCSL


{low(l)}
shared = l
{low(shared)}
+
+
atomic:
shared += 6
+6
+
atomic:
shared += 7
+7
+7, +6
{low(shared)}
return shared
{low(result)}


+6
+7

We can do better.

We can do better

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

Internal timing differences
influence
order of modifications of shared data,
which influences
the final result value

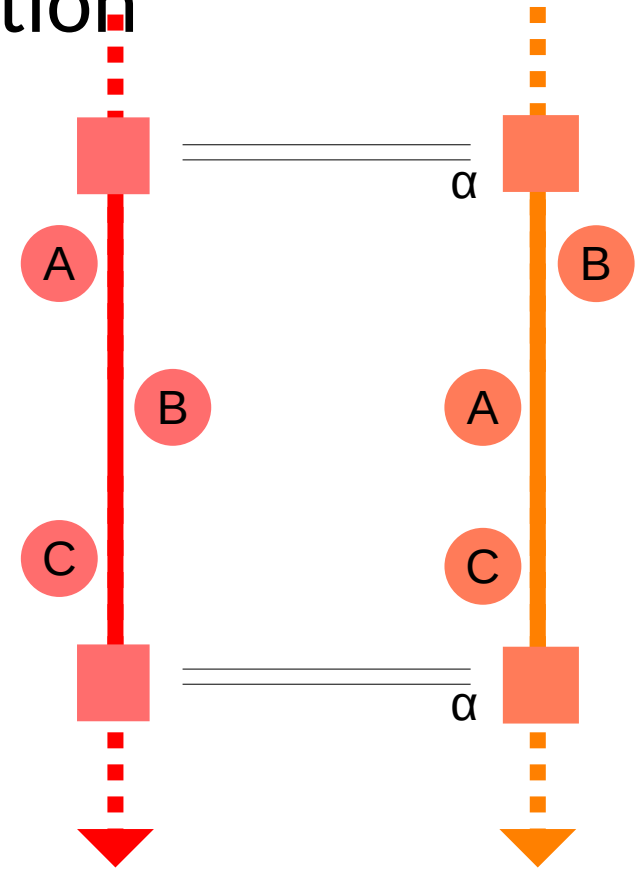
We can do better

```
          shared = new Map()
while i < h:          ||  while j < 100:
  i += 1              ||    j += 1
atomic:               ||  atomic:
  shared.put(1,6)     ||    shared.put(1,7)
          return shared.keySet()
```

Internal timing differences
influence
order of modifications of shared data,
which influences
the final result value

Key Idea:
Commutativity ***modulo abstraction.***

Improved Solution



CommCSL

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

- Other features:
  - More complete support for non-symmetric concurrency

- Formalized and proved sound in Isabelle/HOL
  - Challenging soundness argument distinct from existing logics

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

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

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

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

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

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

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

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

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

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

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

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

HyperViper

- Automated, SMT-based verifier
  - Based on Viper verification infrastructure and Z3
  - Relational reasoning using Modular Product Programs
- User provides resource specifications, pre- and postconditions, invariants
- Dynamic thread creation, multiple shared resources,

```
lockType MapLock {
  type MyMap[Int, Int]
  invariant(l, v) = [l.lockMap |-> ?mp && isMap(mp)
                     && v == mapValue(mp)]
  alpha(v): Set[Int] = keys(v)

  actions = [(Put, Pair[Int, Int], duplicable)]

  action Put(v, arg)
    requires low(fst(arg))
  { (put(v, fst(arg), snd(arg))) }
}
```

Evaluation

Example	Data structure	Abstraction	LOC	Ann.	T
Counter	Integer	None	84	39	2.36
Accumulator-Add	Integer	None	84	39	2.53
Bitwise-And	Integer	None	84	39	2.48
All-High	Integer	Constant	84	39	2.49
List-Append-Mean	List	Mean	103	55	2.62
List-Append-Multiset	List	Multiset	90	43	2.65
List-Append-Length	List	Length	88	42	2.60
Set-Add-Tree	Set	None	210	107	24.52
Set-Add-List	Set	None	151	67	3.08
Map-Keyset	Map	Key set	89	40	2.63
Map-Disjoint	Map	None	93	45	2.05
Map-Sum	Map	None	118	52	2.71
Map-Conditional	Map	None	116	46	2.60
1-Producer-1-Consumer	Queue	Consumed sequence	120	59	1.97
Pipeline	Two queues	Consumed sequences	162	73	2.40
2-Producers-2-Consumers	Queue	Produced multiset	213	111	6.23

Conclusion

- 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, automated in prototype verifier

- Should be on arXiv in a couple of weeks

- Future work
  - Fine-grained concurrency
  - Static analysis etc.