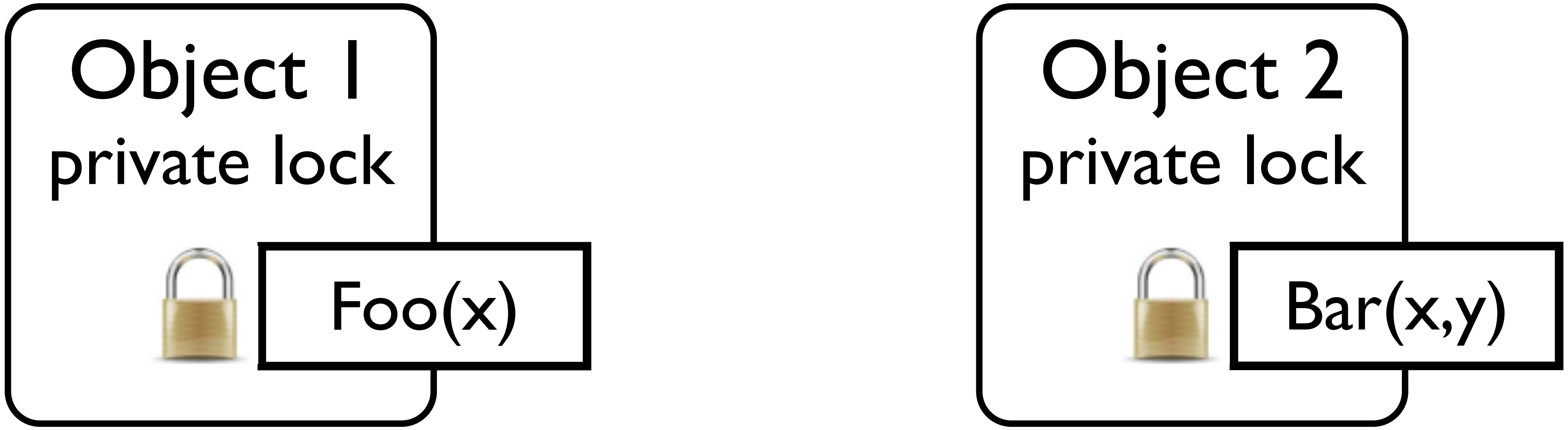
Open issues in extensible libraries

Aaron Turon
Northeastern University



Object 1

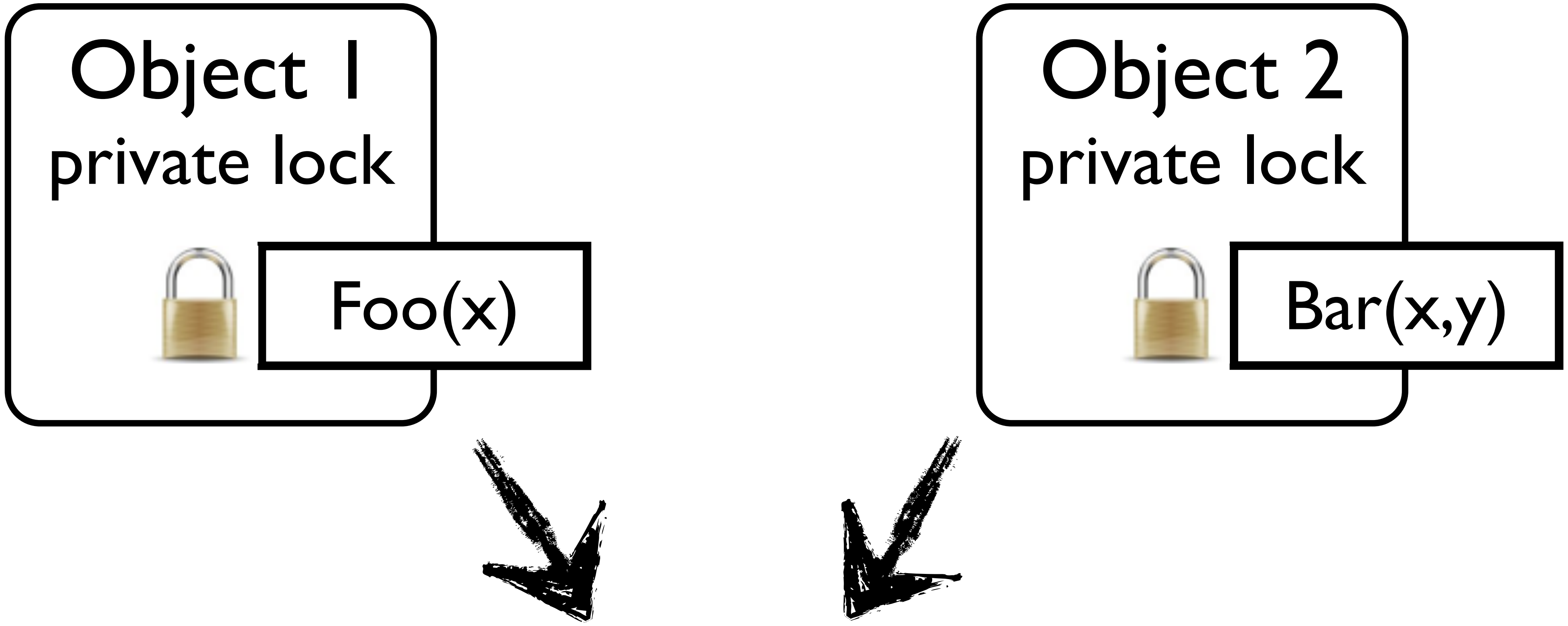
private lock

Foo(x)

Object 2

private lock

Bar(x,y)



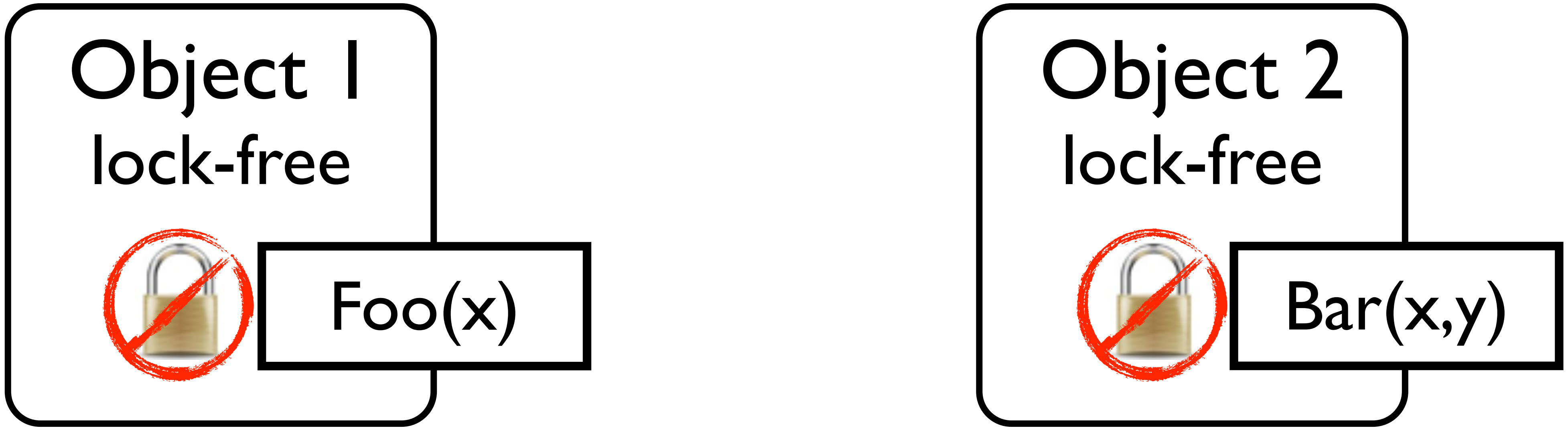
Object 1
private lock

Foo(x)

Object 2
private lock

Bar(x,y)

```
atomic {
   o1.Foo(x);
   o2.Bar(x, y);
}
```



Object 1

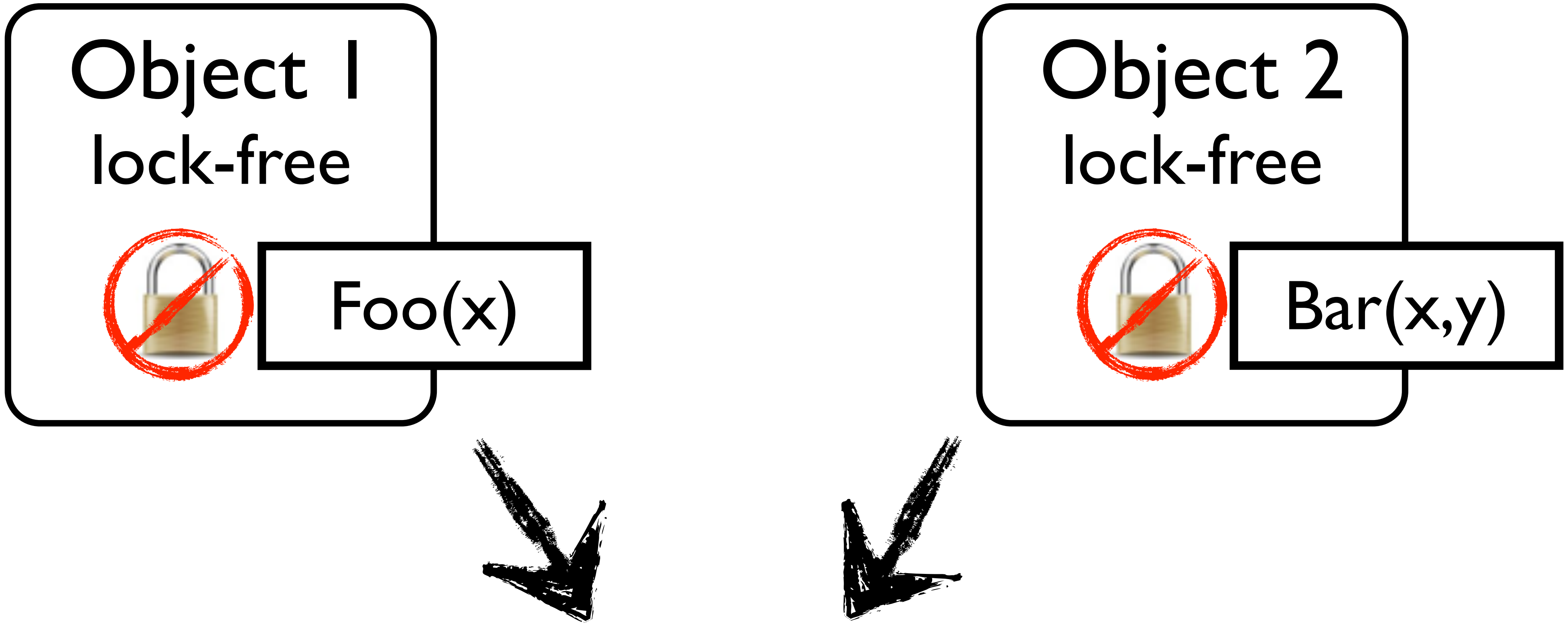
lock-free

Foo(x)

Object 2

lock-free

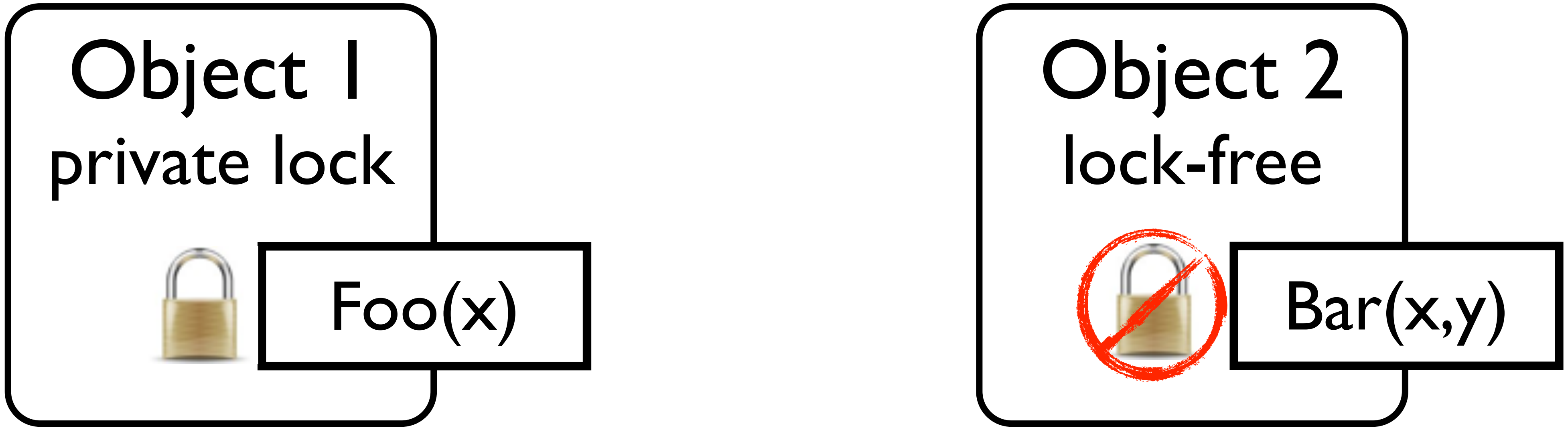
Bar(x,y)



Object 1
lock-free
Foo(x)

Object 2
lock-free
Bar(x,y)

```
atomic {
    o2.Bar(o1.Foo(x), y);
}
```

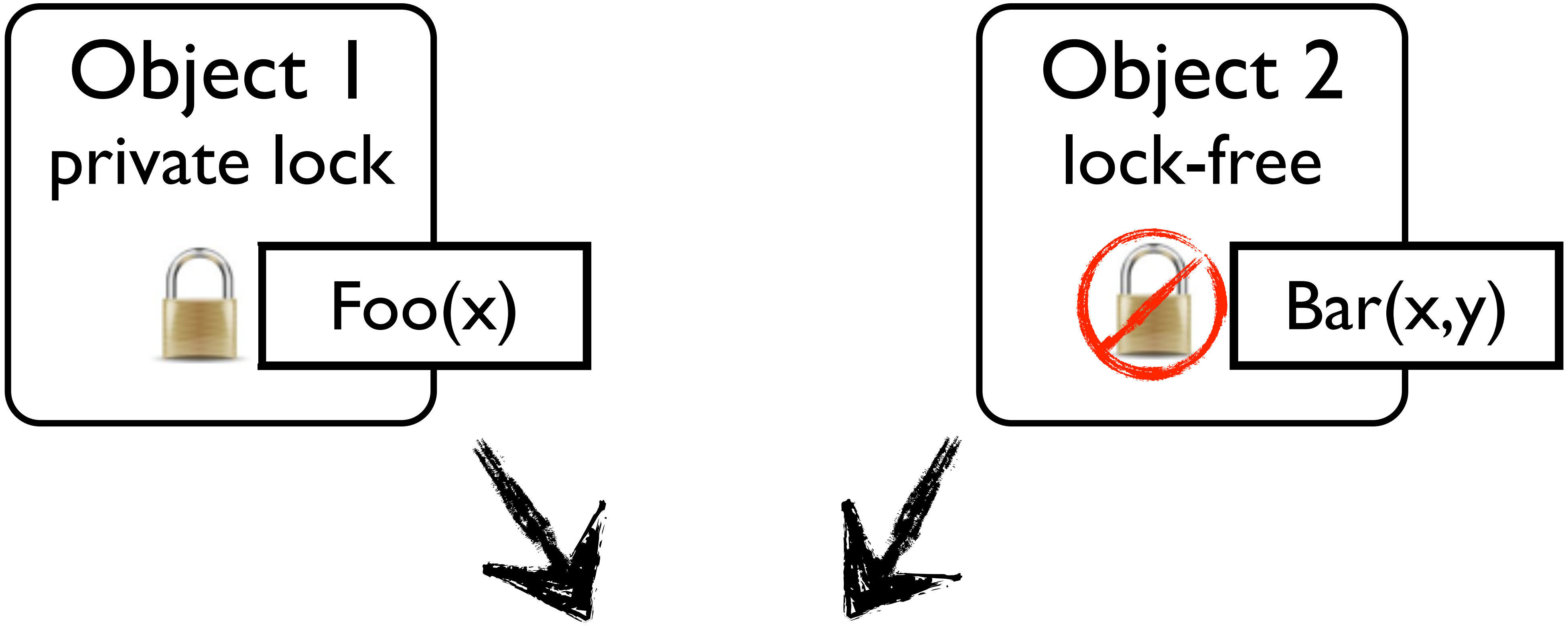


Object 1
private lock

Foo(x)

Object 2
lock-free

Bar(x,y)



Object 1

private lock

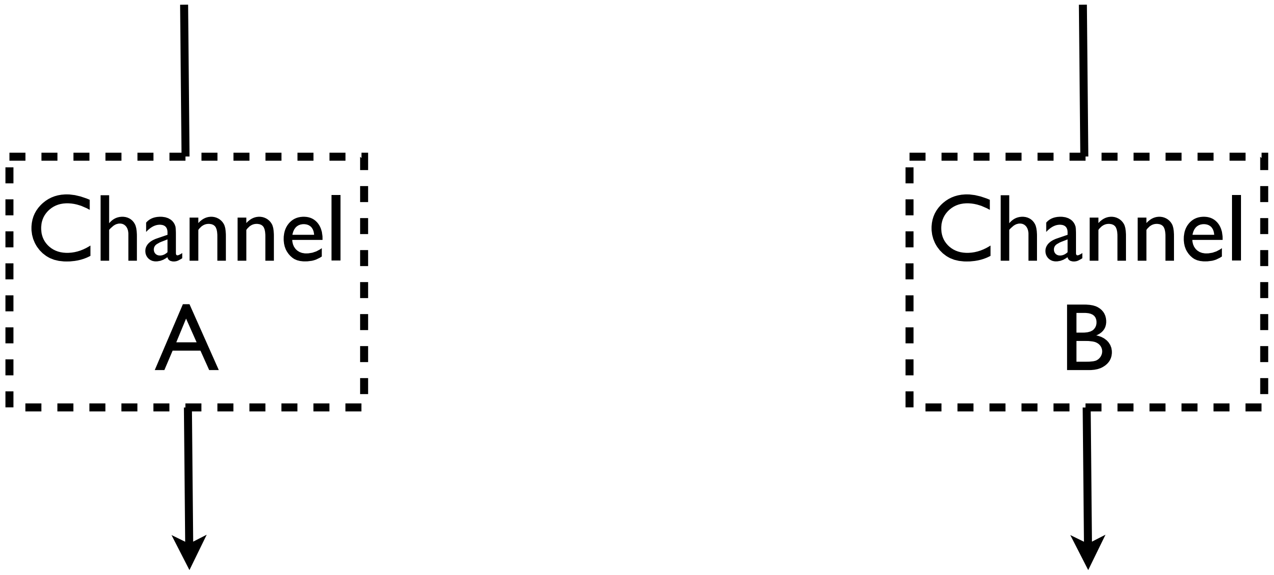
Foo(x)

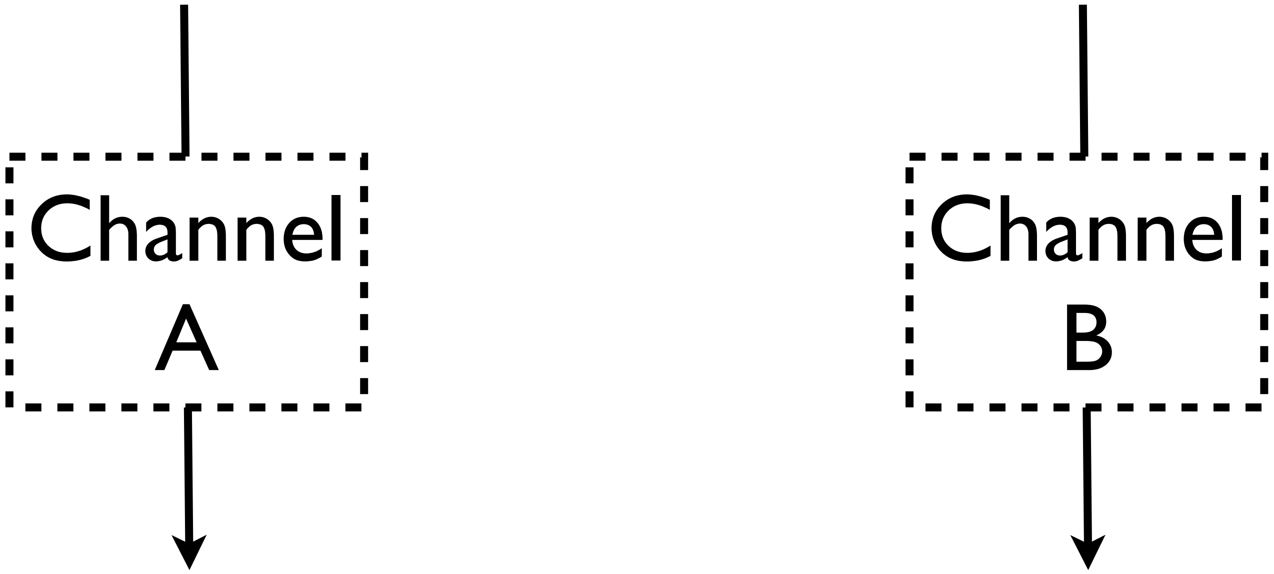
Object 2

lock-free

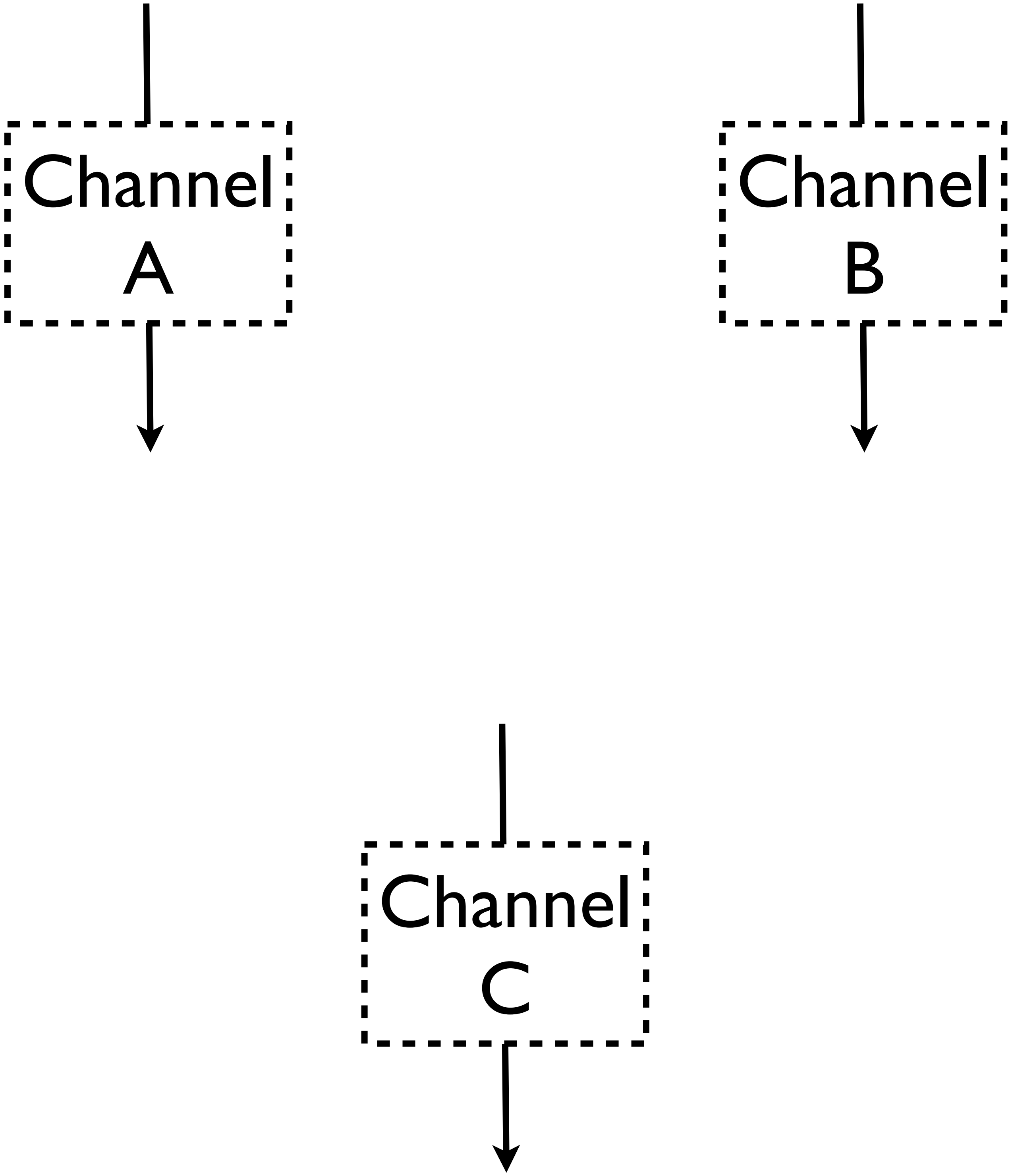
Bar(x,y)

```
atomic {
   o2.Bar(o1.Foo(x), y);
}
```

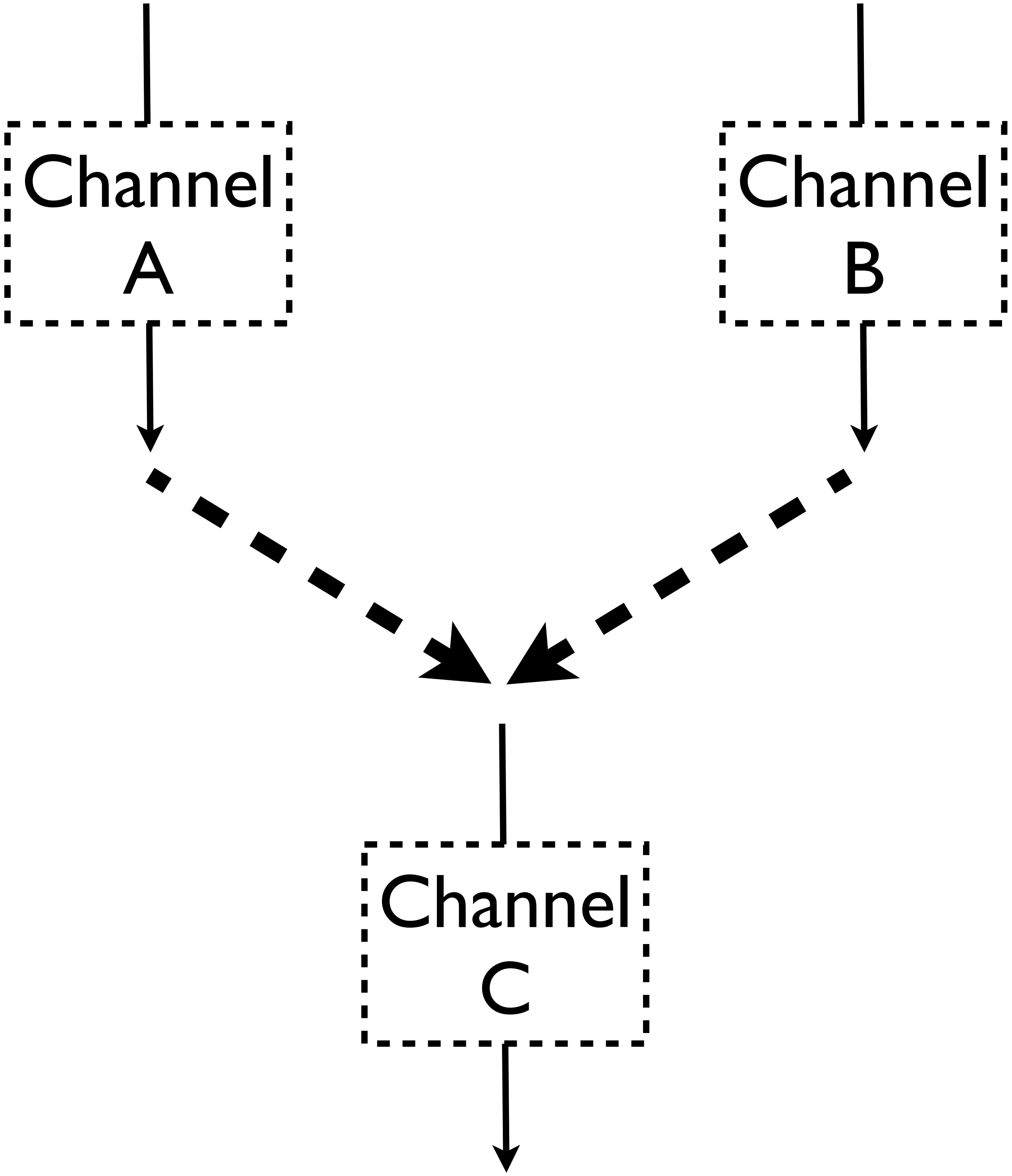

Channel A
Channel B

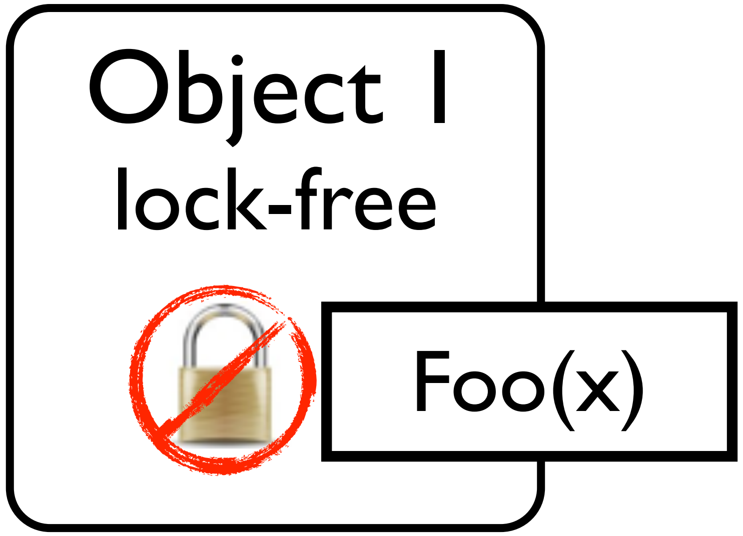


```
atomic {
   (receive(A), receive(B))
}
```



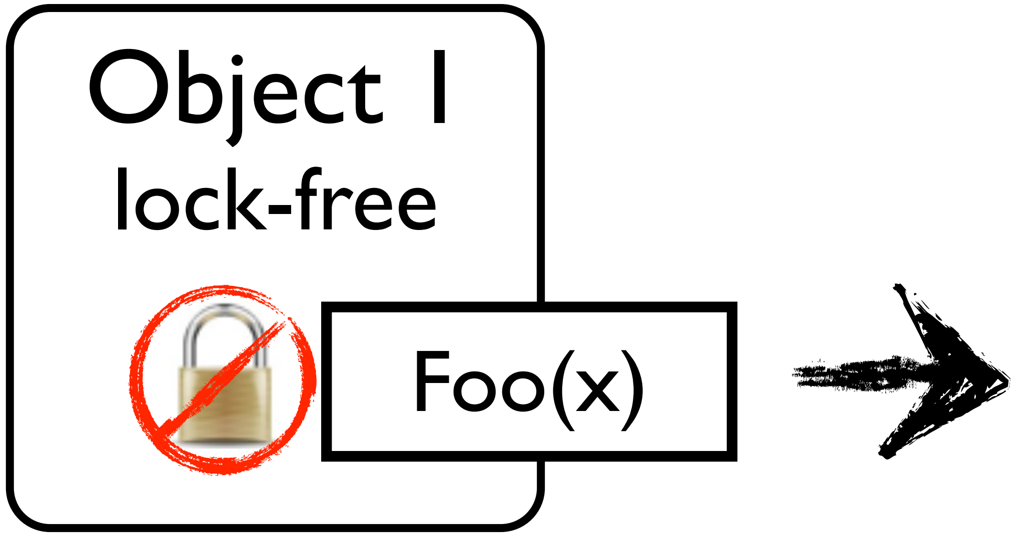
Channel A
Channel B
Channel C


Channel A
Channel B
Channel C



Object 1
lock-free

Foo(x)



Object 1

lock-free

Foo(x)

```
atomic {
   if (o1.Foo(x) == null)
     block;
}
```

Cards on the table

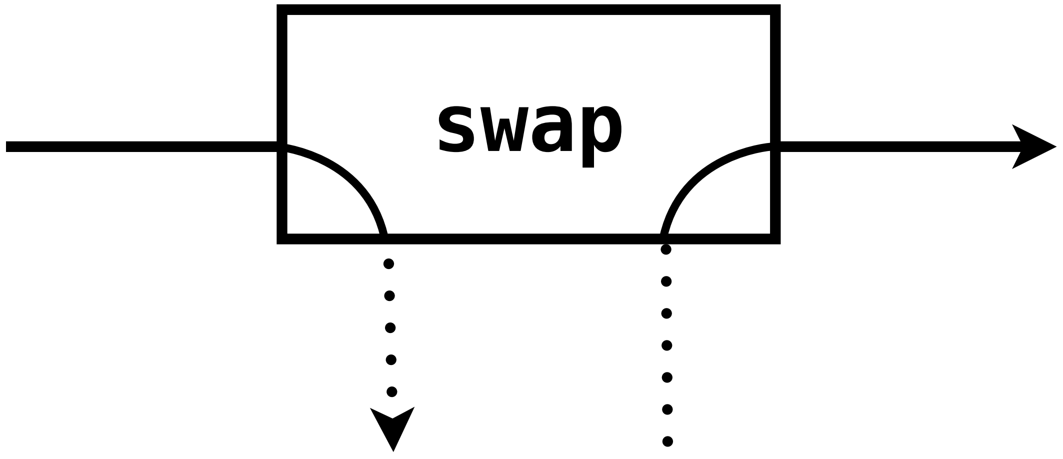
- **Assumption:** programmers will use a mixture of concurrency paradigms
- **Assumption:** programmers want to compose code they do not control
- **Conclusion:** the semicolon is not enough

A (big!) open issue:

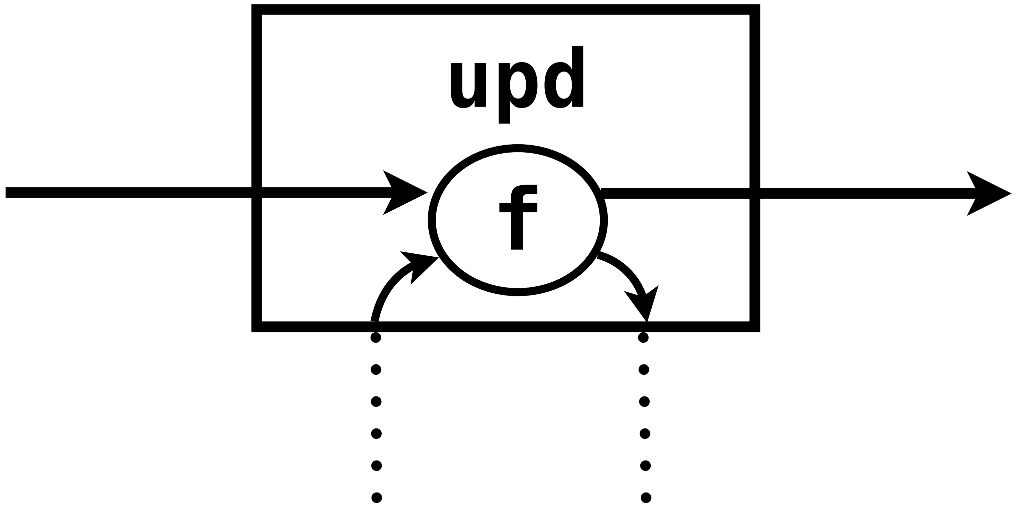
How do we support
abstraction and composition
across multiple paradigms,
without sacrificing performance?

My stake in the ground:
“Reagents”

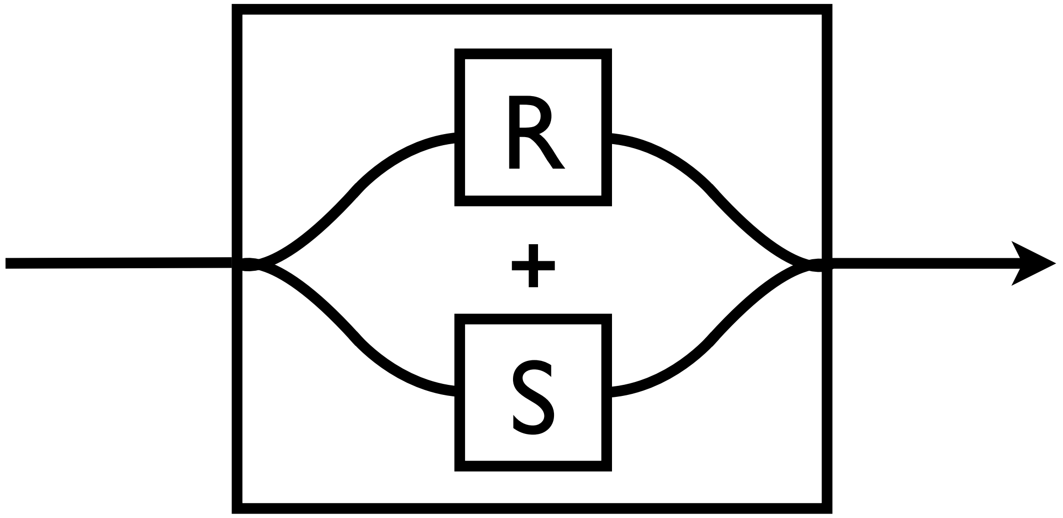
Message passing



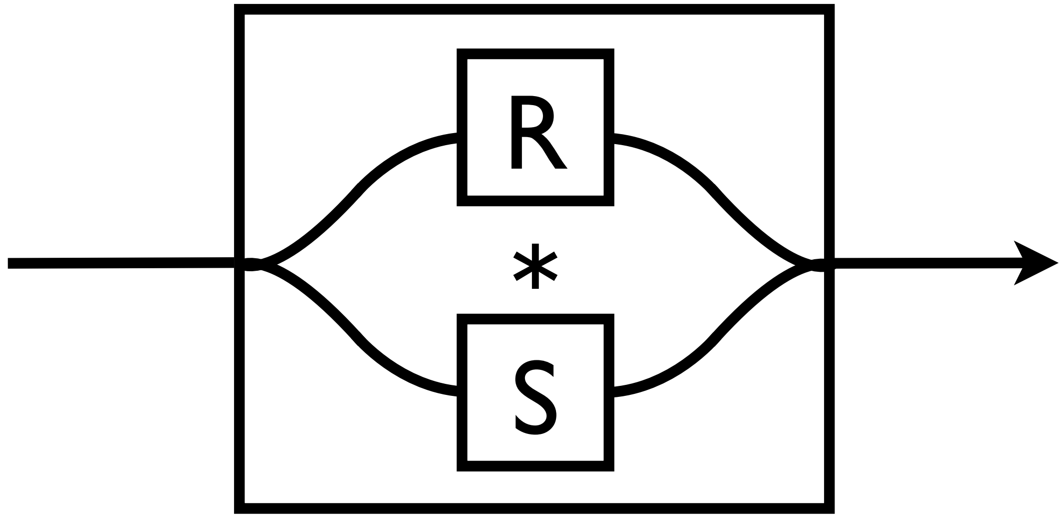
Shared state



Disjunction



Conjunction



```
class TreiberStack [A] {
  private val head = new Ref[List[A]](Nil)
  val push   = upd(head)(cons)
  val tryPop = upd(head)(trySplit)
  val pop    = upd(head)(split)
}
```

```
class TreiberStack [A] {
  private val head = new Ref[List[A]](Nil)
  val push   = upd(head)(cons)
  val tryPop = upd(head)(trySplit)
  val pop    = upd(head)(split)
}

class EliminationStack [A] {
  private val stack = new TreiberStack[A]
  private val (send, recv) = new Chan[A]
  val push = stack.push + swap(send)
  val pop  = stack.pop  + swap(recv)
}
```

Lessons from reagents

- Make composition is **pay-as-you-go**, *e.g.*, *k*CAS only when you use it
- Fully embrace underlying paradigms, even if it requires **escape hatches**
- Restrained ambitions: some compositions are **illegal** (*i.e.*, *ceci n'est pas une STM*)

But there’s more to learn

Isolation
Shared state

Interaction
Message passing

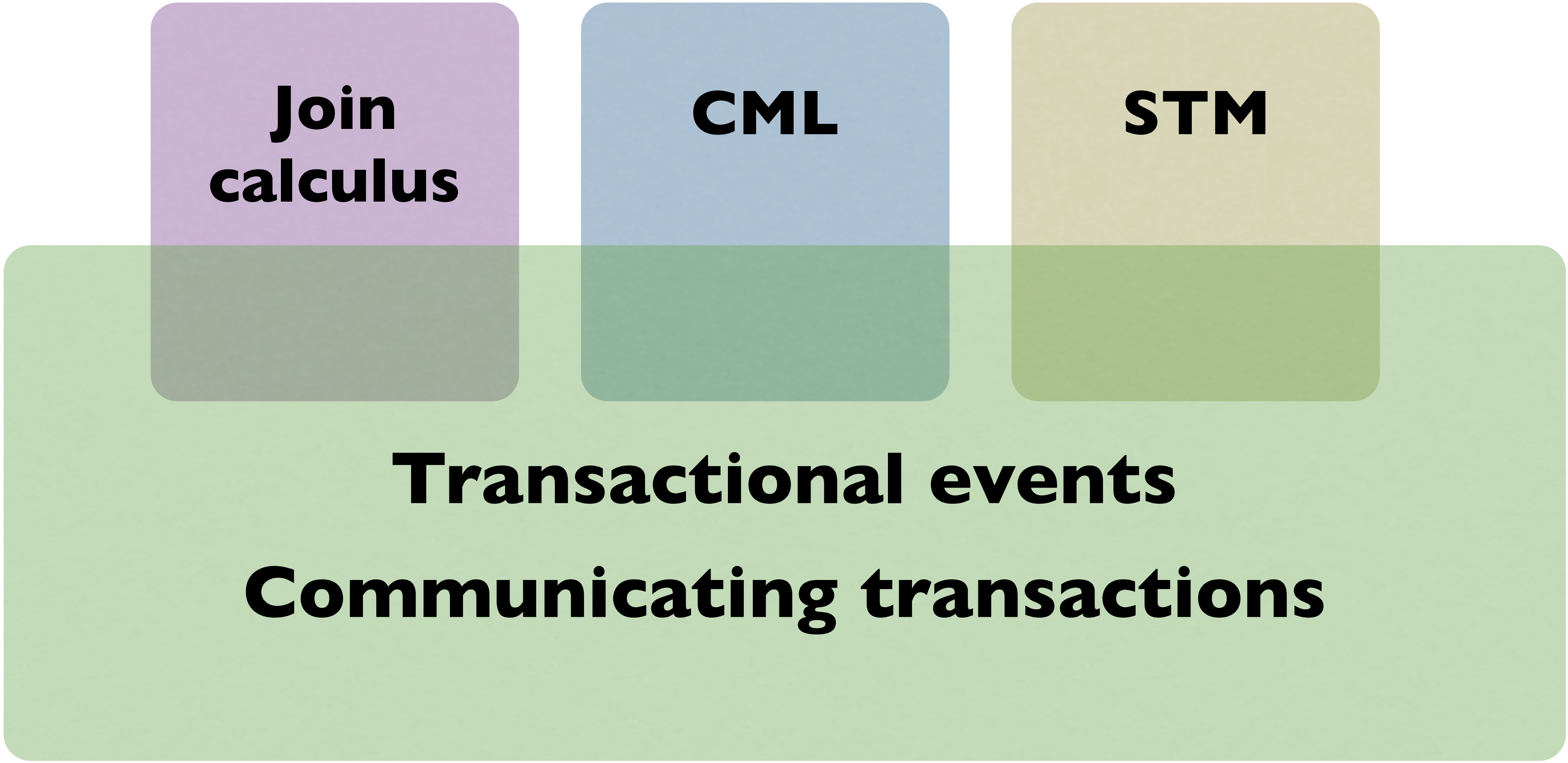
(A very small part of) The design space

Join calculus

CML

STM

(A very small part of)
The design space



Join calculus

CML

STM

Transactional events

Communicating transactions