Task Types for Pervasive Atomicity

Aditya Kulkarni, **Yu David Liu**
State University of New York at Binghamton
&
Scott Smith
Johns Hopkins University

October 2010 @OOPSLA

Task Types for Pervasive **Atomicity**

Aditya Kulkarni, **Yu David Liu**
State University of New York at Binghamton
&
Scott Smith
Johns Hopkins University

October 2010 @OOPSLA

Task Types for Pervasive Atomicity

Aditya Kulkarni, **Yu David Liu**
State University of New York at Binghamton
&
Scott Smith
Johns Hopkins University

October 2010 @OOPSLA

Task Types for Pervasive Atomicity

Aditya Kulkarni, **Yu David Liu**

State University of New York at Binghamton

&

Scott Smith

Johns Hopkins University

October 2010 @OOPSLA

Atomicity

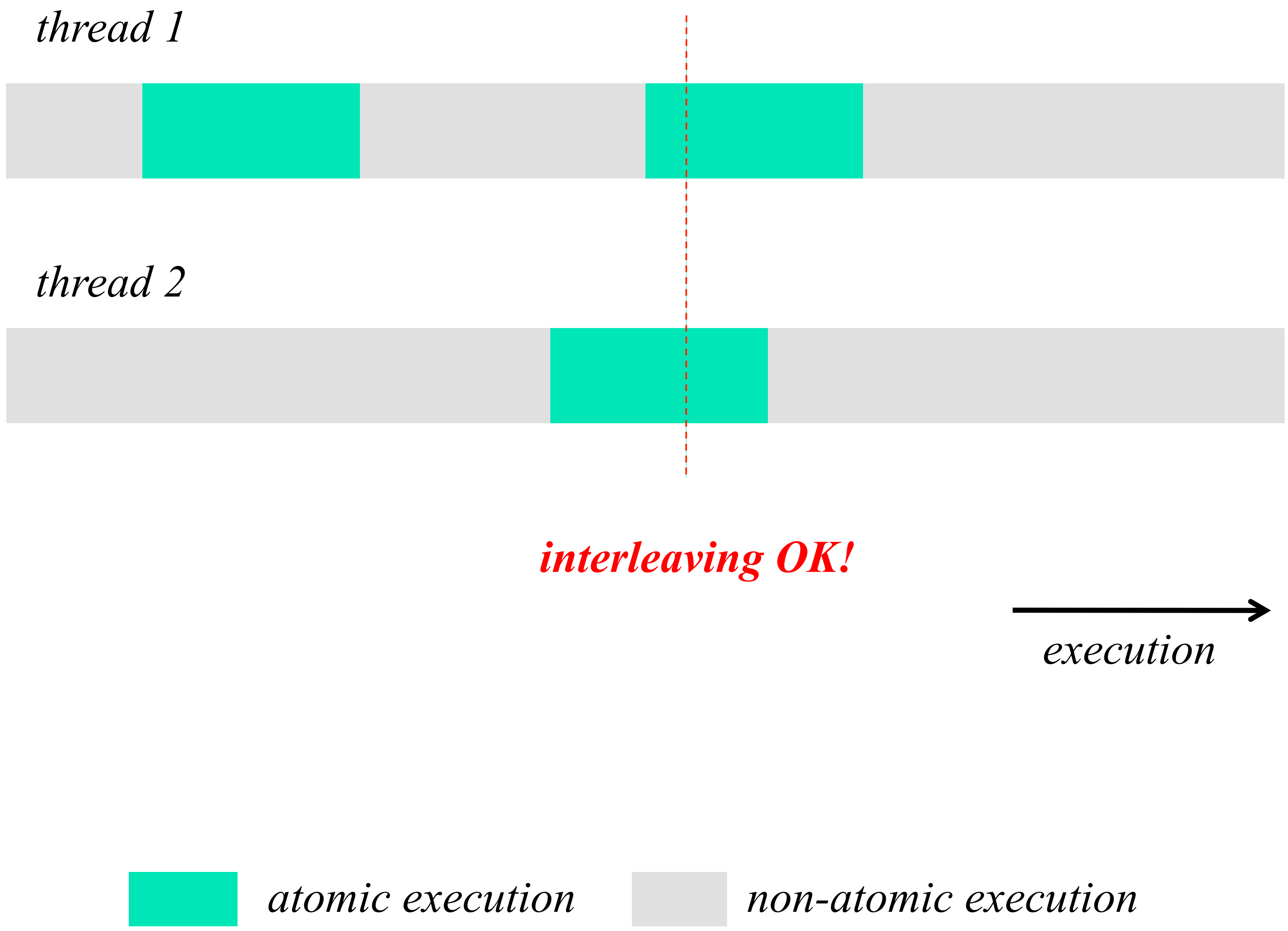
- Atomicity: a piece of code may interleave with others, but always behaves as if no interleaving happened
  - Important for program understanding and analysis for multi-core software
- A lot of existing work on implementation strategies:
  - pessimistic lock-based
  - optimistic transaction-based (STM, HTM)
- This talk largely independent on the choices of implementation strategies

Atomic Blocks: Deceptively Simple

```
class Bank {
   …
   void transfer (Account from, Account to, int amount) {
      …
      atomic {
            from.decrease(amount);
            to.increase(amount);
      }
   }

}
```

Atomic Blocks: Atomicity Zones as Islands



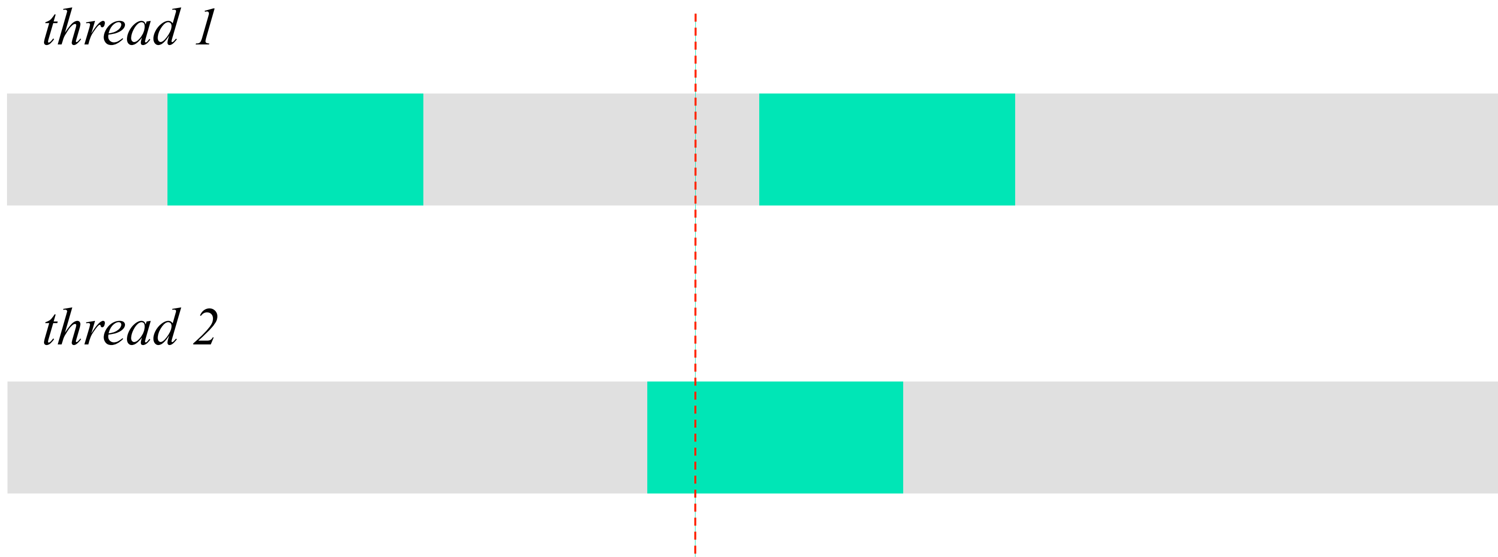
The Problems of Atomic Blocks (I)

```
class Bank {
   …
   void transfer (Account from, Account to, int amount) {
      …
      atomic {
            from.decrease(amount);
            to.increase(amount);
      }
   }

   void deposit (Account acc, int amount) {
            acc.increase(amount);
   }

}
```

The Problems of Atomic Blocks (I)



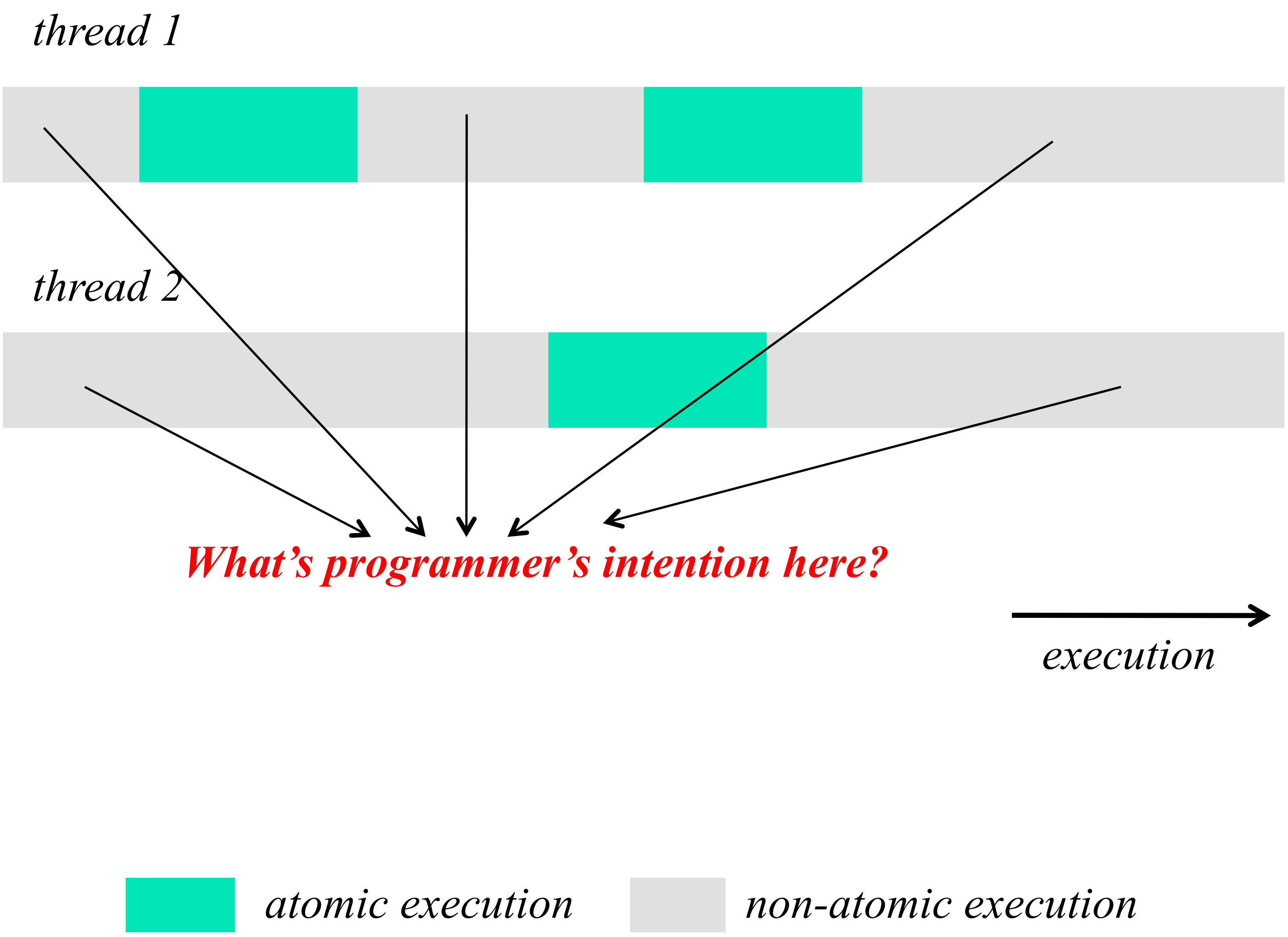
interleaving OK?

weak atomicity -> strong atomicity

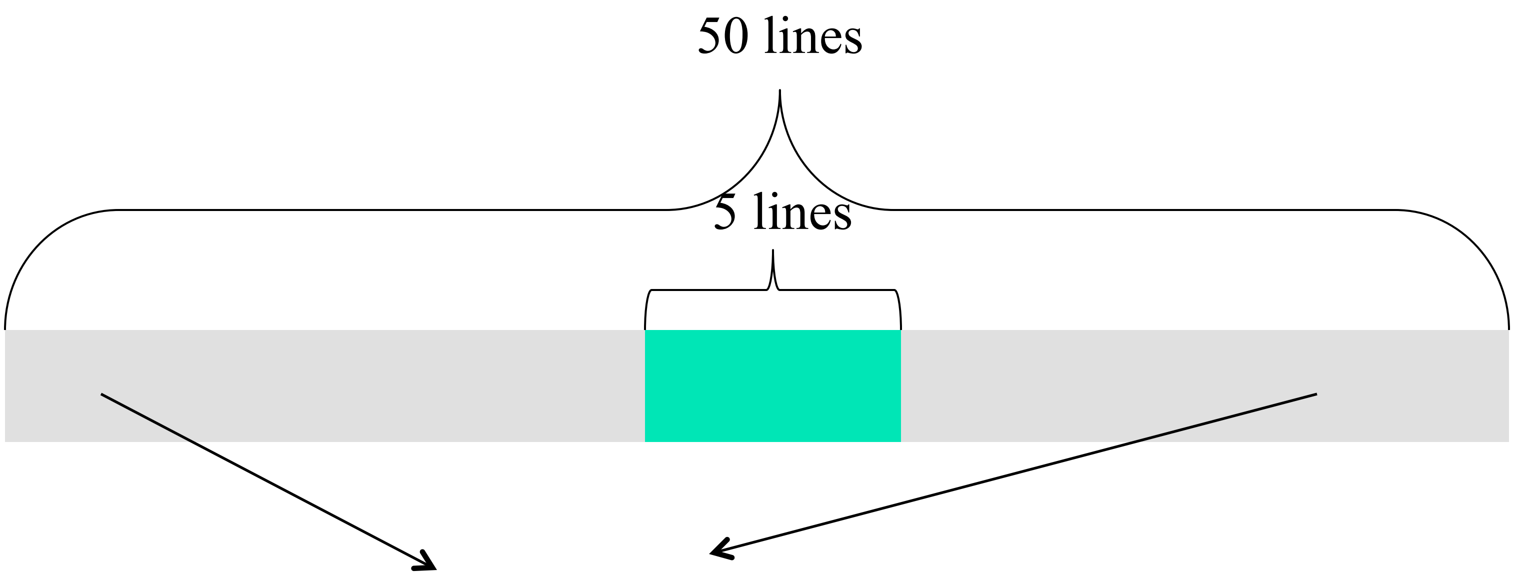
execution

atomic execution *non-atomic execution*

The Problems of Atomic Blocks (II)



The Problems of Atomic Blocks (II)

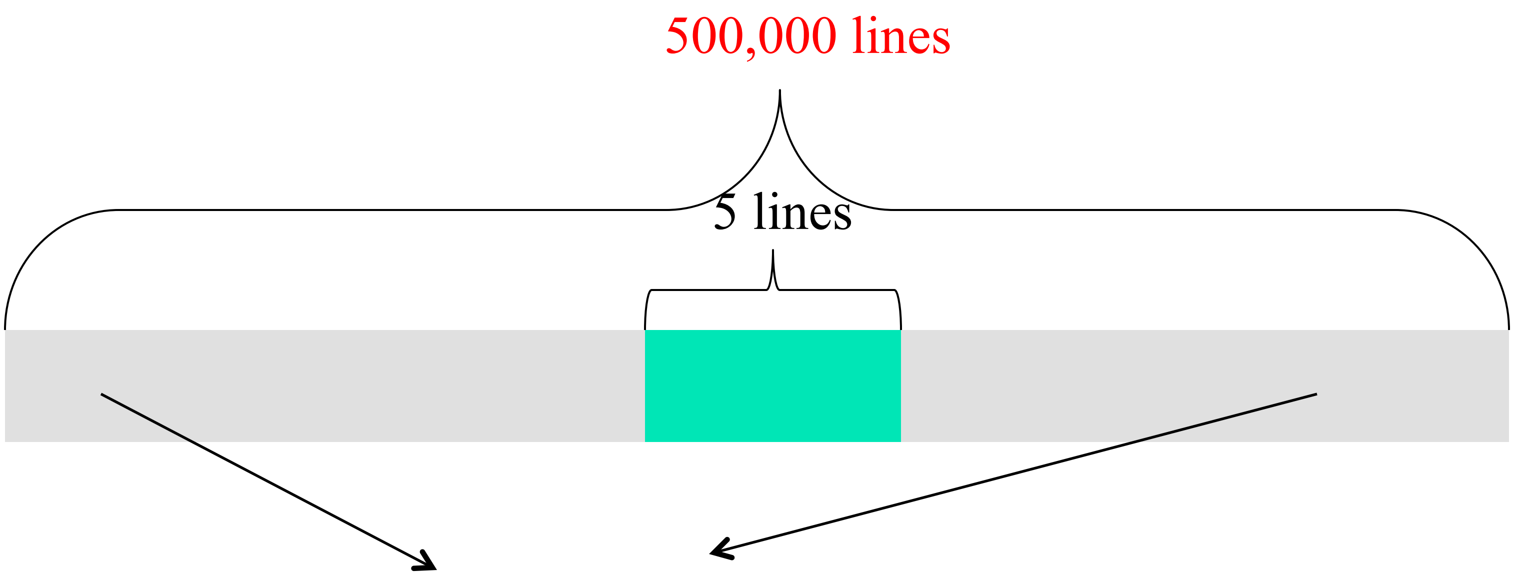


What's programmer's intention here?

"After carefully perusing 45 lines of code, I decide they are harmless to be outside atomic blocks, because:

1) *they never involve in interleaving, or*
2) *interleaving never changes their observable behavior, or*
3) *interleaving changes their behavior that I expect"*

The Problems of Atomic Blocks (II)



What's programmer's intention here?

"After carefully perusing 499,995 *lines of code, I decide they are harmless to be outside atomic blocks, because:*

1) *they never involve in interleaving, or*
2) *interleaving never changes their observable behavior, or*
3) *interleaving changes their behavior that I expect"*

The Problems of Atomic Blocks

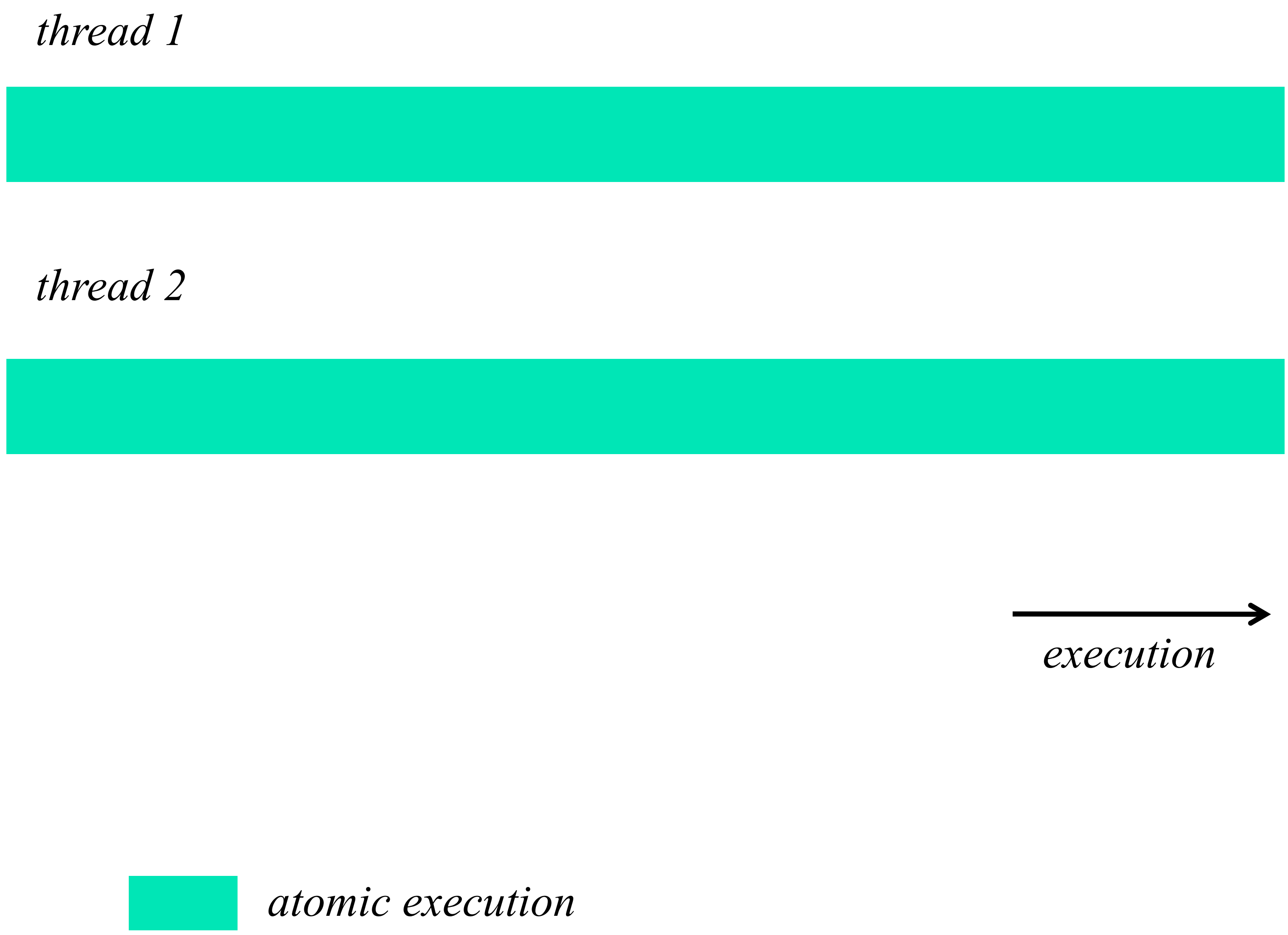
perhaps this island-building

language abstraction should be abandoned

execution

atomic execution *non-atomic execution*

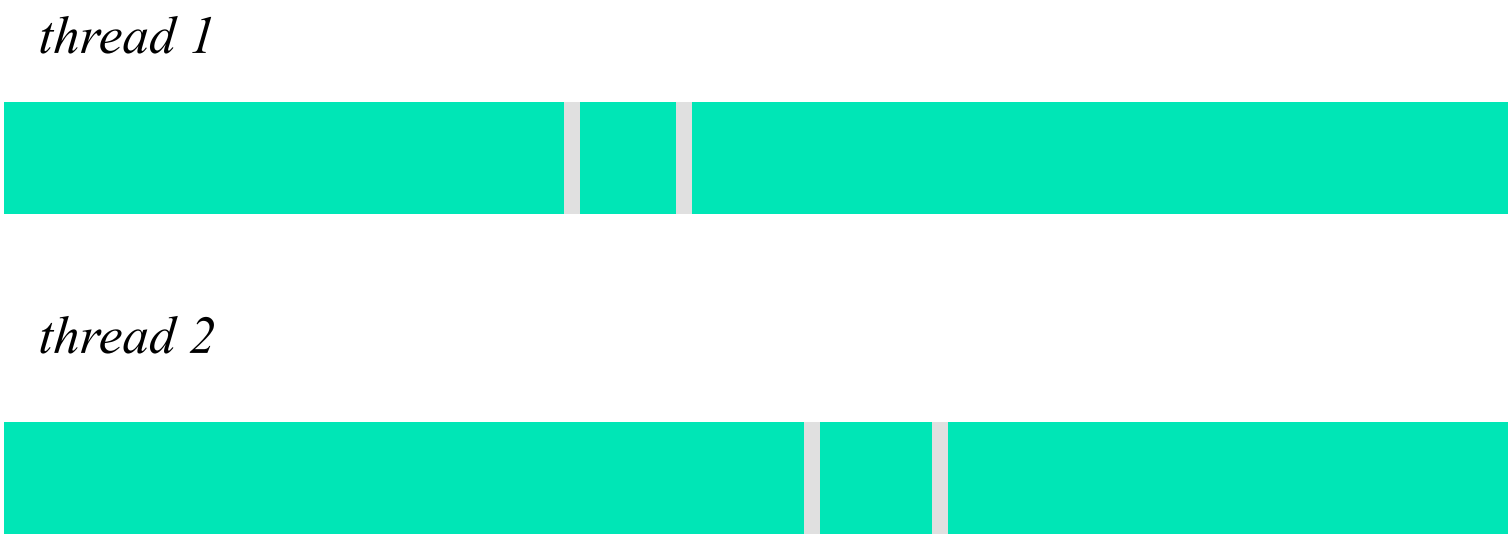
Let's Be Audacious



You Ask…

- Question 1: wouldn't threads accessing "exclusive resources" end up waiting each other for a long time (or rolling back a lot)?
- Question 2: familiar Java-like communication/sharing patterns, such as rendezvous?
- Question 3: "pervasive atomicity"??? You mean "pervasive run-time locks/transactions?

Atomicity Break Points



pervasive atomicity:

every line of code still lives in SOME atomic zone!

execution

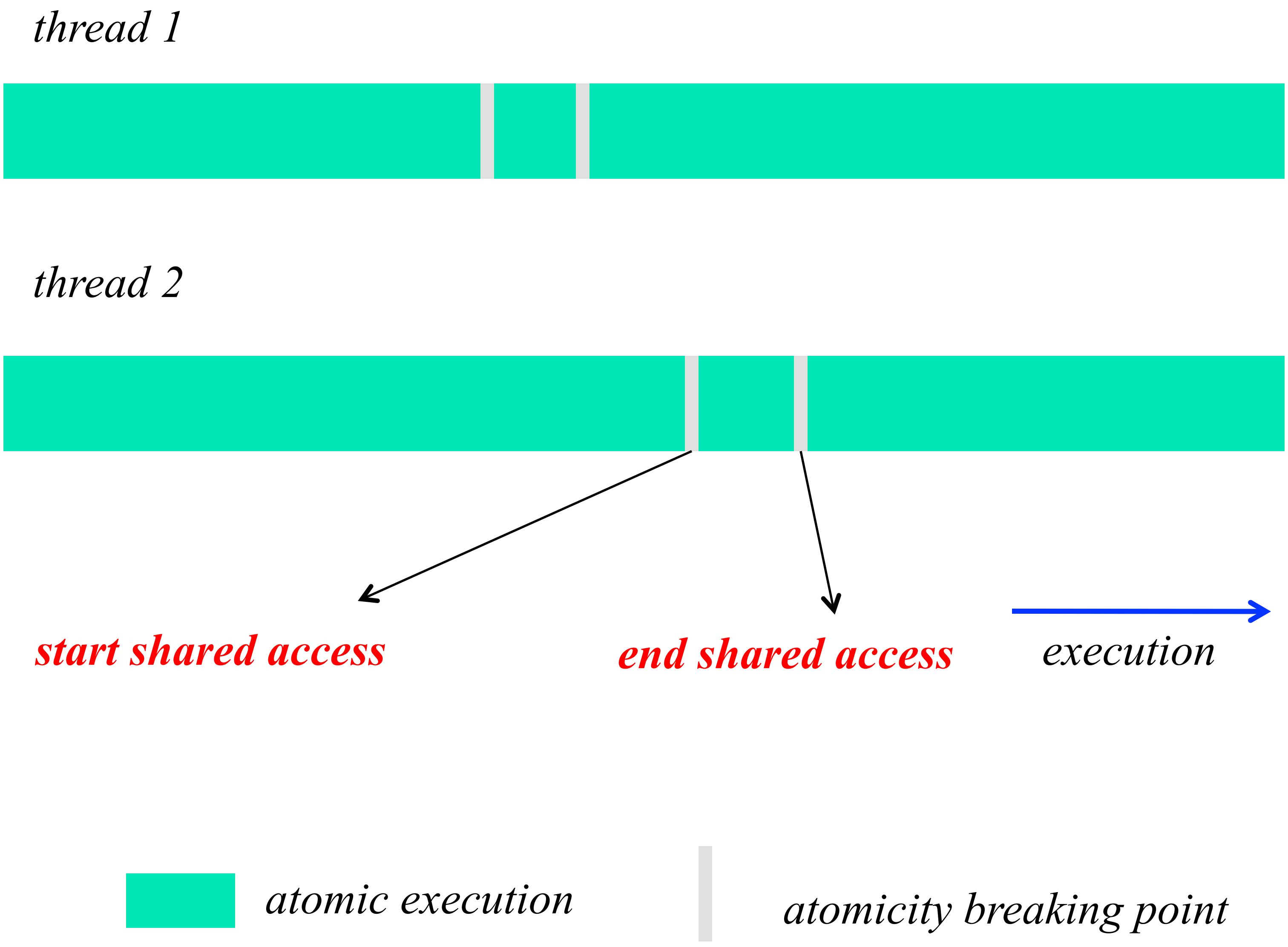
atomic execution

atomicity breaking point

You Ask…

- Question 1: wouldn't threads accessing "exclusive resources" end up waiting each other for a long time? (or rolling back a lot)
- Question 2: : familiar Java-like communication/ sharing patterns, such as rendezvous?
- Question 3: "pervasive atomicity"??? You mean "pervasive run-time locks/transactions?

Shared Access as Atomicity Break Points



You Ask…

- Question 1: wouldn't threads accessing "exclusive resources" end up waiting each other for a long time (or rolling back a lot)?
- Question 2: : familiar Java-like communication/ sharing patterns, such as rendezvous?
- Question 3: "pervasive atomicity"??? You mean "pervasive run-time locks/transactions?

***Task Types**, a type system that overlays a non-shared-memory-by-default model on top of the Java-like shared memory*

The Language Design

This Talk

A Java-like programming language, Coqa (first appeared in CC'08), with

- pervasive atomicity:
  - Benefits: the scenarios of interleaving are significantly reduced by language design, hence promoting better programming understanding and easier bug detection
- sharing-aware programming
- a static type system to enforce non-shared-memory-by-default

Example I

```
class Cheese {
     int c;
     void move() { c--; }
}
task class Person {
     void eat () {
        (new Cheese()).move();
     }
}
```

```
class Main {
   void main() {

     (new Person())->eat();
     (new Person())->eat();
   }
}
```

Example I

```
class Cheese {
    int c;
    void move() { c--; }
}
task class Person {
    void eat () {
        (new Cheese()).move();
    }
}
```

```
class Main {
    void main() {

        (new Person())->eat();
        (new Person())->eat();
    }
}
```

A "task" is a logical thread preserving pervasive atomicity

(created by sending a -> message to a "task object")

Example I

```
class Cheese {
     int c;
     void move() { c--; }
}
task class Person {
     void eat () {
        (new Cheese()).move();
     }
}
```

```
class Main {
   void main() {

     (new Person())->eat();
     (new Person())->eat();
   }
}
```

The inversion of Java's default – all classes without any modifiers are statically enforced task-local objects ("ordinary objects")

The two "eat" tasks are atomic: no surprise such as "Who moved my Cheese?"

Benefits of Static Isolation

- Access to them does not break atomicity
- Access to them does not need runtime protection
- Static semantics gives programmers confidence that pervasive atomicity does not translate to pervasive runtime overhead

Types of Coqa Objects

	default	*"shared"*
task units ("accessor")	*task objects*	*shared task objects*
data ("accessee")	*statically isolated objects*	*dynamically isolated objects*

Task Types

- Task Types: static locality/non-shared-memory enforcement for ordinary objects
- Can be viewed as a region/ownership type system where ordinary objects are assigned to regions – the static representation of tasks – but with unique challenges

Parametric Polymorphic Locality Typing

```
class Cheese {
     int c;
     void move() { c--; }
}
task class Person {
     void eat () {
        (new Cheese()).move();
     }
}
```

```
class Main {
   void main() {

     (new Person())->eat();
     (new Person())->eat();
   }
}
```

type variable t1

type variable t2

type variable t3 (for call site at t1)

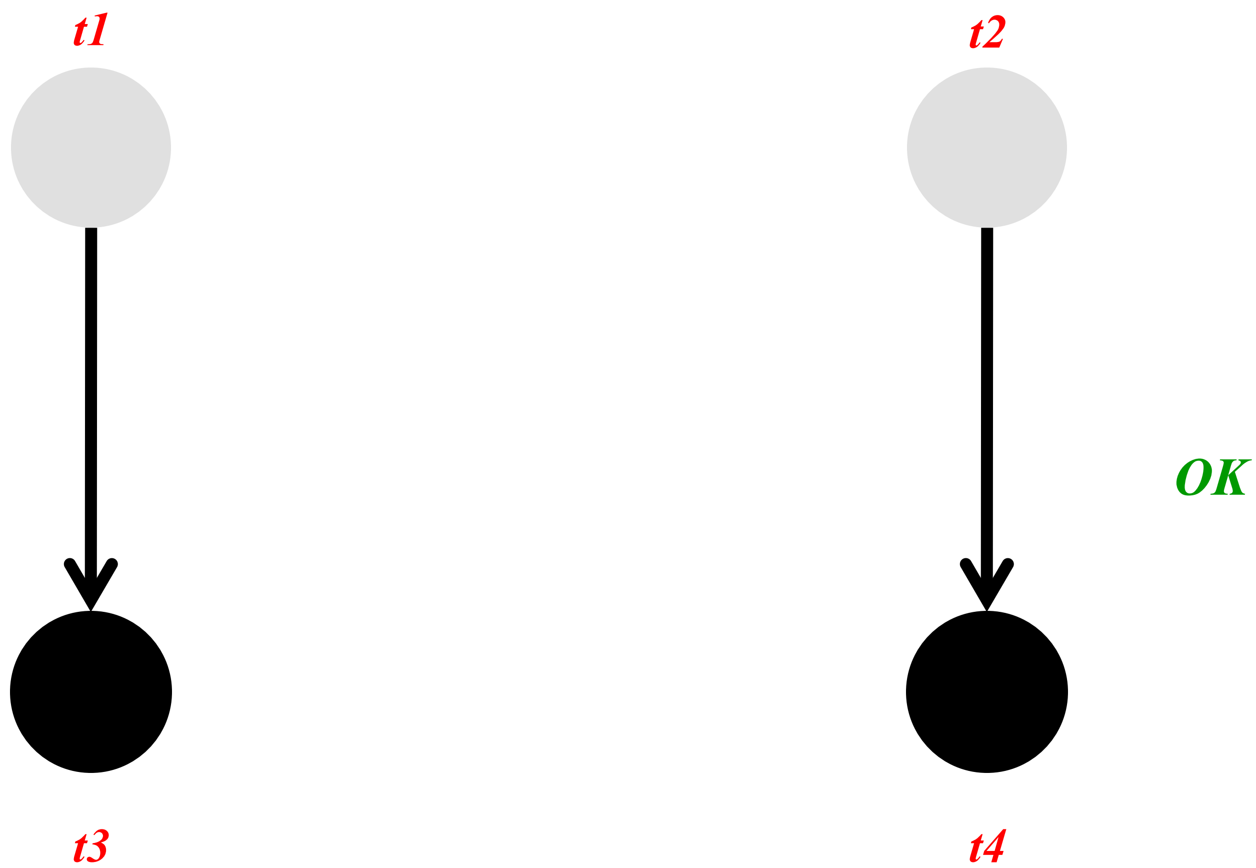
type variable t4 (for call site at t2)

t1 accesses t3

t2 accesses t4

parametric polymorphic type inference / context sensitivity

(Oversimplified) Static Access Graph



every type variable for ordinary objects
has to commit to one region/owner

Example II

```
class Cheese {
     int c;
     void move() { c--; }
}
task class Person {
     void eat (Cheese c) {
        c.move();
     }
}
```

```
class Main {
   void main() {
     Cheese c = new Cheese();
     (new Person())->eat(c);
     (new Person())->eat(c);
   }
}
```

type variable t1

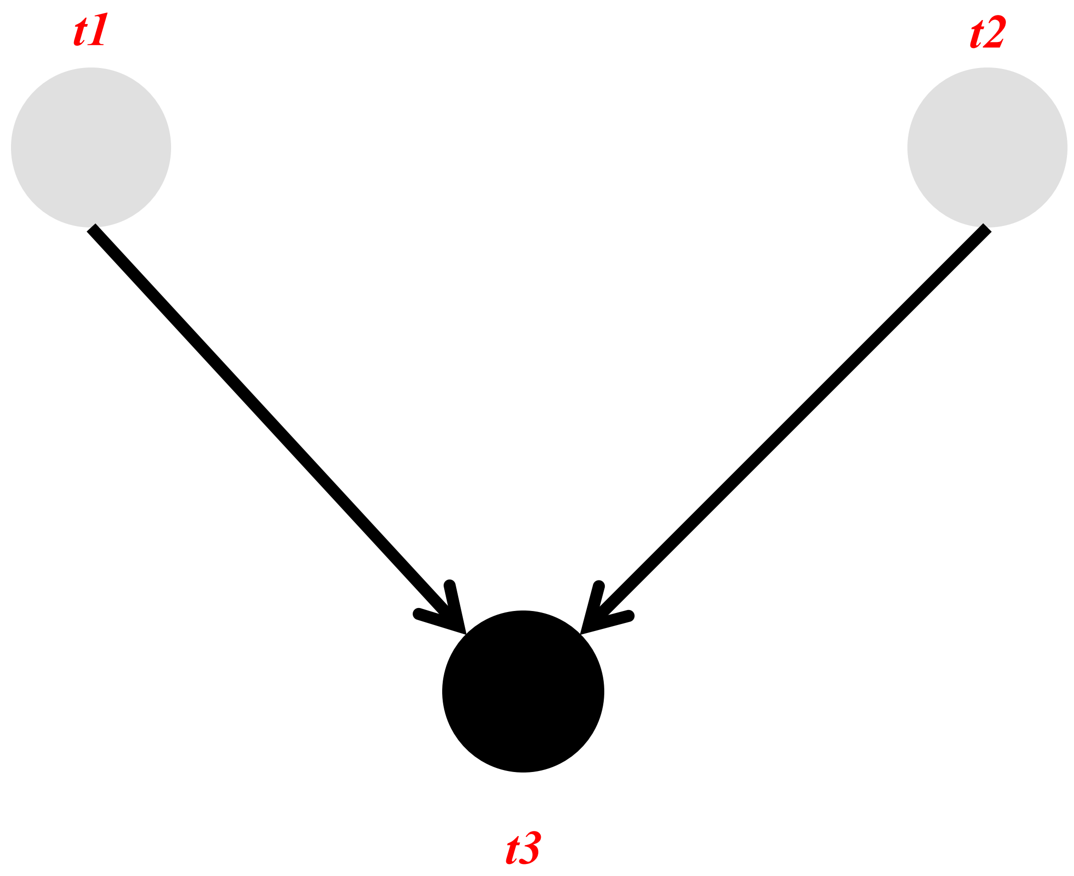
type variable t2

type variable t3

t1 accesses t3

t2 accesses t3

(Oversimplified) Static Access Graph



Rejected

Design Challenges of Task Types

- Full inference - no need to declare region-type-like parameterized classes and parametric types
- The number of regions (tasks) cannot be bound statically
- Complexity in the presence of explicit sharing

Design Challenges of Task Types

- Full inference – no need to declare region-type-like parameterized classes and parametric types
- The number of regions (tasks) cannot be bound statically
- Complexity in the presence of explicit sharing

Task Types in this light are a polymorphic region inference algorithm with instantiation-site polymorphism and method invocation-site polymorphism

Design Challenges of Task Types

- Full inference – no need to declare region-type-like parameterized classes and parametric types
- The number of regions (tasks) cannot be bound statically
- Complexity in the presence of explicit sharing

preserving soundness is not a trivial issue in presence of recursion:

can't directly borrow from region/ownership types

Example III

```
class Cheese {
     int c;
     void move() { c--; }
}
task class Person {
     void eat (Cheese c) {
        c.move();
     }
}
```

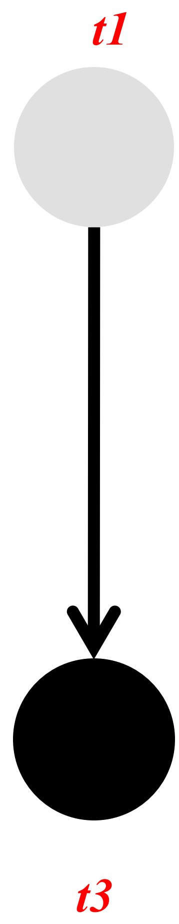
type variable t1

```
class Main {
   void main() {
     Cheese c = new Cheese();
     loop {
         (new Person())->eat(c);
     }
   }
}
```

type variable t3

t1 accesses t3

(Oversimplified) Static Access Graph



OK?

Task Twinning

- For each instantiation site of task objects, create a pair of type variables
  - Goal: to mimic the loss of information in (potentially) recursive contexts

Previous Example III

```
class Cheese {
     int c;
     void move() { c--; }
}
task class Person {
     void eat (Cheese c) {
        c.move();
     }
}
```

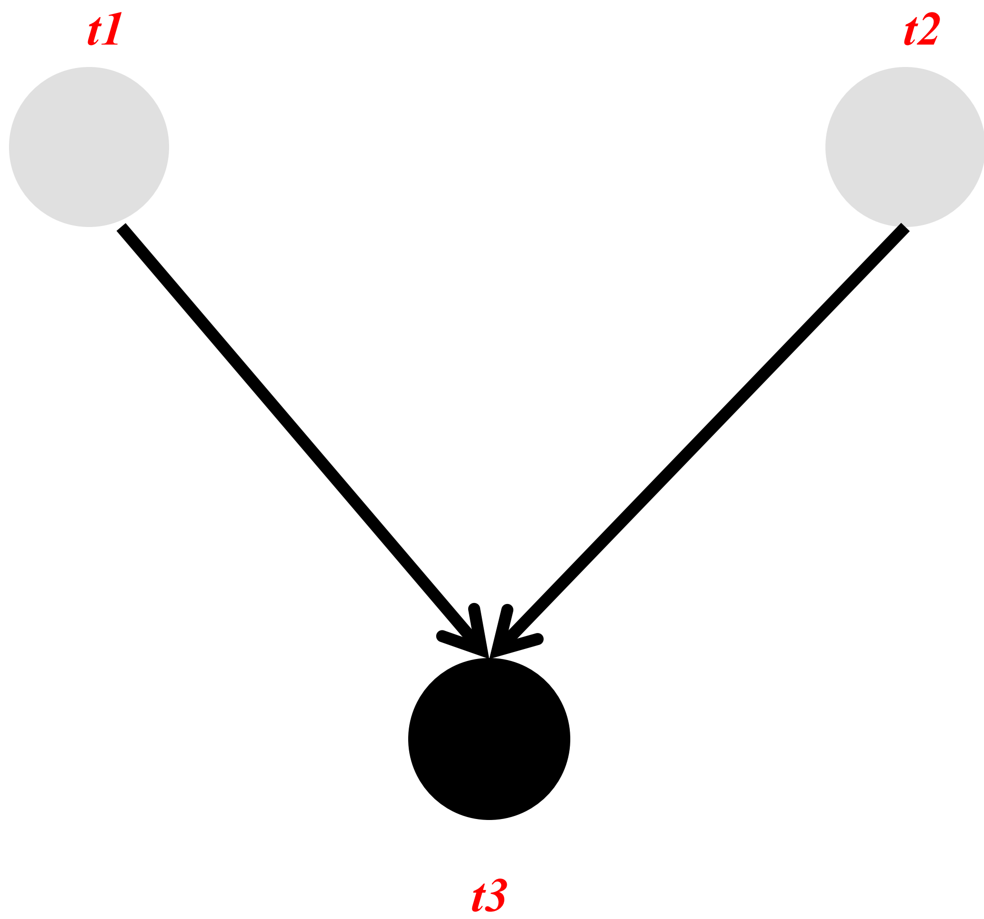
t1 accesses t3

```
class Main {
   void main() {
     Cheese c = new Cheese();
     loop {
         (new Person())->eat(c);
     }
   }
}
```

type variables t1, t2

type variable t3

Static Access Graph for Ex. III



Rejected

Previous Example II

```
class Cheese {
      int c;
      void move() { c--; }
}
task class Person {
      void eat (Cheese c) {
         c.move();
      }
}
```

```
class Main {
   void main() {
     Cheese c = new Cheese();
     (new Person())->eat(c);
     (new Person())->eat(c);
   }
}
```

type variables t1, t1'

type variable t2, t2'

type variable t3

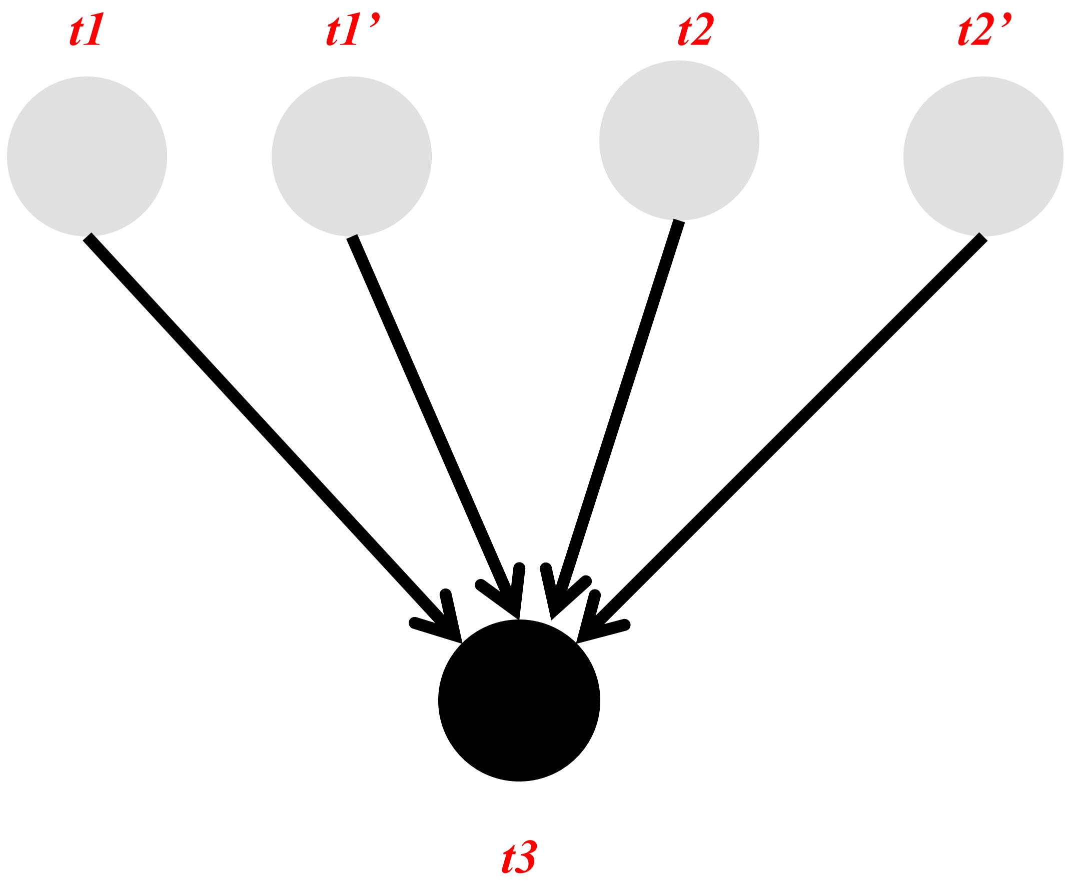
t1 accesses t3

t1' accesses t3

t2 accesses t3

t2' accesses t3

Static Access Graph for Ex. II



Rejected

Design Issues for Task Twinning

- Why two are enough?
- Wouldn't twinning make every program fail to typecheck?
- Optimizations?

Design Issues for Task Twinning

- Why two are enough?
- Wouldn't twinning make every program fail to typecheck?
- Optimizations?

Design Issues for Task Twinning

- Why two are enough?
- Wouldn't twinning make every program fail to typecheck?
- Optimizations?

a conflict (compile-time type error) only needs two accesses to form

Design Issues for Task Twinning

- Why two are enough?
- Wouldn't twinning make every program fail to typecheck?
- Optimizations?

Previous Example I

```
class Cheese {
     int c;
     void move() { c--; }
}
task class Person {
     void eat () {
        (new Cheese()).move();
     }
}
```

```
class Main {
   void main() {

     (new Person())->eat();
     (new Person())->eat();
   }
}
```

type variable t1, t1'

type variable t2', t2'

type variable t3 (for call site at t1)

type variable t3' (for call site at t1')

type variable t4 (for call site at t2)

type variable t4' (for call site at t2')

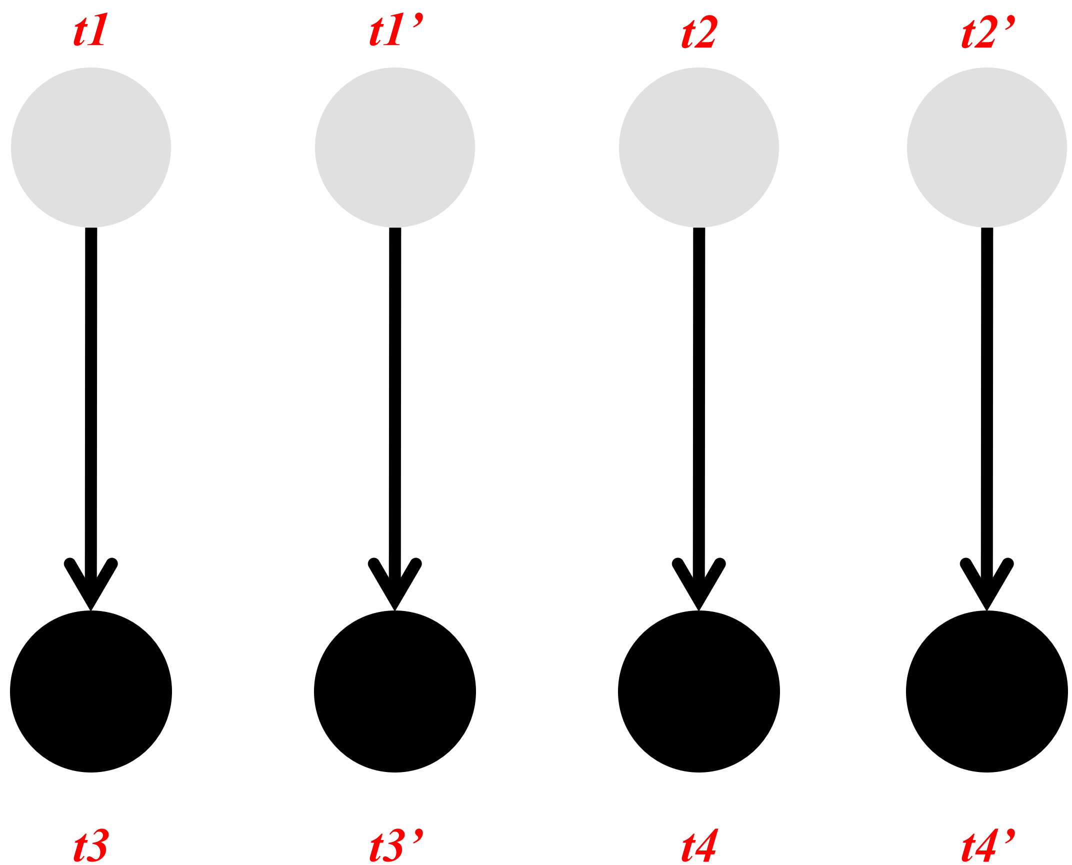
t1 accesses t3

t1' accesses t3'

t2 accesses t4

t2' accesses t4'

Static Access Graph for Ex. I



OK

Design Issues for Task Twinning

- Why two are enough?
- Wouldn't twinning make every program fail to typecheck?
- Optimizations?

1. differentiate read/write access

2. No twinning in non-recursive contexts

3. ...

Design Challenges of Task Types

- Full inference – no need to declare region-type-like parameterized classes and parametric types
- The number of regions (tasks) cannot be bound statically
- Complexity in the presence of explicit sharing

Sharing-Aware Programming

```
class Counter {
      int c;
      void inc() { c++; }
}
shared task class Library {
   Counter c = new Counter();
   void breturn() {
      c.inc();
   }
}
task class Student {
      void visit (Library l) {
         … /* do stuff 1 */
         l !->breturn();
         … /* do stuff 2 */
      }
}
```

```
class Main {
   void main() {
     Library l = new Library();
     (new Student())->visit(l);
     (new Student())->visit(l);
   }
}
```

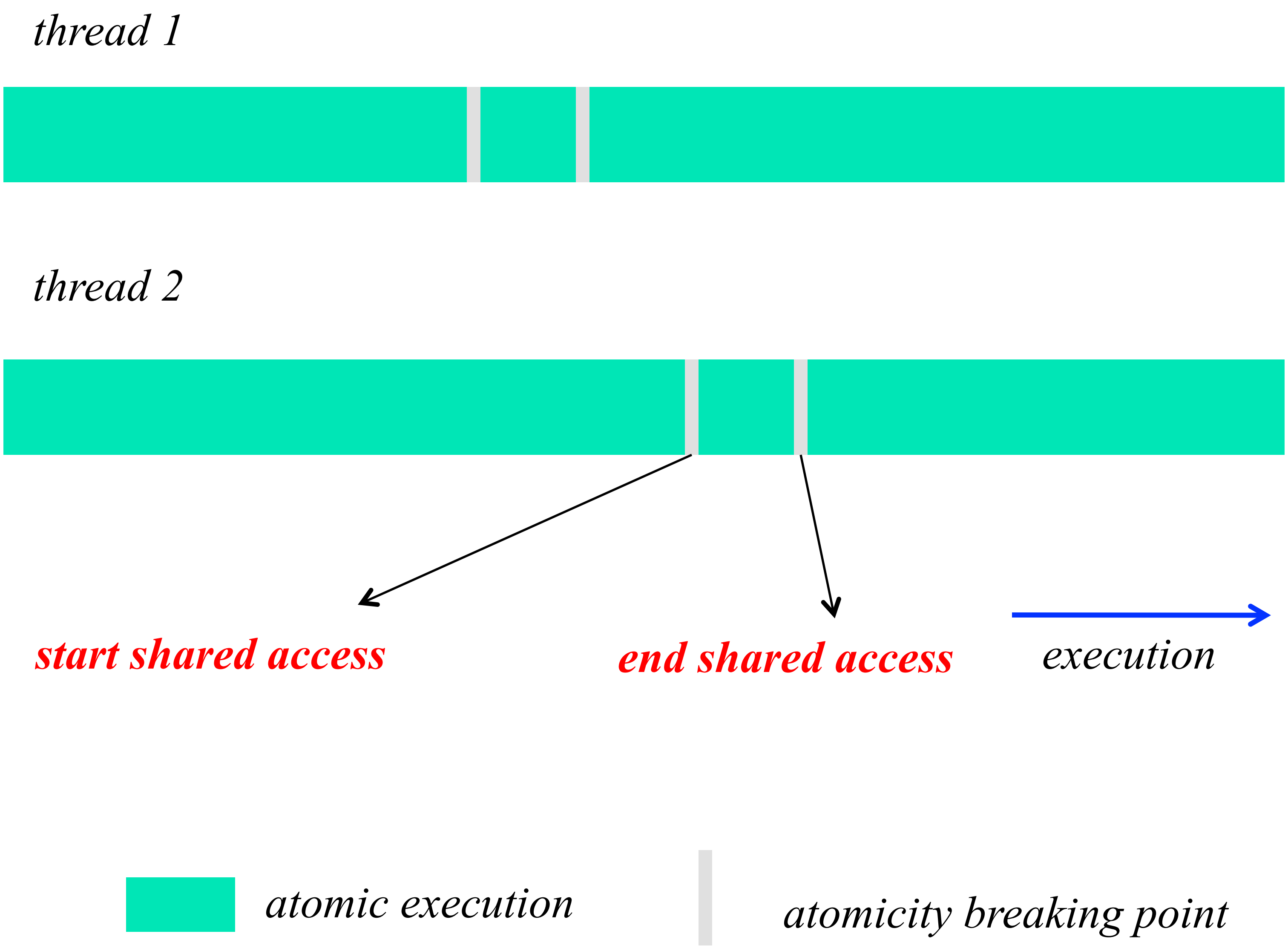
a shared resource wrapped up into an atomic execution of its own

(created by sending a !-> message to a "shared task object")

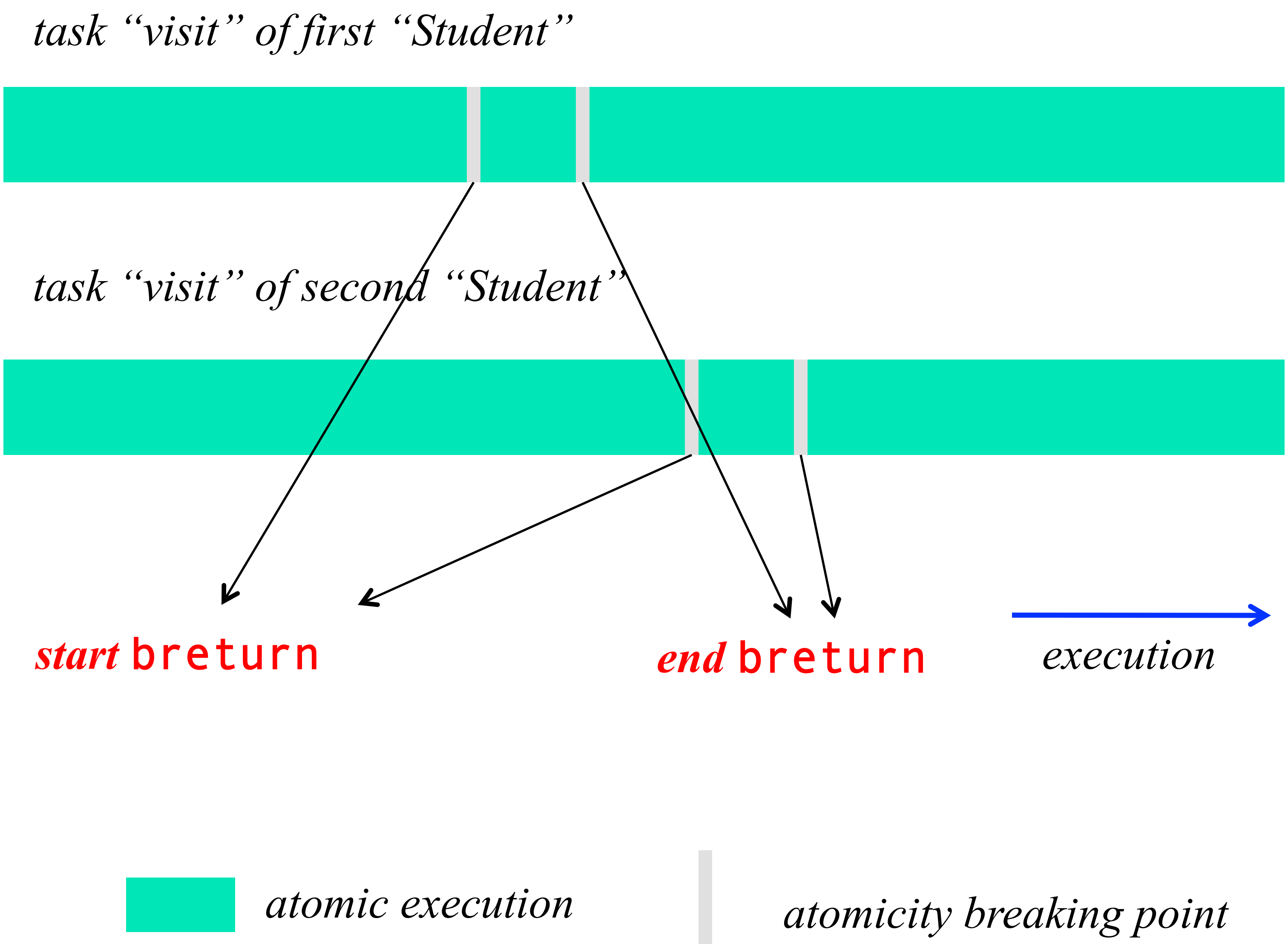
Types of Coqa Objects

	default	*"shared"*
task units ("accessor")	*task objects*	*shared task objects*
data ("accessee")	*statically isolated objects*	*dynamically isolated objects*

Shared Access as Atomicity Break Points



Shared Access as Atomicity Break Points



Shared Tasks

- A programmer's view:
  - an encapsulation of "a shared service" with independent lifecycle of evolution
  - the message sender object "gets the grip of its life" but still lets the world it interacts to evolve
- Designs:
  - Access dynamically protected: one message at a time
  - The sender de facto triggers a synchronous subroutine call

Types of Coqa Objects

+ *programmability*

+ *performance*

- *atomicity*

	default	*"shared"*
task units ("accessor")	*task objects*	*shared task objects*
data ("accessee")	*statically isolated objects*	*dynamically isolated objects*

Leftover Cheese as an Example

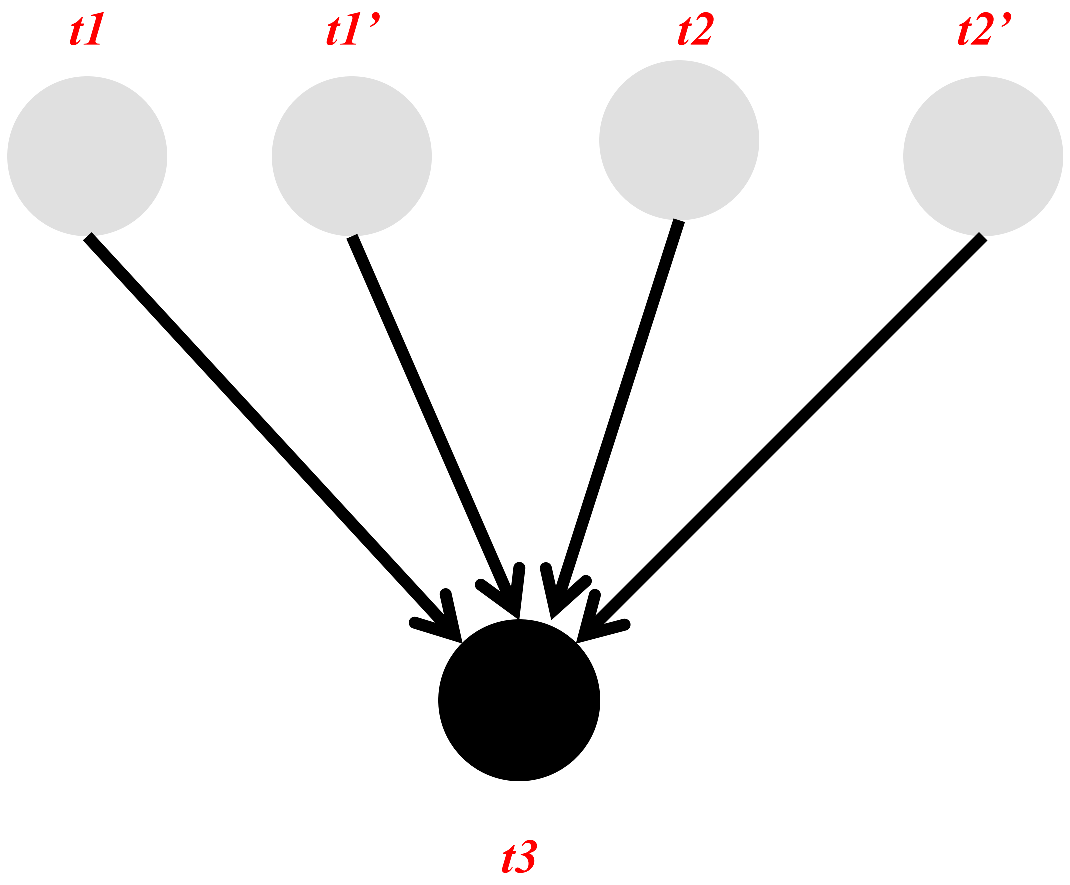
- Transfer as an example: one `Person` task object plans to eat the `Cheese` object, and then give the leftover to another `Person` task object to eat
  - Can't declare `Cheese` as a "shared task"
  - Can't declare `Cheese` as a default ordinary object

Example II

```
class Cheese {
     int c;
     void move() { c--; }
}
task class Person {
     void eat (Cheese c) {
        c.move();
     }
}
```

```
class Main {
   void main() {
     Cheese c = new Cheese();
     (new Person())->eat(c);
     (new Person())->eat(c);
   }
}
```

Static Access Graph for Ex. II



Rejected

Example II Modified

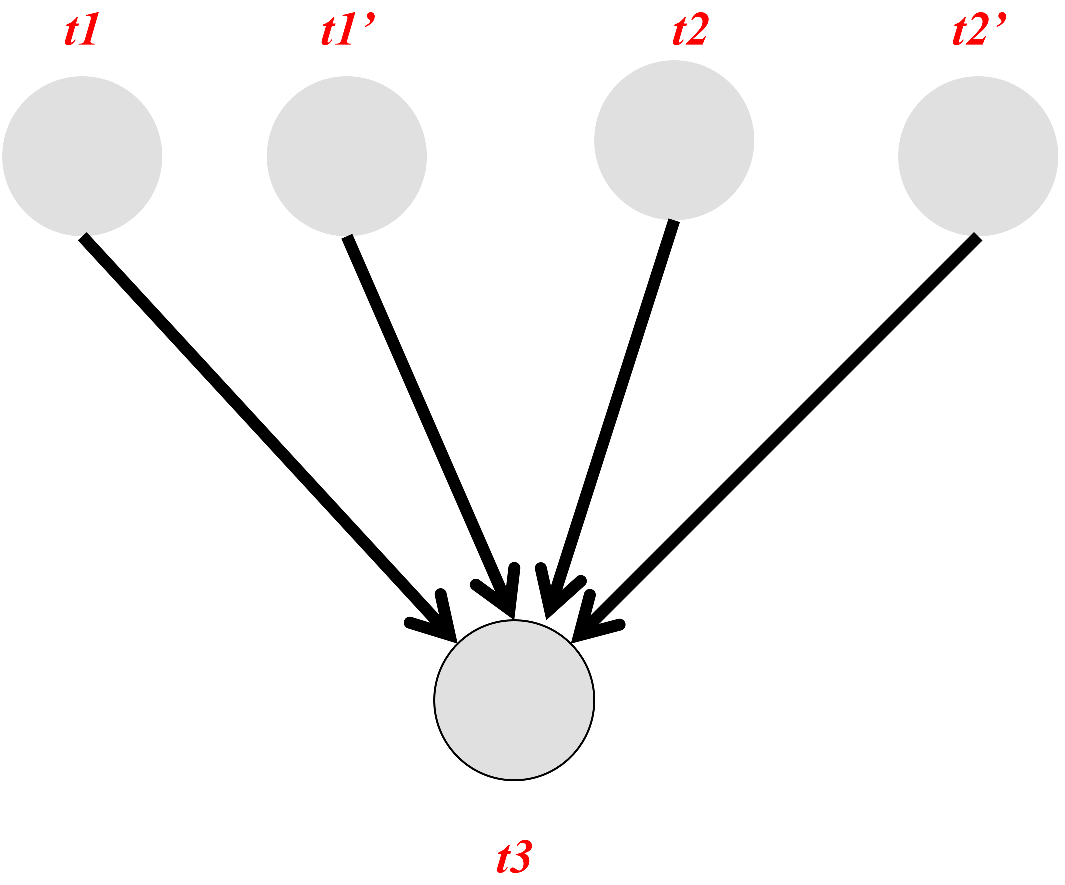
```
shared class Cheese {
     int c;
     void move() { c--; }
}
task class Person {
     void eat (Cheese c) {
        c!.move();
     }
}
```

```
class Main {
   void main() {
     Cheese c = new Cheese();
     (new Preson())->eat(c);
     (new Person())->eat(c);
   }
}
```

dynamic isolated objects
(created by sending a !. message
to a "shared ordinary object")

They are in fact good old Java objects

Static Access Graph Modified



OK

Types of Coqa Objects

	default	*“shared”*
task units (“accessor”)	*task objects*	*shared task objects*
data (“accessee”)	*statically isolated objects*	*dynamically isolated objects*

+ programmability

+ performance

- atomicity

+ programmability

- performance

+ atomicity

Dynamically Isolated Ordinary Objects

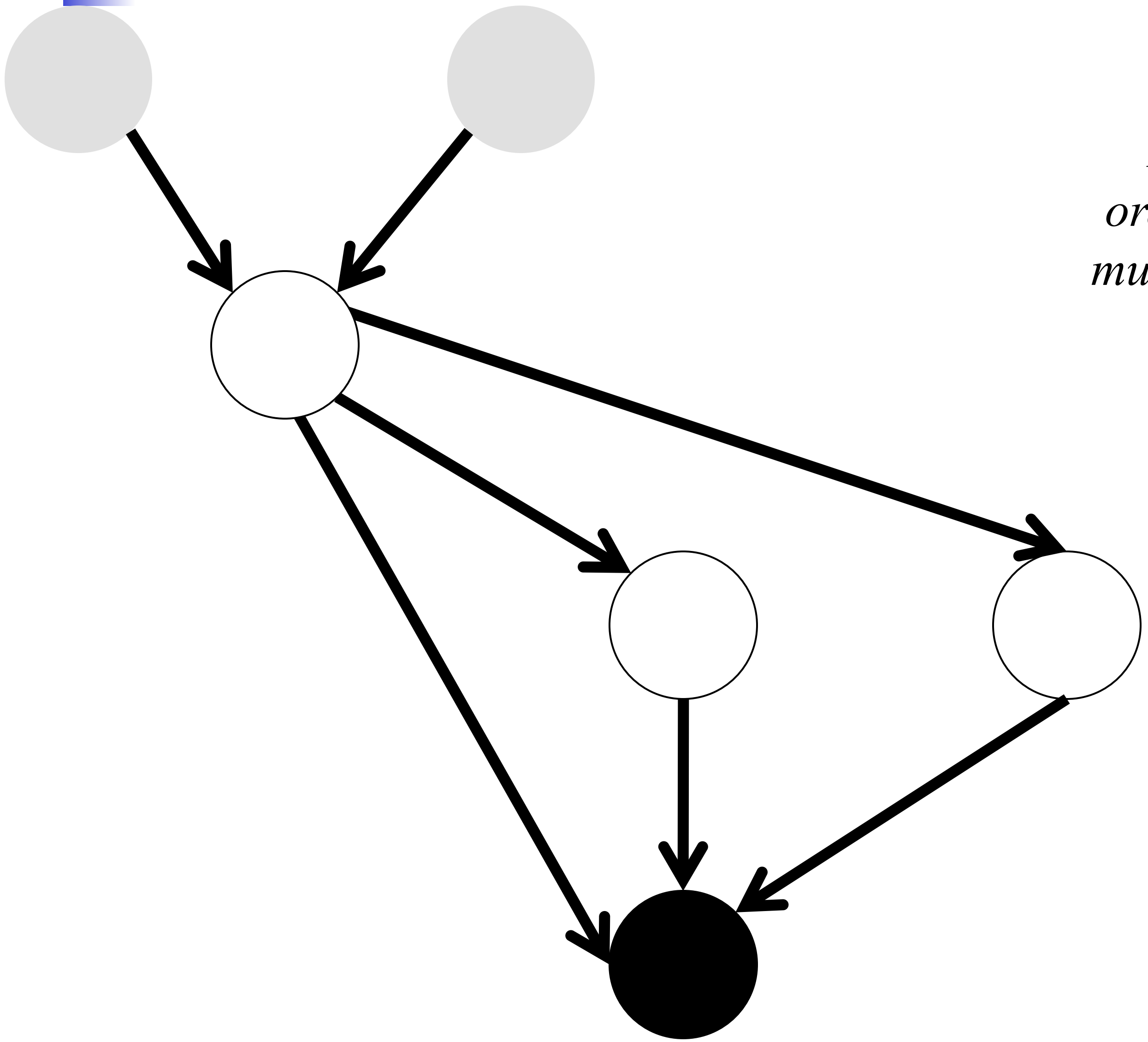
- Can be optimized to be statically protected in many cases, e.g. with flow-sensitive analyses, uniqueness, linear types, temporality enforcement
- Static approaches are always conservative: so there is a reason this style of objects stand as a separate category

Results

Meta-Theory

- Static Isolation
- Type soundness proved via subject reduction and progress
- No race conditions
- Pervasive atomicity enforcement

Static Isolation



For every default ordinary object, there must be a cut vertex on the graph

OK

Meta-Theory

- Static Isolation
- Type soundness proved via subject reduction and progress
- No race conditions
- Pervasive atomicity enforcement

Implementation

- Coqa with Task Types: implemented on top of Polyglot
  - Most Java features except native code and reflection
  - Lock-based semantics
  - Non-exclusive read lock and exclusive write locks
    - Subsumes "access to immutable objects does not lead to atomicity violation"
  - Deadlocks still possible
    - In a non-shared-memory-by-default model, deadlocks are relatively uncommon – no locks no deadlocks!

Initial Case Studies

- Benchmarks:
  - An “embarrassingly parallel” Raytracer
  - A contention-intensive Puzzlesolver
- Results:
  - programmability: the syntactical “diff” between Java and Coqa is minimal: only the new class modifiers and invocation symbols
  - Performance on a 24-core machine:
    - 15-35% faster than purely dynamically enforced atomicity
    - 5-35% slower than correctly synchronized but no atomicity Java

Some Related Work

- Actors, actor-like languages, actor-inspired languages
  - We (roughly) belong to this category, with a focus on minimal change of Java programmer habits, atomicity, and static isolation
- Language designs for atomicity, esp. AME, "yield" by Yi & Flanagan, data-centric atomicity
- Determinism by design
- Static thread locality: escape analysis, type-based isolation
- Talks in this session!

Concluding Remarks

- Pervasive atomicity addresses the need of writing software on multi-core platforms, where interleavings are pervasive
- Enforcing pervasive atomicity with non-shared-memory-as-default achieves efficiency and better program understanding
- Non-shared-memory-by-default can be enforced by a static type system
- Sharing-aware programming helps retain coding habits familiar with Java programmers, with increased declarativity

Thank you!

Typing Shared Task Access

```
class Counter {
     int c;
     void inc() { c++; }
}
shared task class Library {
   Counter c = new Counter();
   void breturn() {
      c.inc();
   }
}
task class Student {
     void visit (Library l) {
        l !->breturn();
     }
}
```

type variable t4 for t1

type variable t4' for t1'

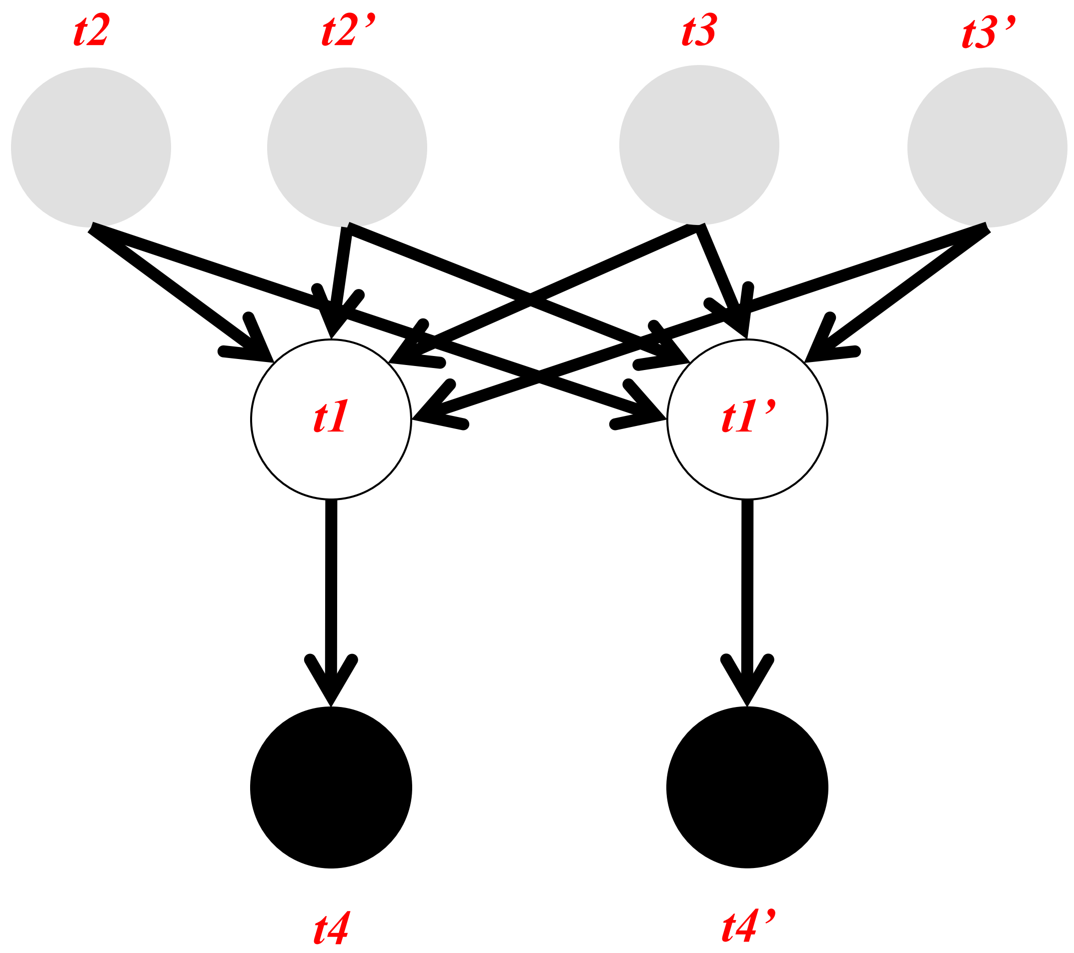
```
class Main {
   void main() {
     Library l = new Library();
     (new Student())->visit(l);
     (new Student())->visit(l);
   }
}
```

type variable t2, t2'

type variable t3, t3'

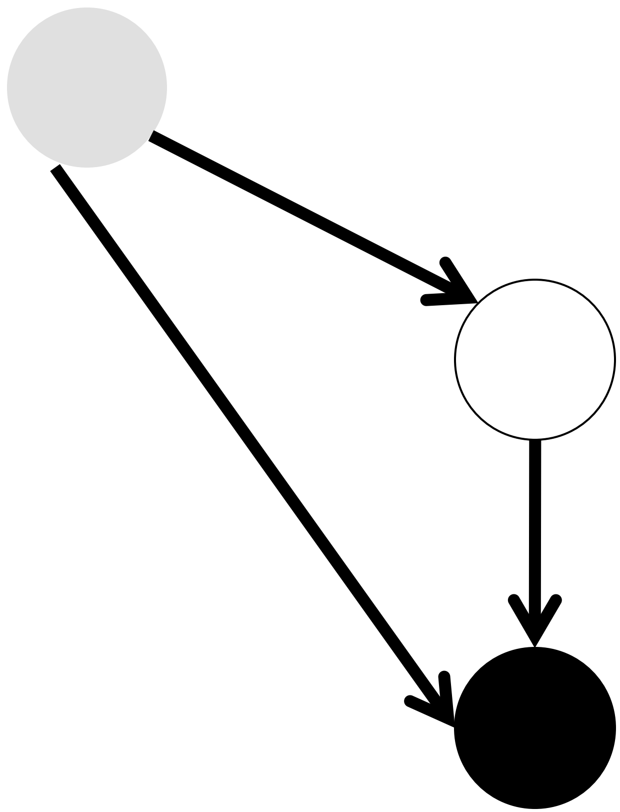
type variable t1, t1'

Static Access Graph



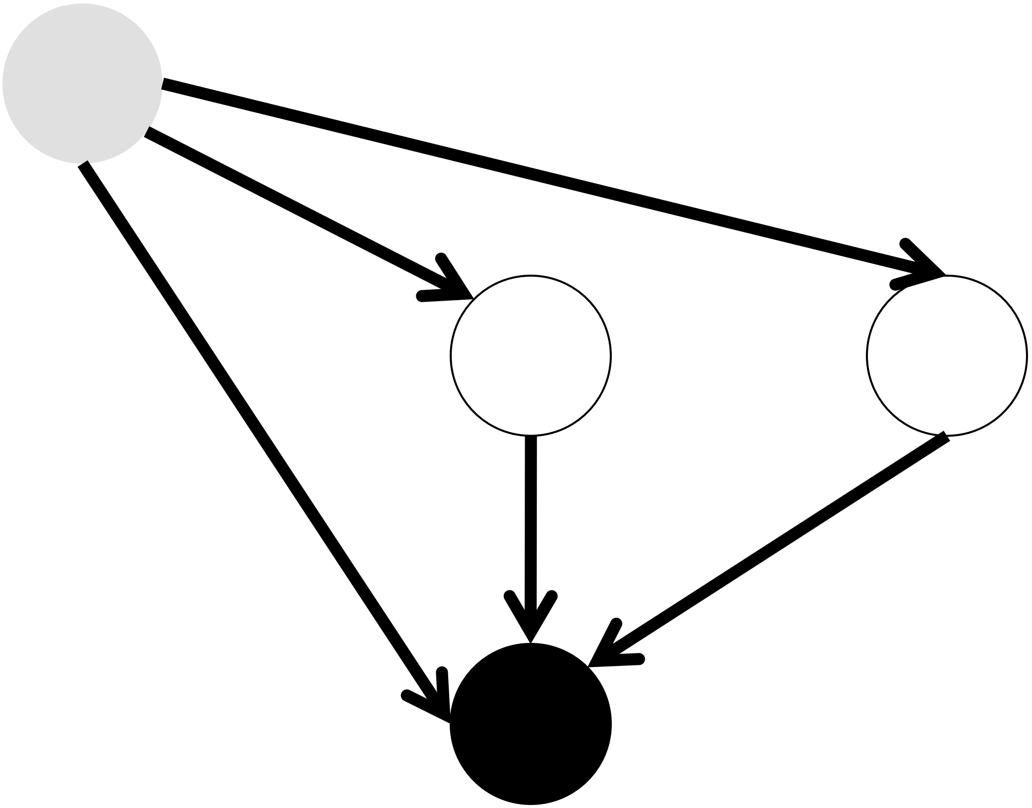
OK

More Access Graphs



OK!

More Access Graphs



OK!