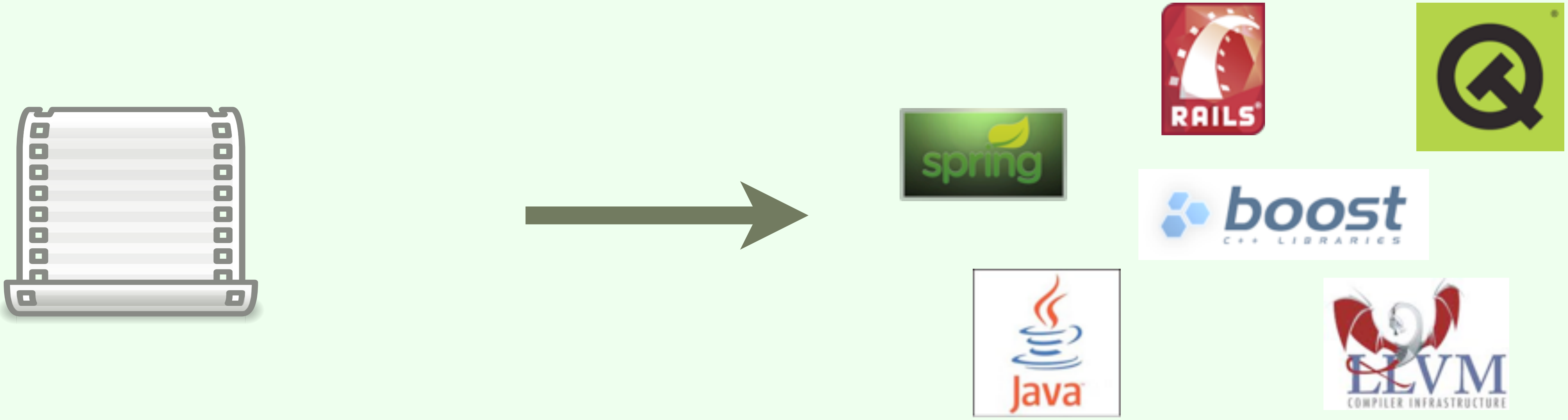
ÆMINIUM

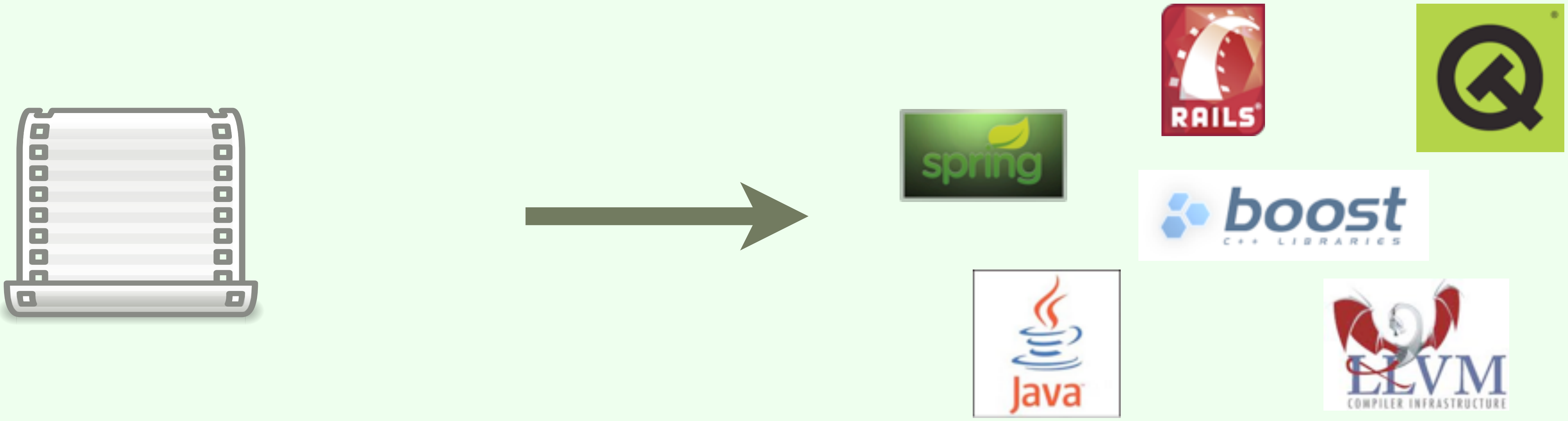
Freeing Programmers from the Shackles of Sequentiality

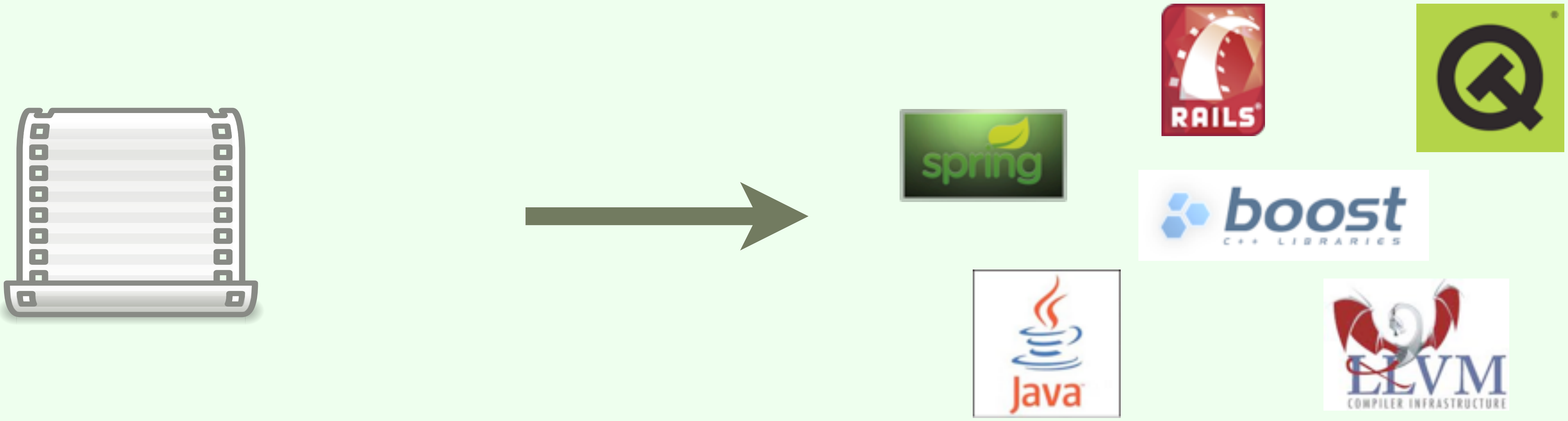
Thesis Proposal Talk
Sven Stork

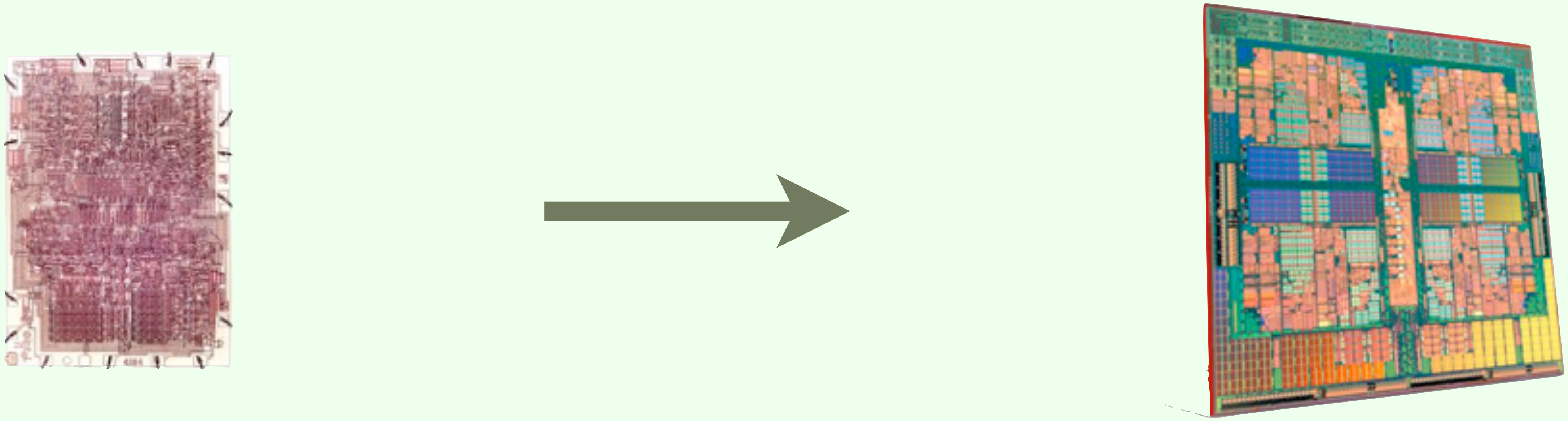
Committee

Jonathan Aldrich (CMU)	Paulo Marques (UC)
Todd Mowry (CMU)	Ernesto Costa (UC)
William Scherlis (CMU)	Marco Viera (UC)

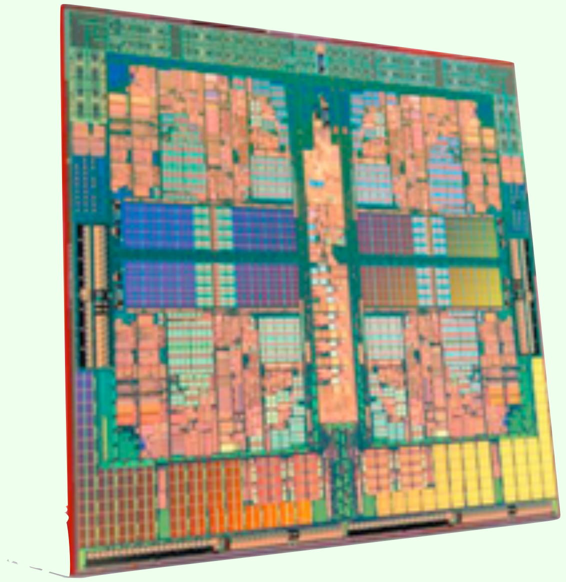

spring
RAILS
boost
C++ LIBRARIES
Java
LLVM
COMPILER INFRASTRUCTURE


spring
RAILS
boost
C++ LIBRARIES
Java
LLVM
COMPILER INFRASTRUCTURE


RAILS
spring
boost
C++ LIBRARIES
Java
LLVM
COMPILER INFRASTRUCTURE



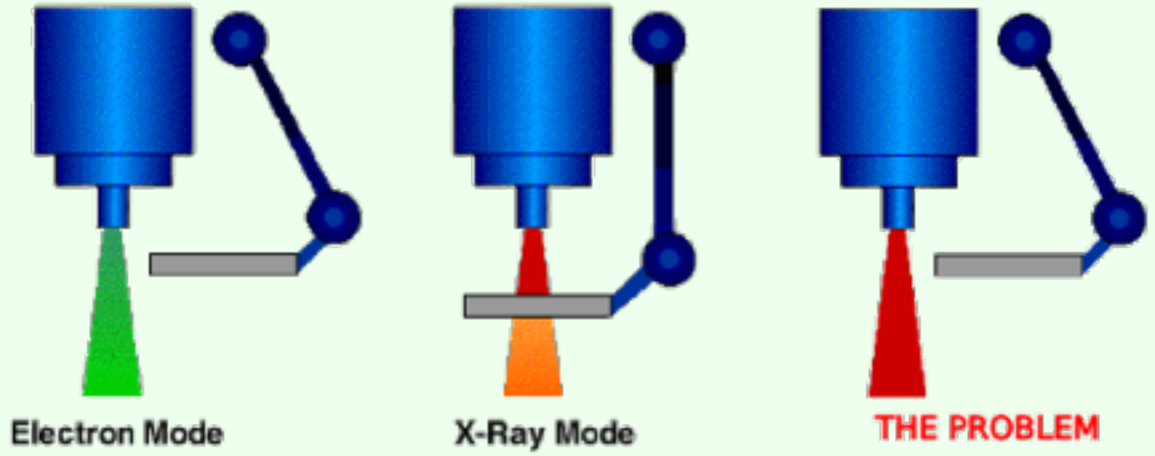
How to write and use frameworks and libraries correctly?



How to write correct parallel/concurrent code?

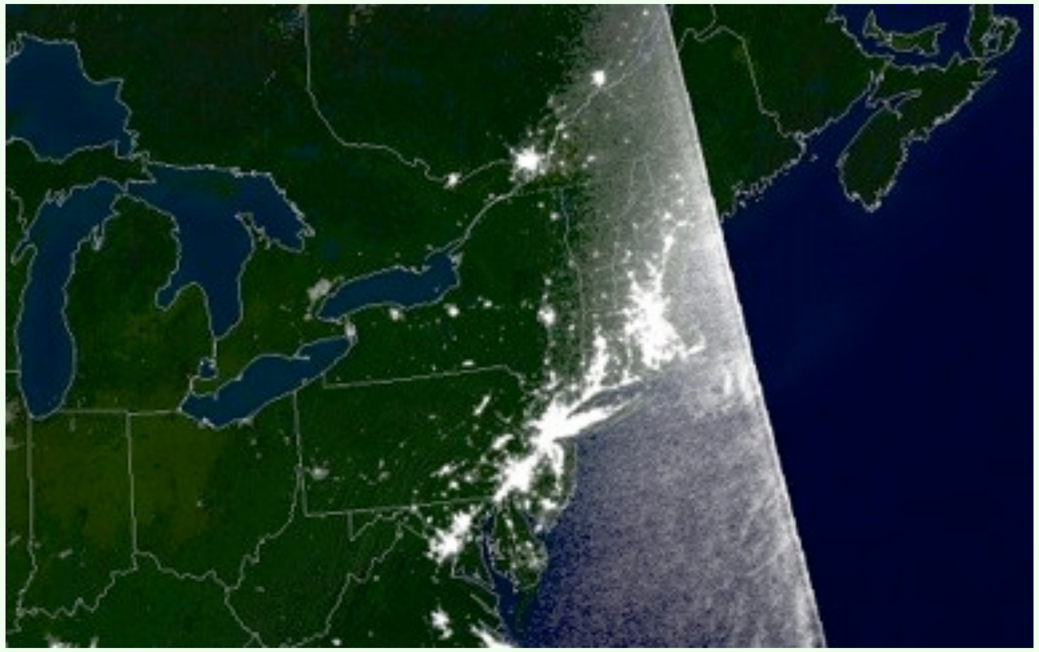
Why correctness matters?

- Therac-25
- race condition
- 3 deaths
- 3 heavy injuries



Why correctness matters?

- Blackout (2003)
- race condition
- 55,000,000 people affected



How to solve these problems?

Step by step

- Kevin Bierhoff check correct object usage:
  - **type state** to check **object protocols**
  - **access permissions** to tackle **aliasing**
- Plural [sequential protocols]

Step by step

- Nels Beckman extend Bierhoff's work to verify object protocols in concurrent settings
    - **access permission** to check correct **synchronization**
    - **access permissions** to **optimize** STM
- NIMBY/Sync' or Swim [concurrent protocols]

Step by step

- So far we can check that programs
  - obey object protocols
  - are properly synchronized
- How to write concurrent programs in first place?

How to write concurrent programs?

- Experiment
  - Implemented a few programs in various parallel programming abstractions
- Observation
  - no silver bullet
  - implicit parallelism appeared better
  - no solutions for future

Pushing the Envelope

- How should we write parallel code in 20-30 years?

Pushing the Envelope

- How should we write parallel code in 20-30 years?

> Don’t do it!
>
> -- Doug Lea

Pushing the Envelope

- make experiment
- ÆMINIUM ⟷ parallelism
  garbage collector ⟷ memory management
- automatically parallelization of code
  - composable
  - modular

Thesis Statement

The flow of access- and group-permissions provides a powerful abstraction to capture common programming idioms while simultaneous enabling the safe extraction of efficient concurrency.

In other words ...

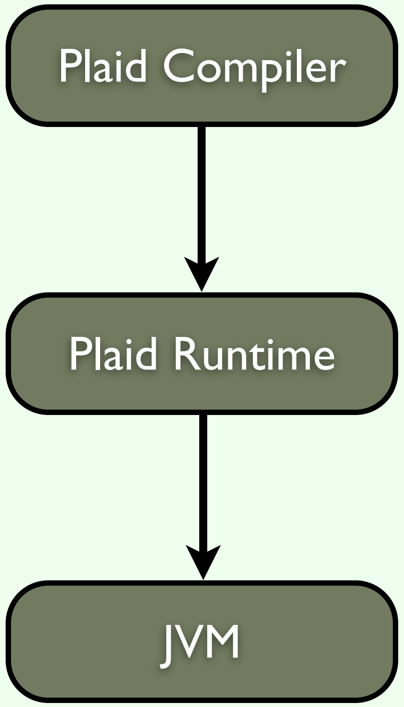
- propose abstract concept (ÆMINIUM)
- use permission information for automatic parallelization of programs
- permissions are suitable abstraction
  - can express common concurrent programming patters
  - allow us to achieve better performance

Hypotheses

- The ÆMINIUM approach is
  - **save** (i.e., no data races)
  - **efficient** (i.e., achieve speedup)
  - **practical** (i.e., express common programming paradigms)

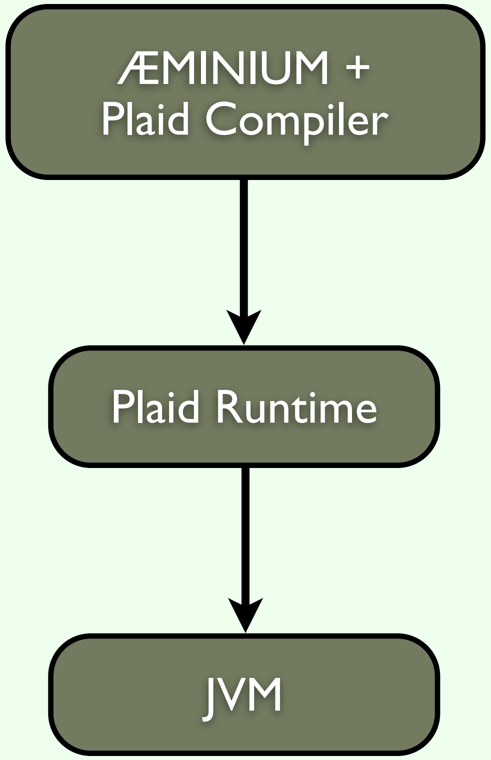
Approach

- **formalizing** and **implementation** of the ÆMINIUM approach



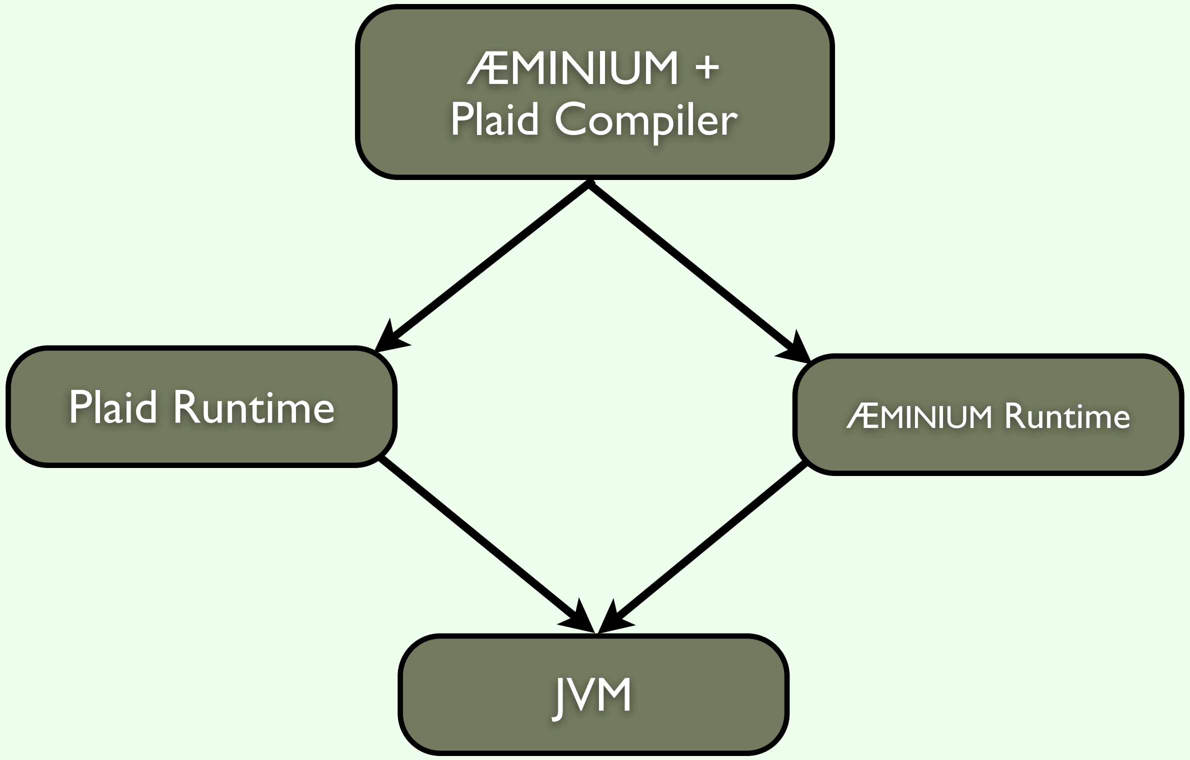
Approach

- formalizing and implementation of the ÆMINIUM approach



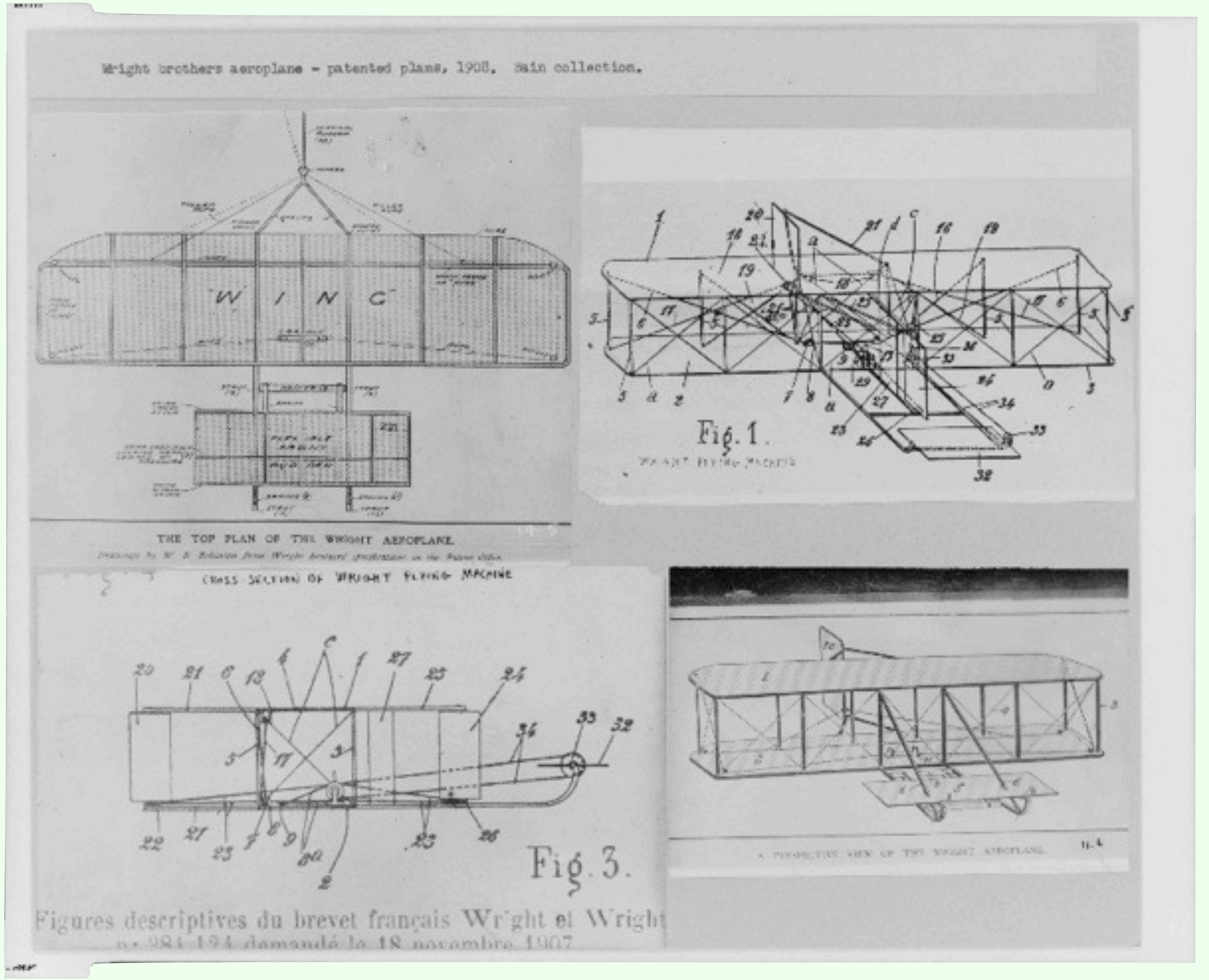
Approach

- formalizing and implementation of the ÆMINIUM approach



Contributions

- formal system of ÆMINIUM
- proof of concept implementation
- evaluation of feasibility


Wright brothers aeroplane - patented plans, 1908. Main collection.
W I N G
THE TOP PLAN OF THE WRIGHT AEROPLANE
Fig. 1.
WRIGHT FLYING MACHINE
CROSS SECTION OF WRIGHT FLYING MACHINE
Fig. 3.
Figures descriptives du brevet français Wright et Wright
n° 384 124 demandé le 18 novembre 1907
A PERSPECTIVE VIEW OF THE WRIGHT AEROPLANE

The Approach Explained

Access Permissions

- abstract capabilities associated with object references that encode
  - **access rights** (e.g., read/write)
  - **aliasing** information
- extensively used for verification (e.g. concurrency, protocols)

Access Permissions

Aliasing

Access		1	N
	RW	unique	shared
	R	immutable	immutable

Access Permissions

- **linear logic** (resource logic)

- **split** and **join**

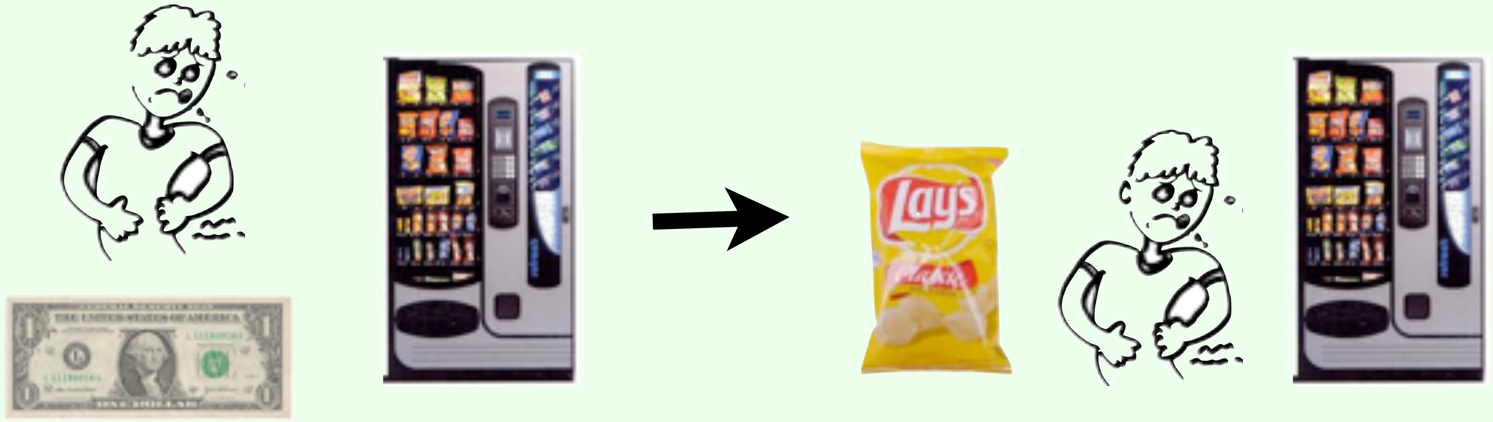
Access Permissions

- **linear logic** (resource logic)

- **split** and **join**

Access Permissions

- **linear logic** (resource logic)



- **split** and **join**

Access Permissions

- **linear logic** (resource logic)

- **split** and **join**

Access Permissions

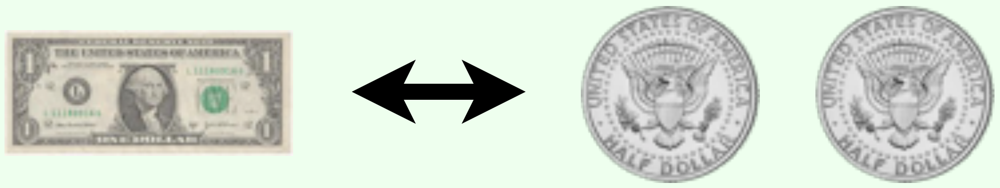
- **linear logic** (resource logic)

- **split** and **join**

Access Permissions

- **linear logic** (resource logic)

- **split** and **join**

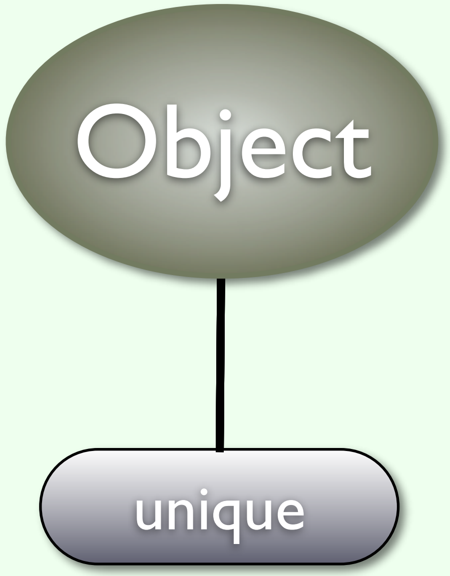


Access Permissions

- **linear logic** (resource logic)

- **split** and **join**

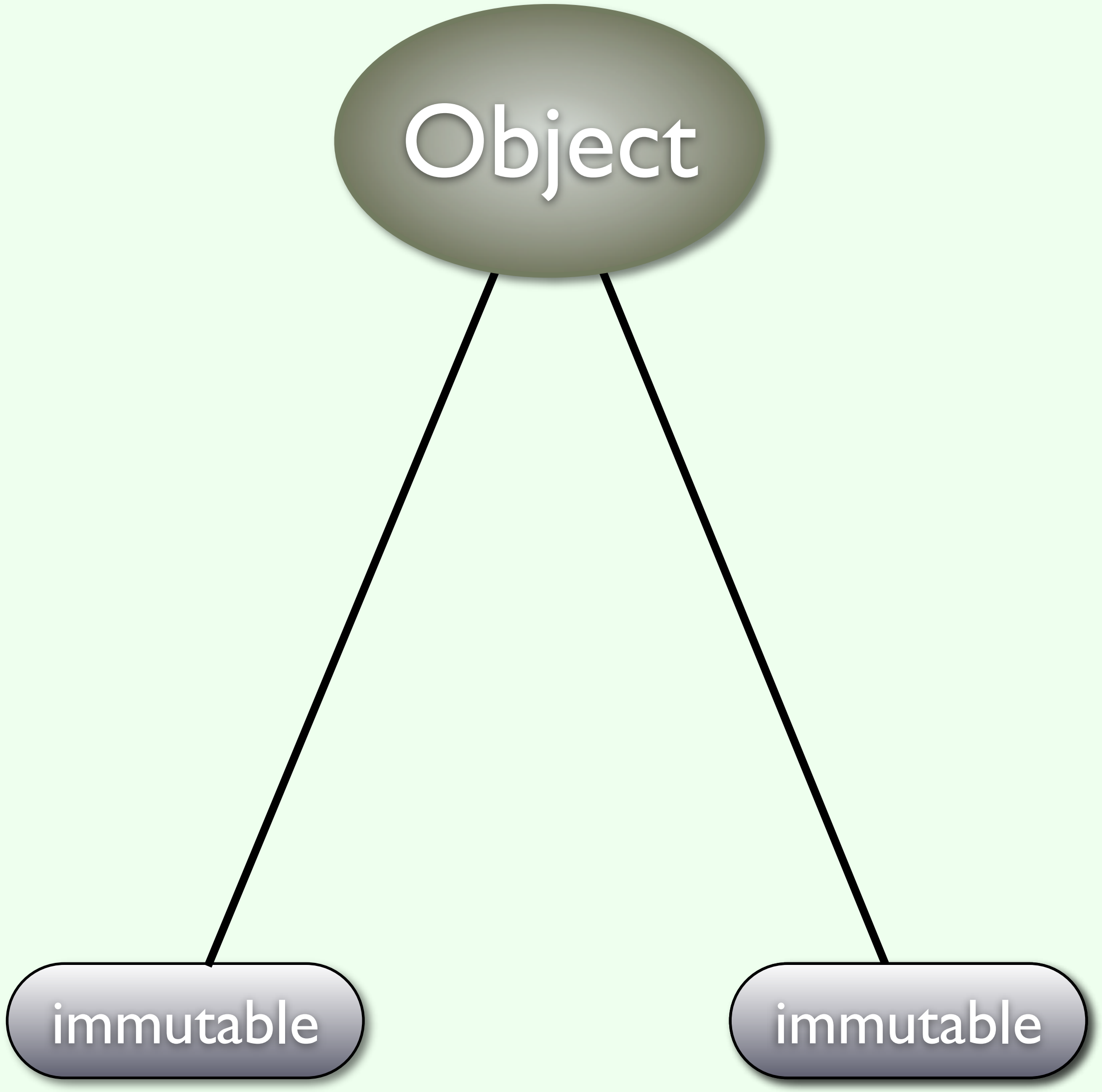
Access Permissions



Unique Permission

- aliases = 1
- access= RW
- “thread local”
- no synchronization

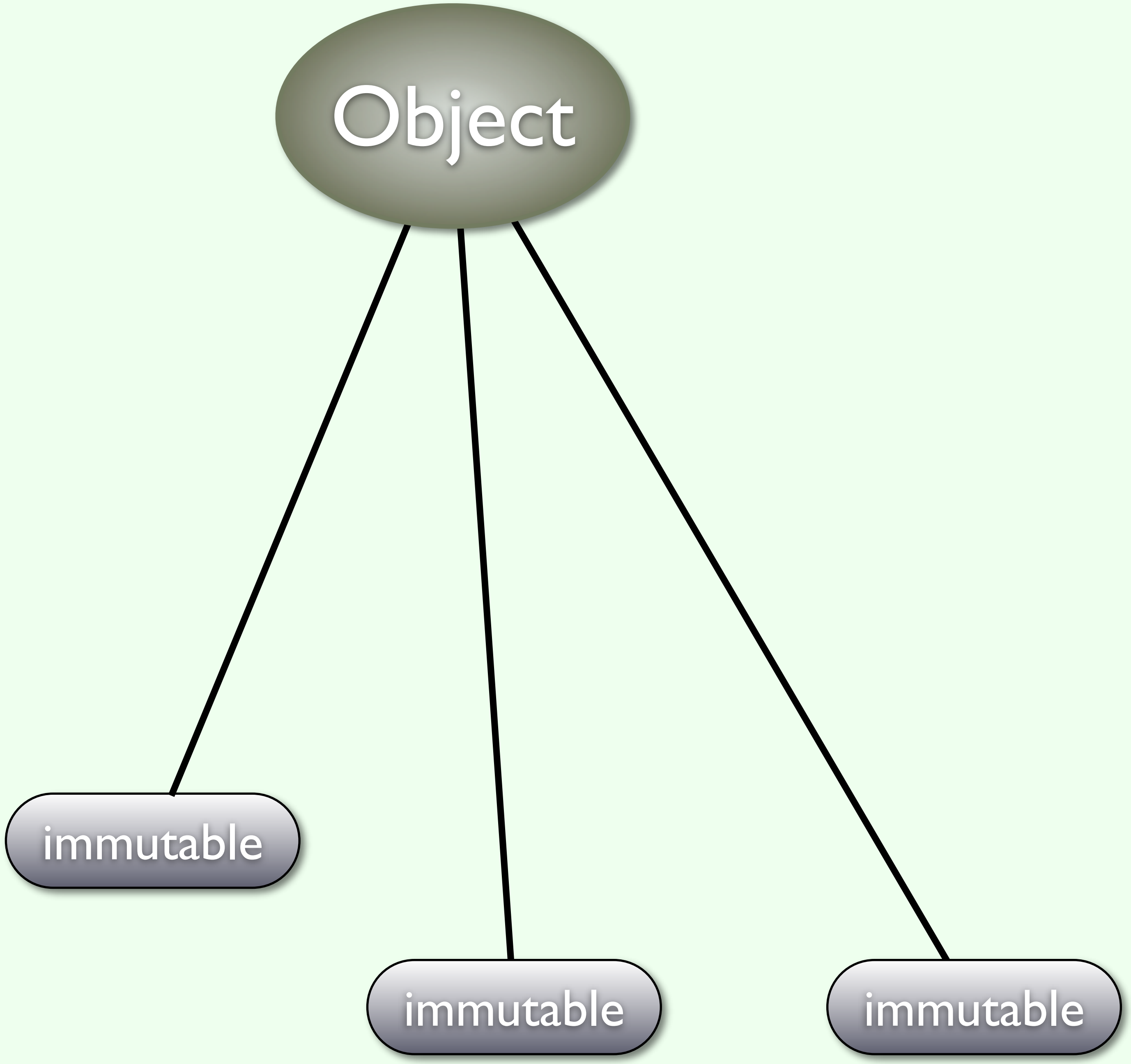
Access Permissions



Immutable Permission

- aliases = N
- access= R
- “constant”
- no synchronization

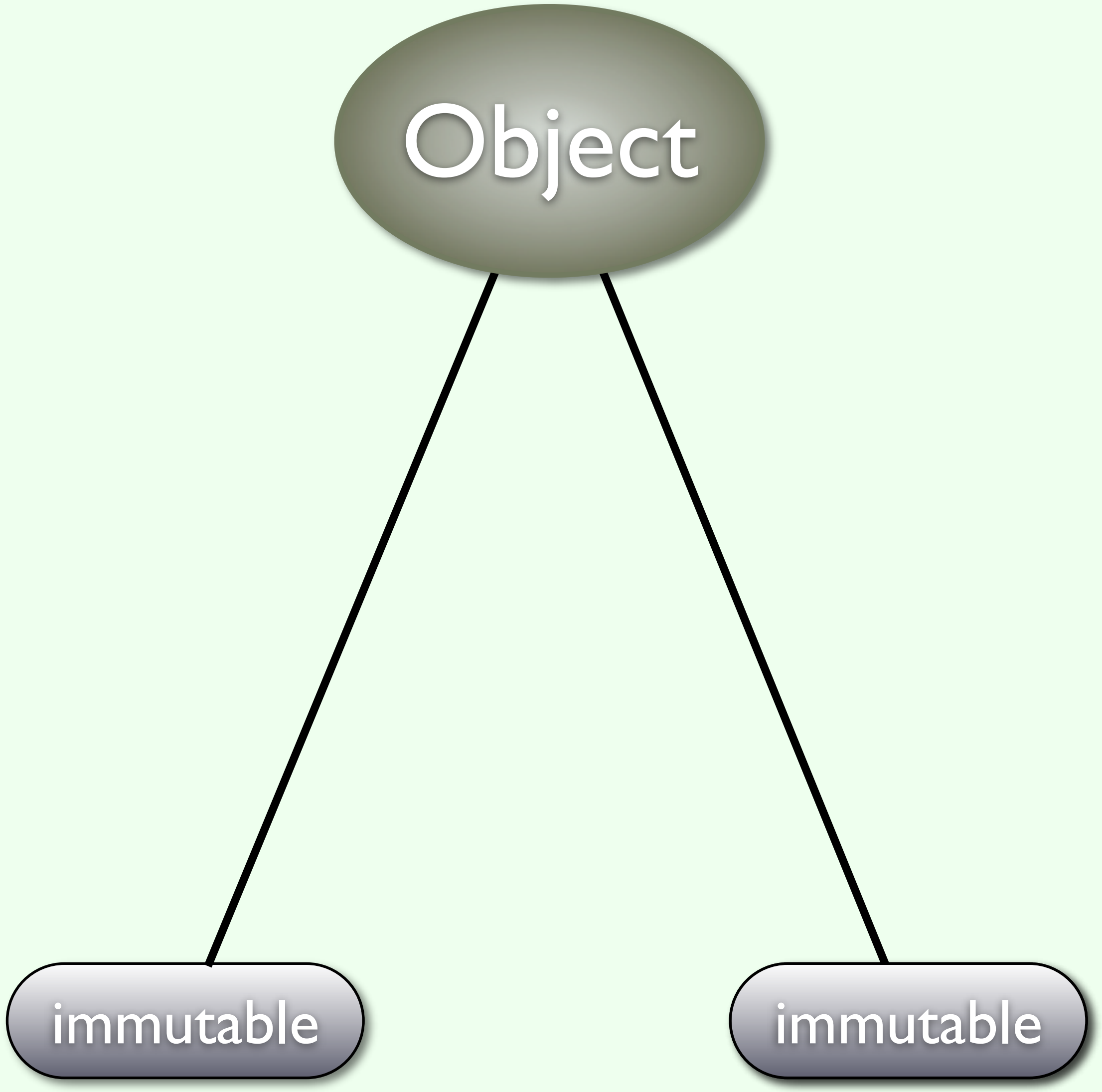
Access Permissions



Immutable Permission

- aliases = N
- access= R
- “constant”
- no synchronization

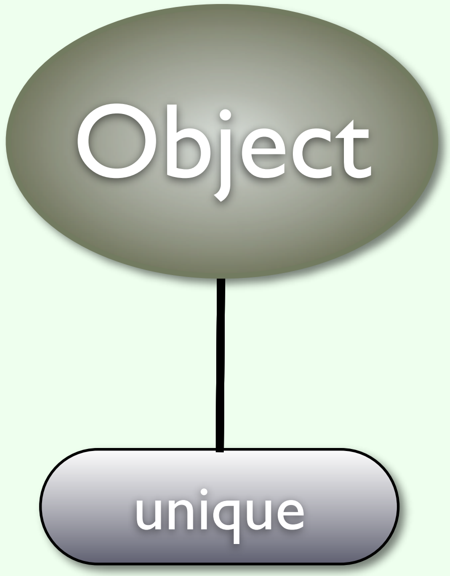
Access Permissions



Immutable Permission

- aliases = N
- access= R
- “constant”
- no synchronization

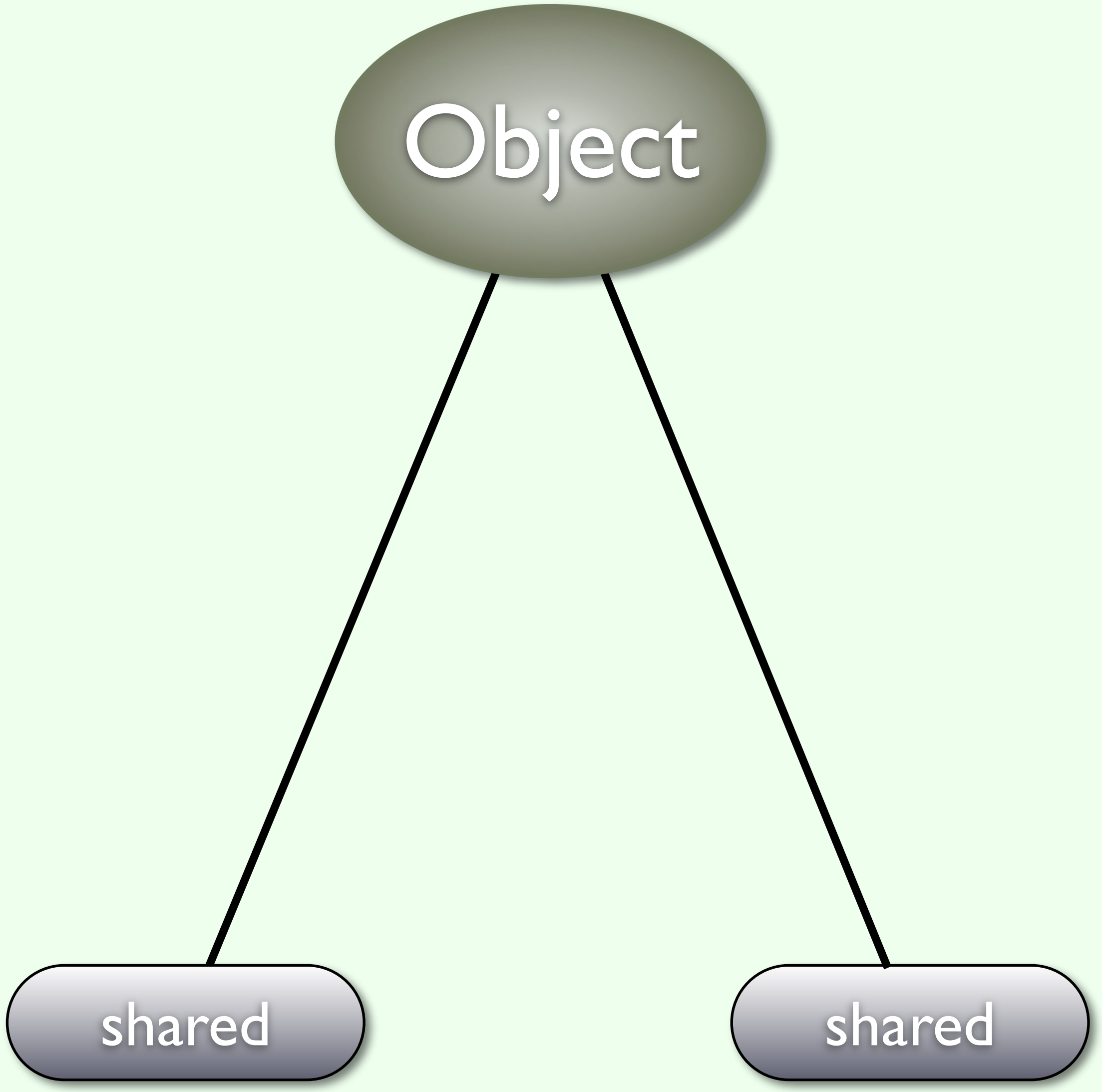
Access Permissions



Unique Permission

- aliases = 1
- access= RW
- “thread local”
- no synchronization

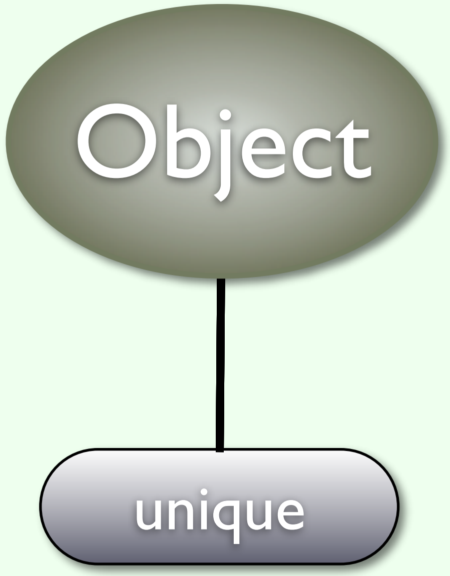
Access Permissions



Shared Permission

- aliases = N
- access= RW
- “shared data”
- requires synchronization

Access Permissions



Unique Permission

- aliases = 1
- access= RW
- “thread local”
- no synchronization

Permission Example

```
public void deposit(unique Account account, immutable Amount amount) {...}
public void withdraw(unique Account account, immutable Amount amount){...}


public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {


    withdraw(from, amount);


    deposit(to, amount);


}
```

Permission Example

```
public void deposit(unique Account account, immutable Amount amount) {...}
public void withdraw(unique Account account, immutable Amount amount){...}


public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {


   withdraw(from, amount);


   deposit(to, amount);


}
```

Syntax: permission [>> permission] type var

BORROW: unique Account from

unique >> unique Account from

CHANGE: unique >> immutable Account account

Permission Example

```
public void deposit(unique Account account, immutable Amount amount) {...}
public void withdraw(unique Account account, immutable Amount amount){...}

public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {

    withdraw(from, amount);

    deposit(to, amount);

}
```

Permission Example

```
public void deposit(unique Account account, immutable Amount amount) {...}
public void withdraw(unique Account account, immutable Amount amount){...}


public void transfer(unique Account from,
                            unique Account to,
                            immutable Amount amount) {
          // to: unique        from: unique        amount: immutable

    withdraw(from, amount);


    deposit(to, amount);


}
```

Permission Example

```
public void deposit(unique Account account, immutable Amount amount) {...}
public void withdraw(unique Account account, immutable Amount amount){...}


public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {
    // to: unique      from: unique      amount: immutable
  withdraw(from, amount);


  deposit(to, amount);


}
```

Permission Example

```
public void deposit(unique Account account, immutable Amount amount) {...}
public void withdraw(unique Account account, immutable Amount amount){...}


public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {
      // to: unique        from:             amount:
    withdraw(from, amount);


    deposit(to, amount);


}
```

Permission Example

```
public void deposit(unique Account account, immutable Amount amount) {...}
public void withdraw(unique Account account, immutable Amount amount){...}


public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {

    withdraw(from, amount);

        // to: unique        from:              amount:

    deposit(to, amount);


}
```

Permission Example

```
public void deposit(unique Account account, immutable Amount amount) {...}
public void withdraw(unique Account account, immutable Amount amount){...}


public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {

  withdraw(from, amount);
      // to: unique      from: unique      amount: immutable
  deposit(to, amount);


}
```

Permission Example

```
public void deposit(unique Account account, immutable Amount amount) {...}
public void withdraw(unique Account account, immutable Amount amount){...}


public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {


    withdraw(from, amount);

        // to: unique      from: unique      amount: immutable
    deposit(to, amount);


}
```

Permission Example

```
public void deposit(unique Account account, immutable Amount amount) {...}
public void withdraw(unique Account account, immutable Amount amount){...}


public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {


   withdraw(from, amount);

        // to:                from: unique        amount:

   deposit(to, amount);


}
```

Permission Example

```
public void deposit(unique Account account, immutable Amount amount) {...}
public void withdraw(unique Account account, immutable Amount amount){...}


public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {

   withdraw(from, amount);

   deposit(to, amount);

       // to:                 from: unique        amount:
}
```

Permission Example

```
public void deposit(unique Account account, immutable Amount amount) {...}
public void withdraw(unique Account account, immutable Amount amount){...}


public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {


   withdraw(from, amount);


   deposit(to, amount);

       // to: unique       from: unique       amount: immutable
}
```

Permission Example

```
public void deposit(unique Account account, immutable Amount amount) {...}
public void withdraw(unique Account account, immutable Amount amount){...}

public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {

    withdraw(from, amount);

    deposit(to, amount);
        // to: unique        from: unique        amount: immutable
}
```

Permission Example

```
public void deposit(unique Account account, immutable Amount amount) {...}
public void withdraw(unique Account account, immutable Amount amount){...}


public void transfer(unique Account from,
                                unique Account to,
                                immutable Amount amount) {


    withdraw(from, amount);


    deposit(to, amount);


}
```

Using Permissions for Parallelization

- infer **permissions flow** based on **lexical order**
- define operations can run in parallel iff **intersection** of their required permissions does **not contain unique permissions**

Dataflow Example

```
transfer(unique Account from, unique Account to, immutable Amount amount)
```

Dataflow Example

transfer(unique Account from, unique Account to, immutable Amount amount)

from: unique   to: unique   amount: immutable

Dataflow Example

transfer(unique Account from, unique Account to, immutable Amount amount)

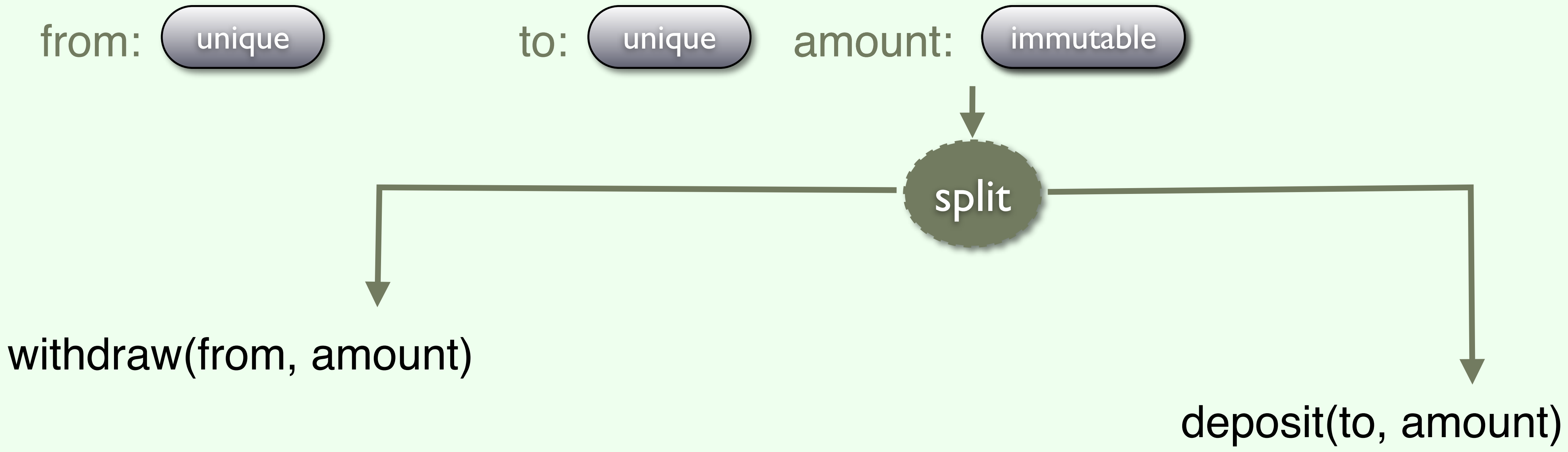
from: unique   to: unique   amount: immutable

withdraw(from, amount)

deposit(to, amount)

Dataflow Example

transfer(unique Account from, unique Account to, immutable Amount amount)



from: unique

to: unique

amount: immutable

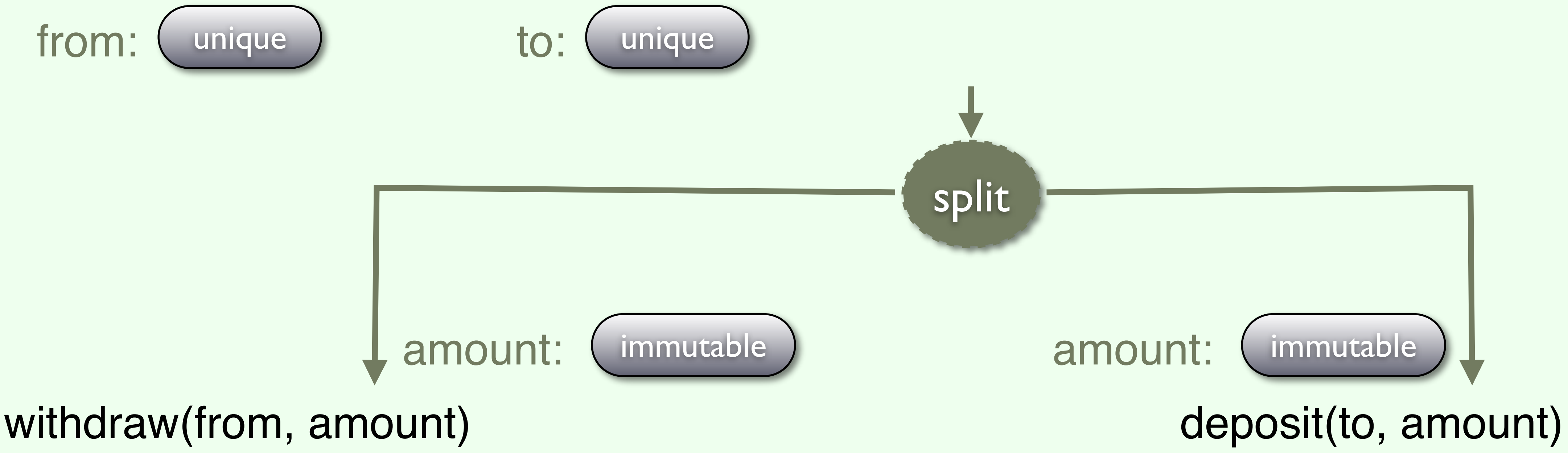
split

withdraw(from, amount)

deposit(to, amount)

Dataflow Example

transfer(unique Account from, unique Account to, immutable Amount amount)



from: unique

to: unique

split

amount: immutable

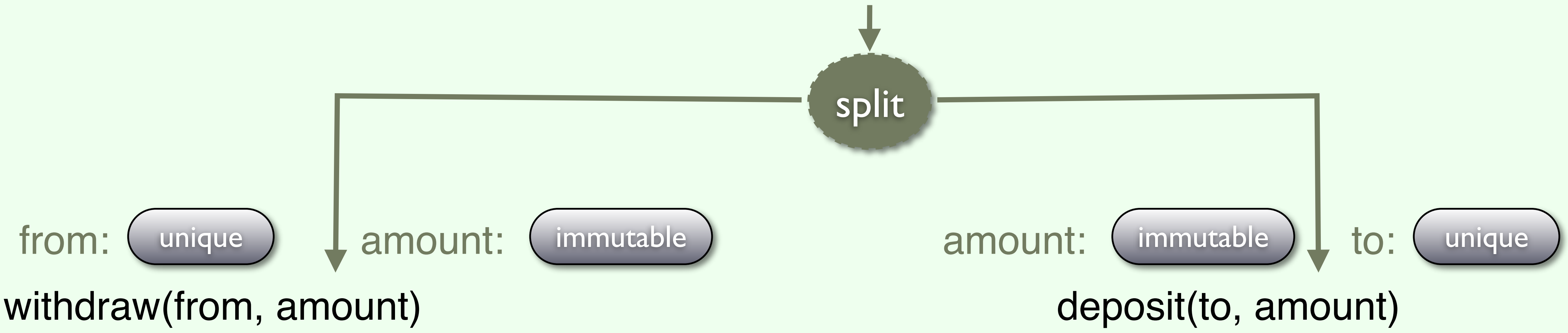
amount: immutable

withdraw(from, amount)

deposit(to, amount)

Dataflow Example

transfer(unique Account from, unique Account to, immutable Amount amount)



split

from: unique

amount: immutable

withdraw(from, amount)

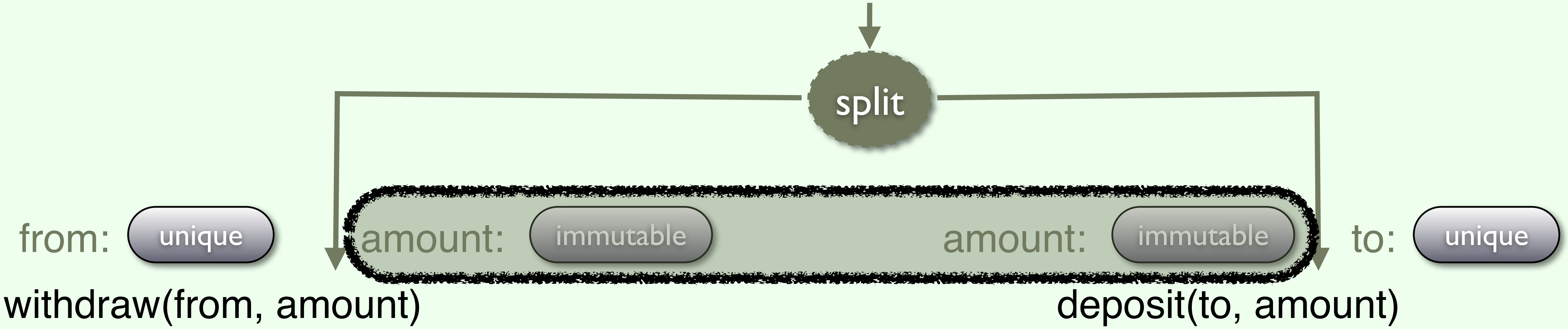
amount: immutable

to: unique

deposit(to, amount)

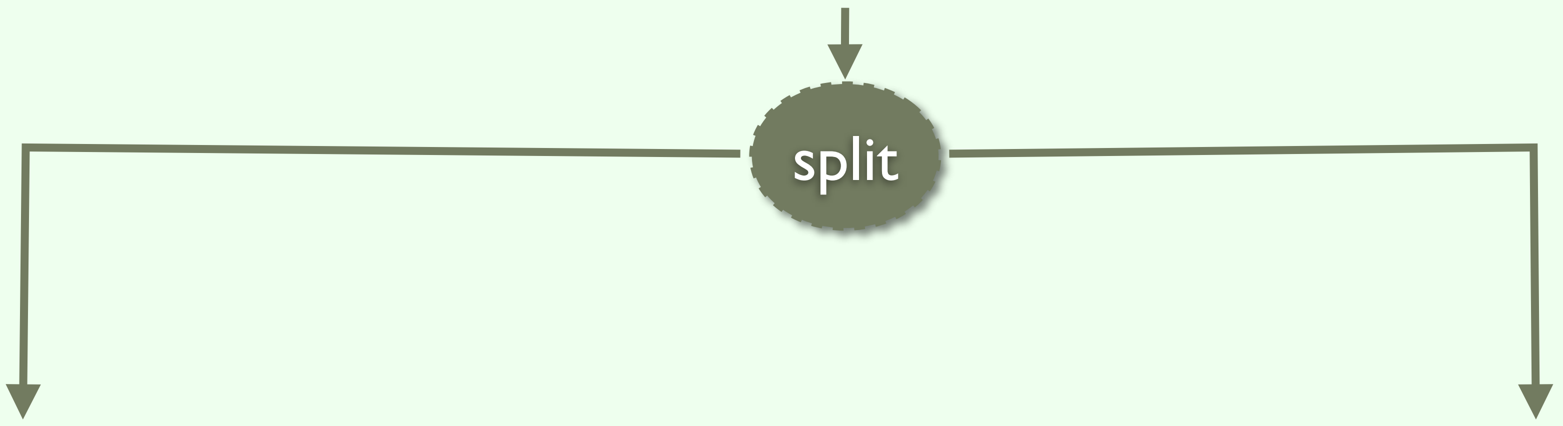
Dataflow Example

transfer(unique Account from, unique Account to, immutable Amount amount)



Dataflow Example

transfer(unique Account from, unique Account to, immutable Amount amount)



withdraw(from, amount)

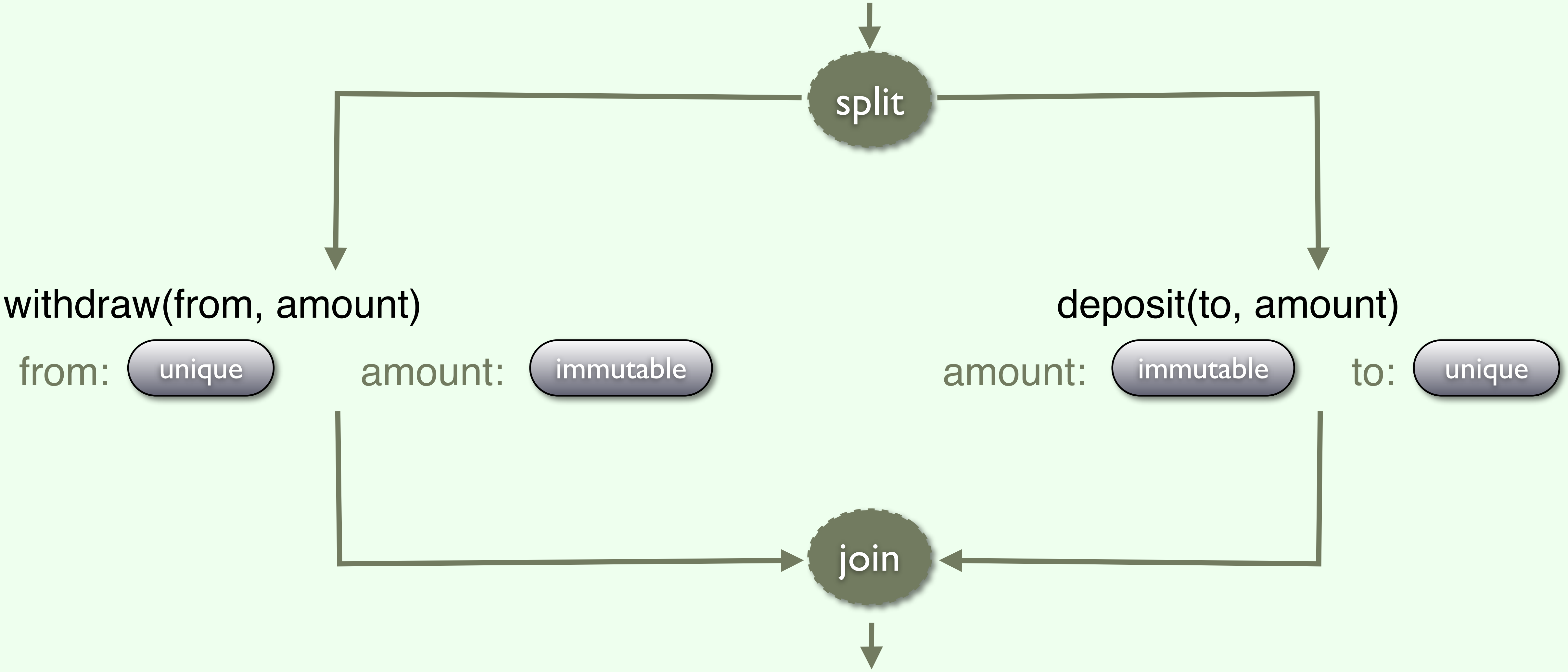
from: unique amount: immutable

deposit(to, amount)

amount: immutable to: unique

Dataflow Example

transfer(unique Account from, unique Account to, immutable Amount amount)



split

withdraw(from, amount)

from: unique amount: immutable

deposit(to, amount)

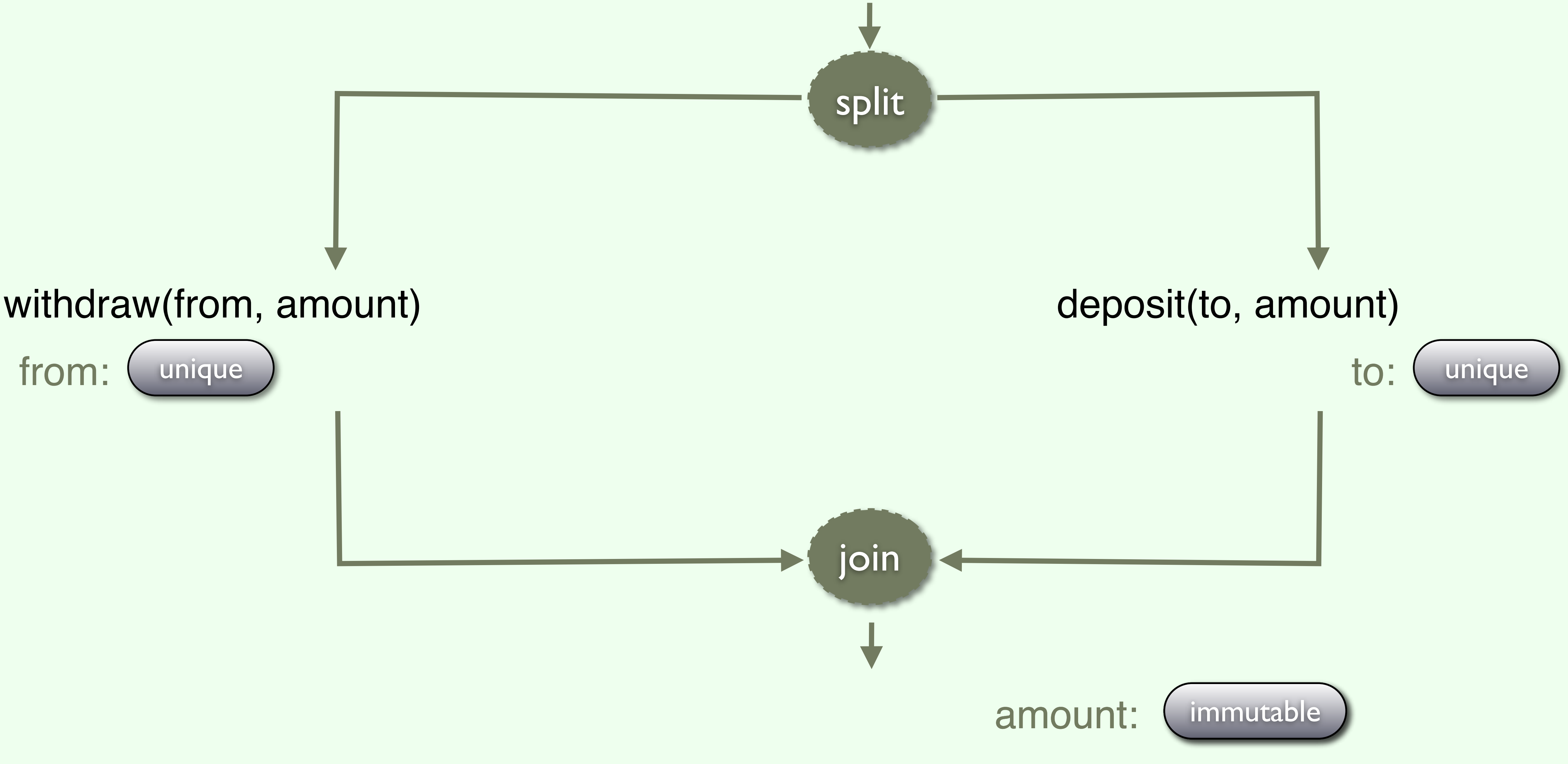
amount: immutable to: unique

join

}

Dataflow Example

transfer(unique Account from, unique Account to, immutable Amount amount)



split

withdraw(from, amount)

from: unique

deposit(to, amount)

to: unique

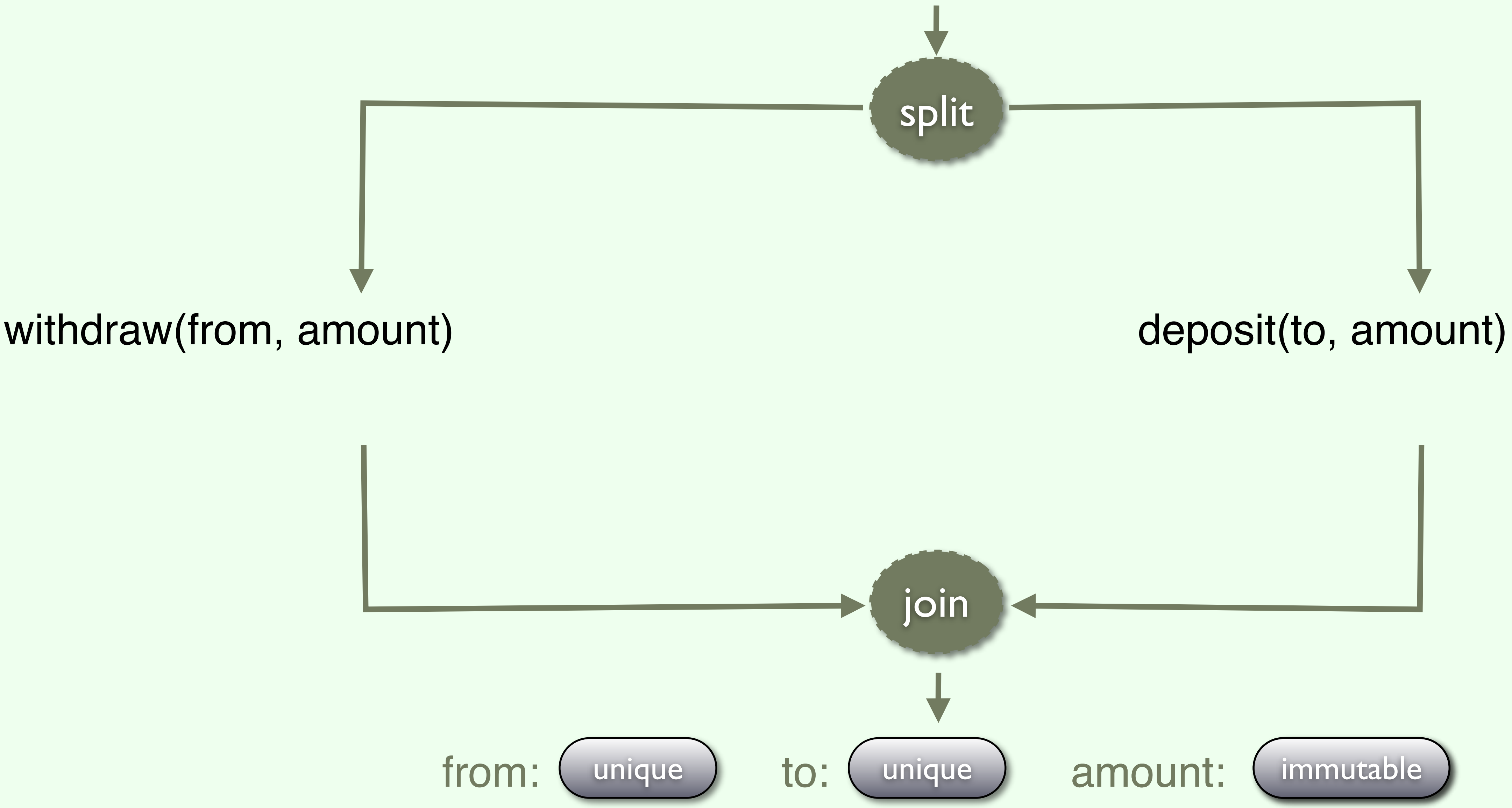
join

amount: immutable

}

Dataflow Example

transfer(unique Account from, unique Account to, immutable Amount amount)



split

withdraw(from, amount)

deposit(to, amount)

join

from: unique

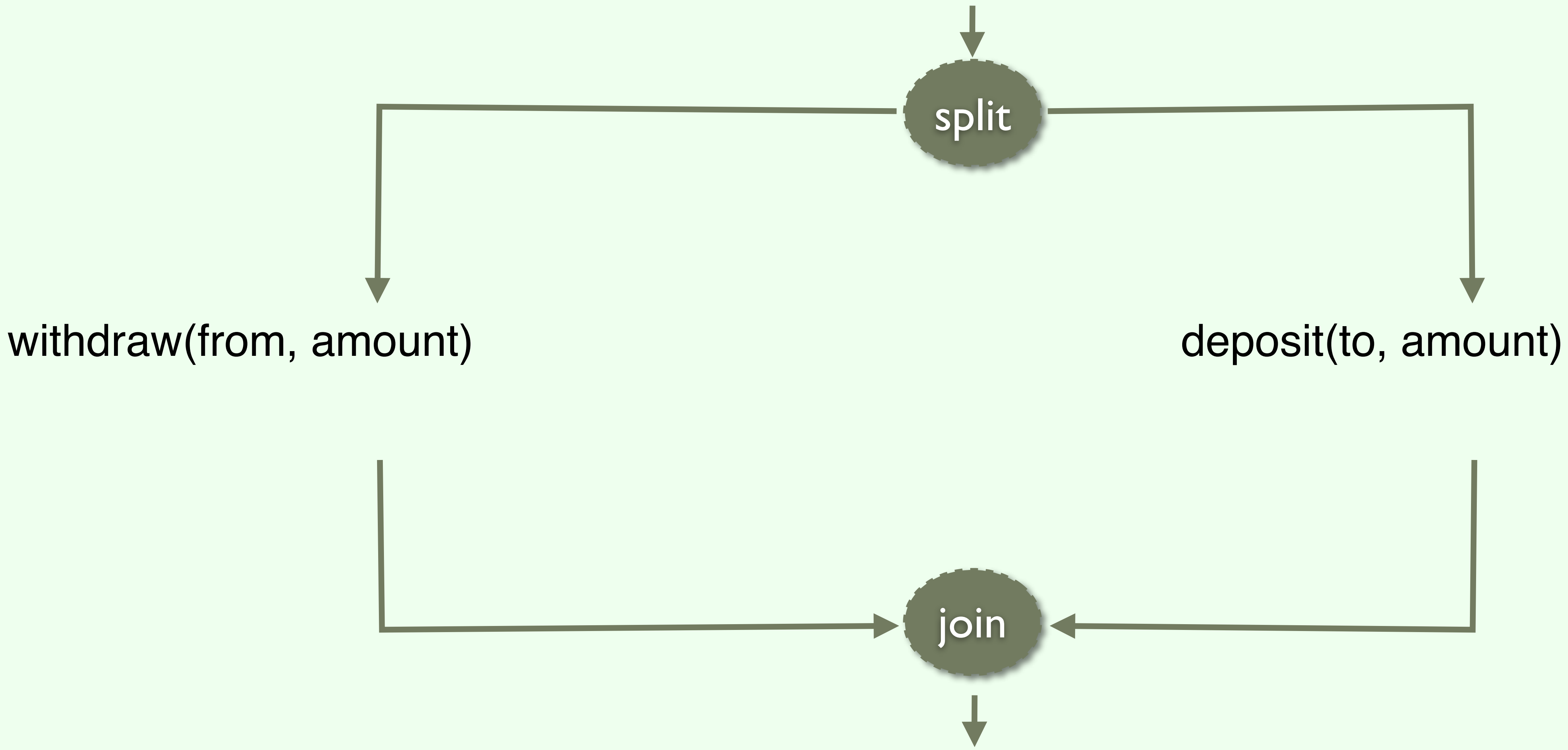
to: unique

amount: immutable

}

Dataflow Example

transfer(unique Account from, unique Account to, immutable Amount amount)



withdraw(from, amount)

deposit(to, amount)

}

Shared Data Issues

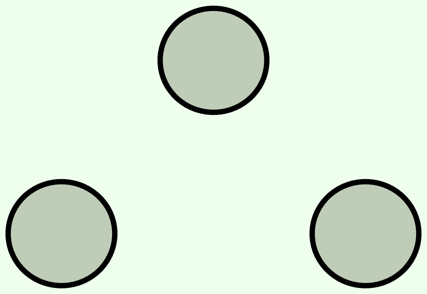
- causes **non-determinism** but sometimes order matters
  - e.g., object that needs to follow protocol
- all accesses to shared objects **require synchronization**
  - sometimes shared permissions are unavoidable
    - e.g., doubly linked list

Data Groups

- bundle shared objects into data groups
    - abstract collection of objects
    - disjoint partitions of heap

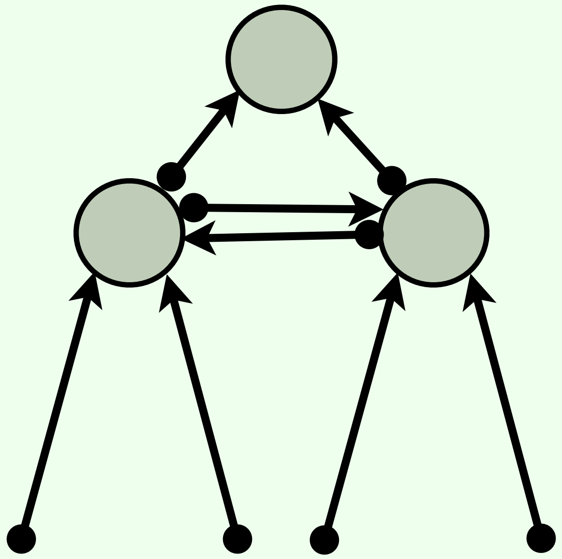
Data Groups

- bundle shared objects into data groups
  - abstract collection of objects
  - disjoint partitions of heap



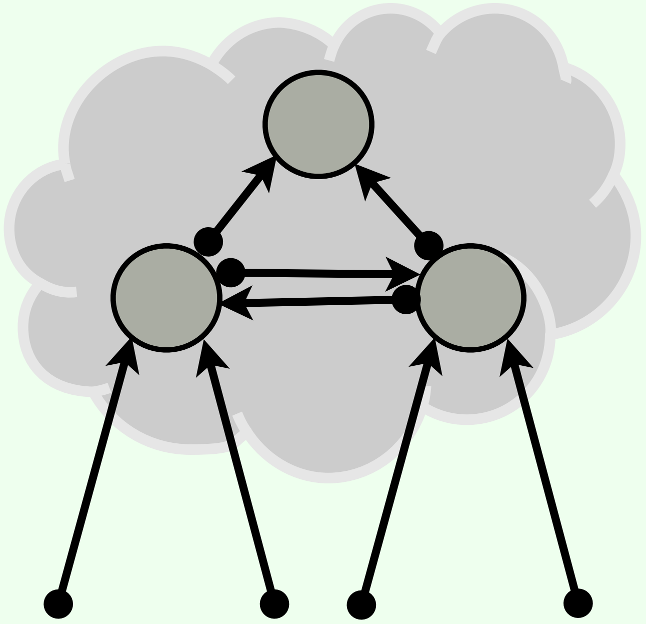
Data Groups

- bundle shared objects into data groups
  - abstract collection of objects
  - disjoint partitions of heap



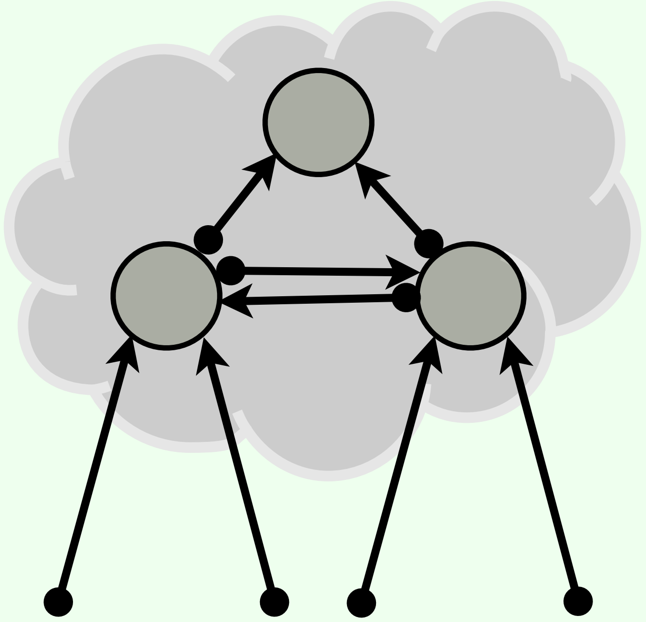
Data Groups

- bundle shared objects into data groups
  - abstract collection of objects
  - disjoint partitions of heap



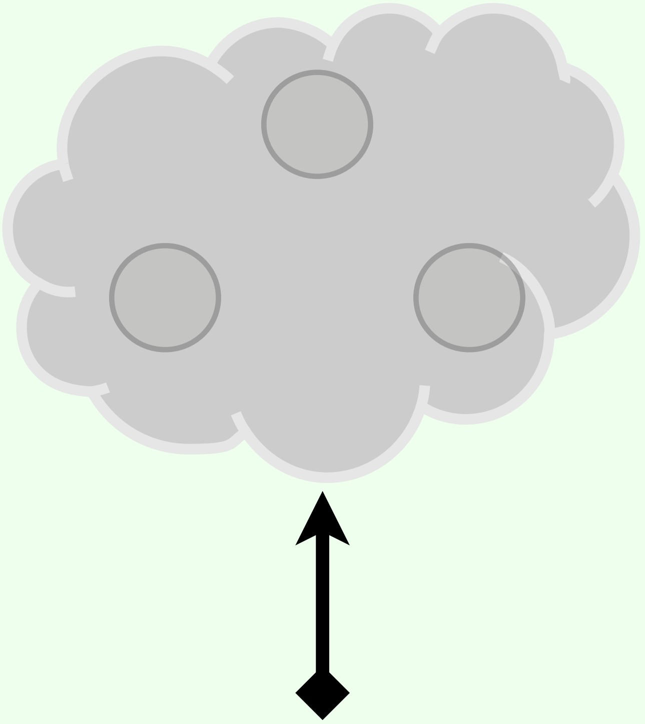
Data Groups Permissions

- similar to access permissions for data groups
- manual split/joining by user
- user controlled mechanism for granularity



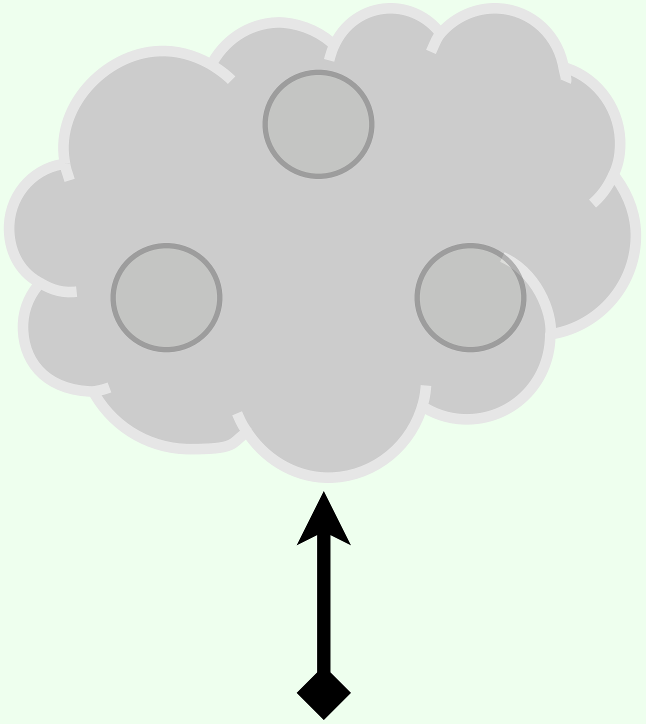
Data Groups Permissions

- similar to access permissions for data groups
- manual split/joining by user
- user controlled mechanism for granularity

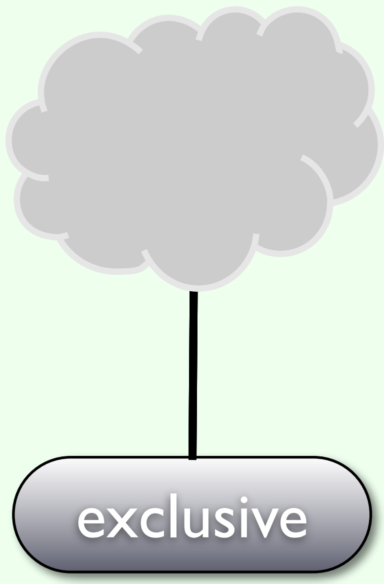


Data Groups Permissions

- data groups are embedded in objects
  - strong encapsulation, ownership
- group permissions are derived from receiver permissions



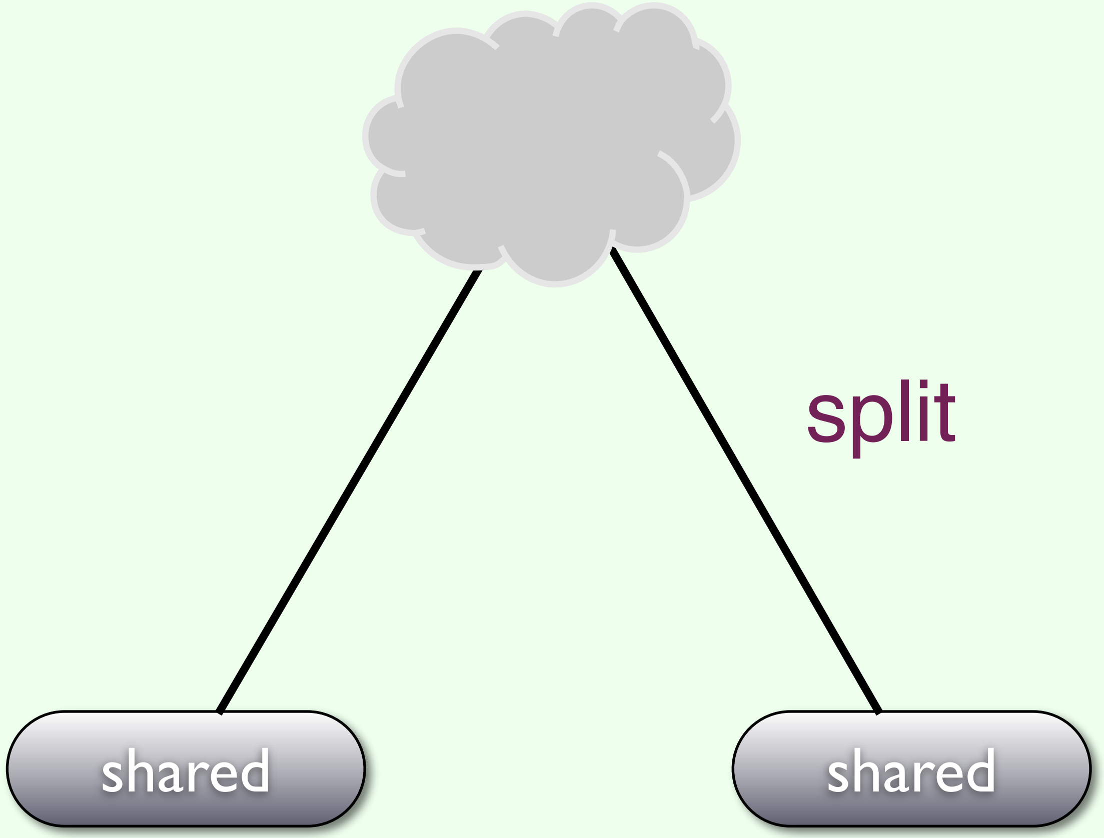
Group Permissions



Exclusive Permission

- aliases = 1
- access= RW
- “thread local”
- no synchronization

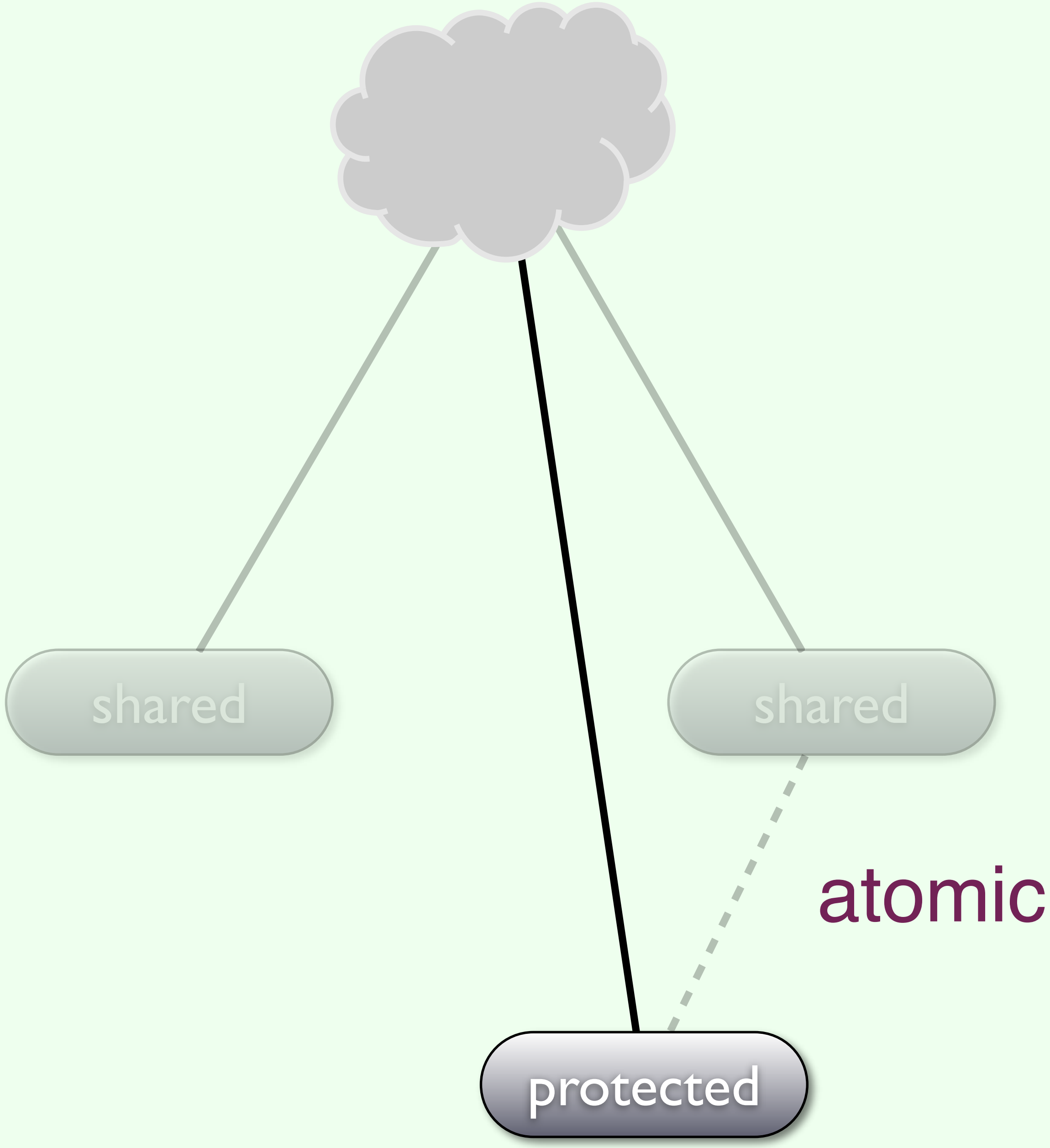
Group Permissions



Shared Permission

- aliases = N
- access= none
- “shared data”
- requires synchronization

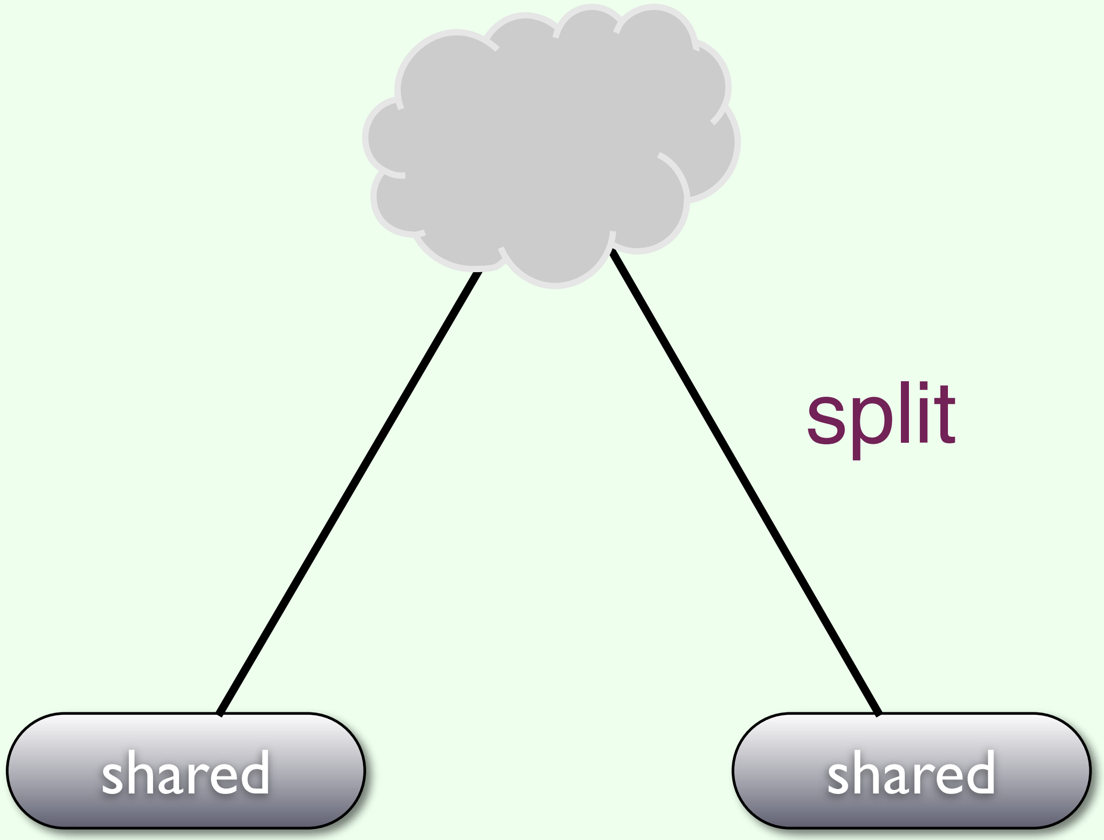
Group Permissions



atomic Permission

- aliases = 1
- access= RW
- “protected ”
- is synchronized

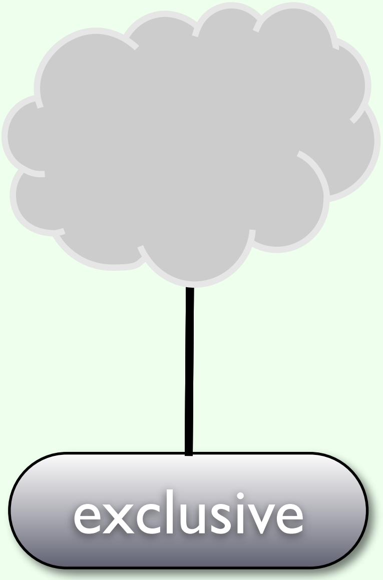
Group Permissions



Shared Permission

- aliases = N
- access= none
- “shared data”
- requires synchronization

Group Permissions

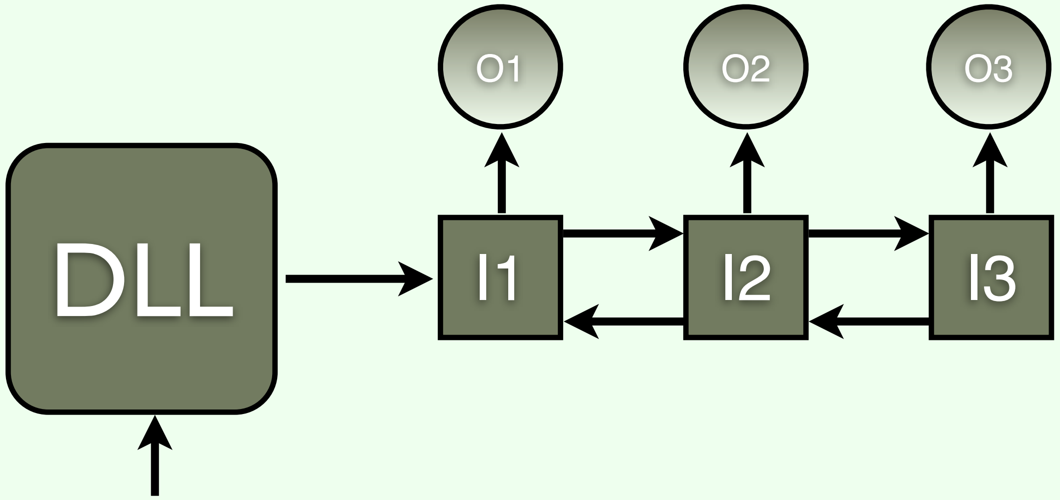


Exclusive Permission

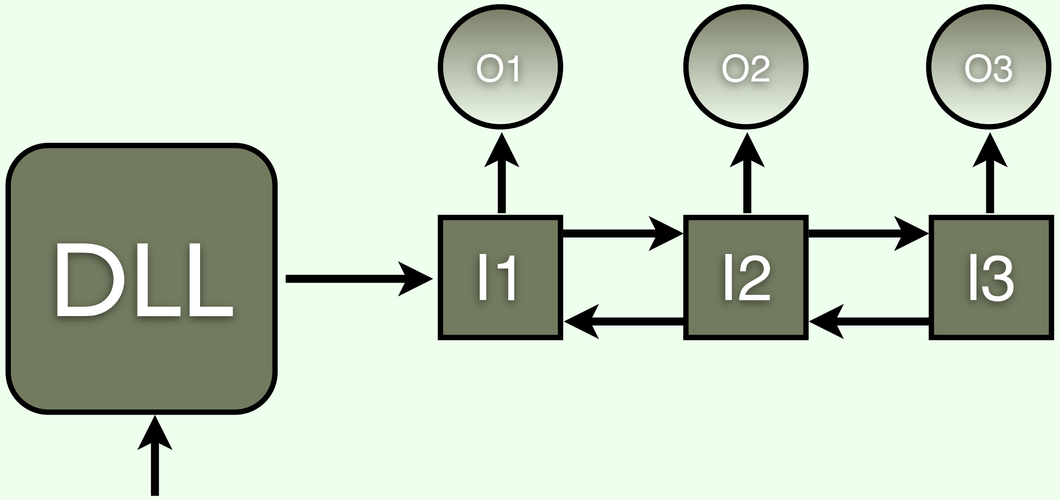
- aliases = 1
- access= RW
- “thread local”
- no synchronization

Data Group Example

```
class DLLItem {
    public Object data;
    public DLLItem prev;
    public DLLItem next;
}

public class DLL {
    private DLLItem head;

    public void add(Object data) {
         DLLItem li = new DLLItem();
         this.head.prev = li;
         li.next = this.head;
         li.data = data;
         this.head = li;
    }
}
```



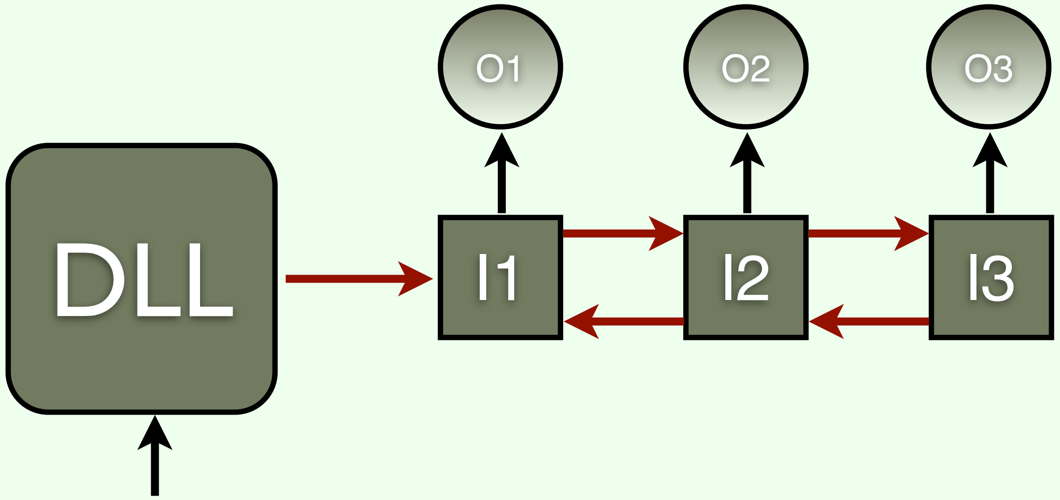
Data Group Example



```
class DLLItem {
    public        Object data;
    public        DLLItem prev;
    public        DLLItem next;
}

public class DLL {
    private        DLLItem head;

    public void add(               Object data)          {
                DLLItem li = new DLLItem();
         this.head.prev = li;
         li.next = this.head;
         li.data = data;
         this.head = li;
    }
}
```

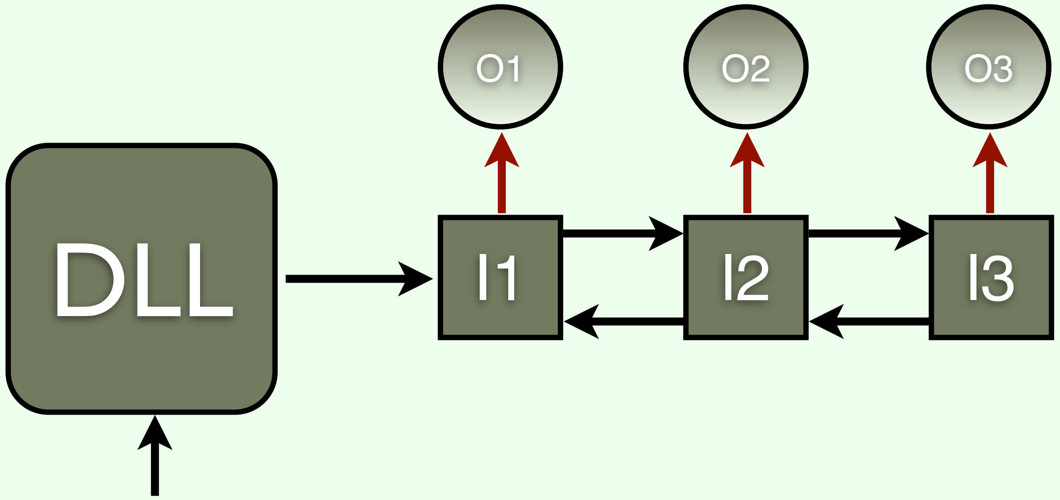
Data Group Example

```
class DLLItem {
    public        Object data;
    public shared DLLItem prev;
    public shared DLLItem next;
}

public class DLL {
    private shared DLLItem head;

    public void add(              Object data)          {
        shared DLLItem li = new DLLItem();
        this.head.prev = li;
        li.next = this.head;
        li.data = data;
        this.head = li;
    }
}
```



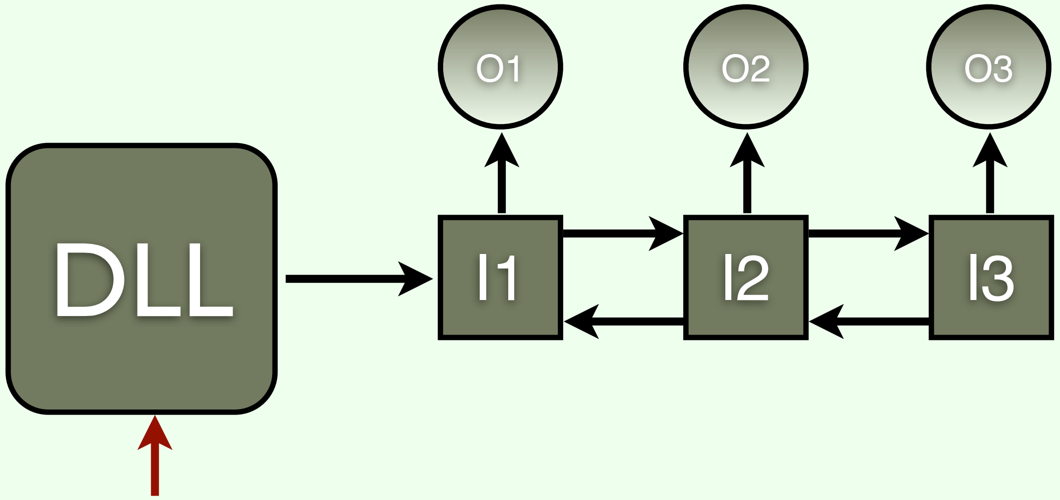
Data Group Example

```
class DLLItem {
    public unique Object data;
    public shared DLLItem prev;
    public shared DLLItem next;
}

public class DLL {
    private shared DLLItem head;

    public void add(unique >> none Object data)          {
         shared DLLItem li = new DLLItem();
         this.head.prev = li;
         li.next = this.head;
         li.data = data;
         this.head = li;
    }
}
```

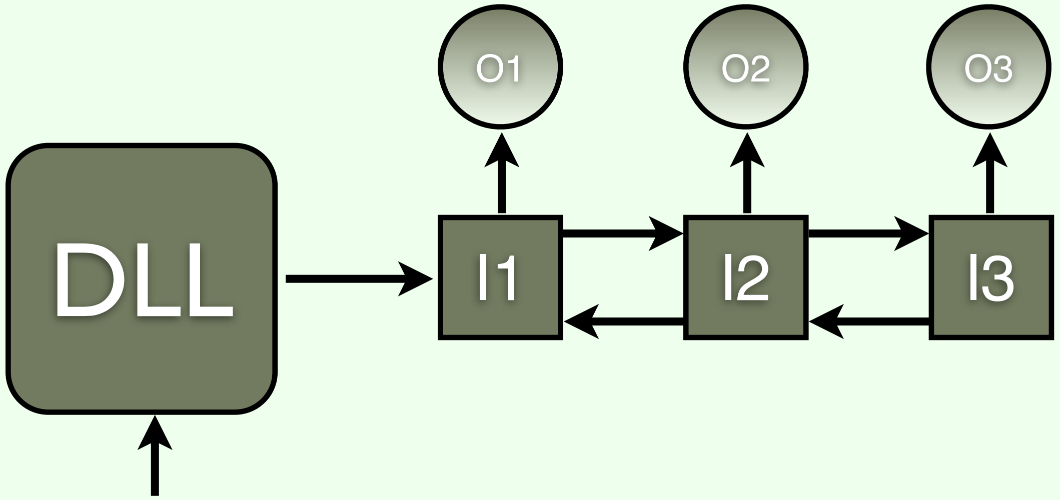


Data Group Example

```
class DLLItem {
    public unique Object data;
    public shared DLLItem prev;
    public shared DLLItem next;
}

public class DLL {
    private shared DLLItem head;

    public void add(unique >> none Object data) : unique {
        shared DLLItem li = new DLLItem();
        this.head.prev = li;
        li.next = this.head;
        li.data = data;
        this.head = li;
    }
}
```



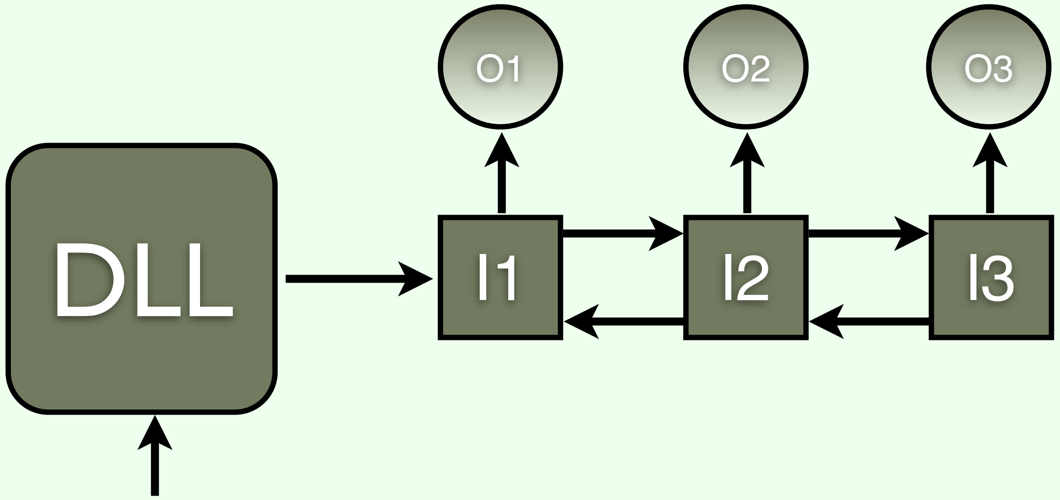
Data Group Example



```
class DLLItem {
    public unique Object data;
    public shared DLLItem prev;
    public shared DLLItem next;
}

public class DLL {
    private shared DLLItem head;

    public void add(unique >> none Object data) : unique {
         shared DLLItem li = new DLLItem();
         this.head.prev = li;
         li.next = this.head;
         li.data = data;
         this.head = li;
    }
}
```

Data Group Example

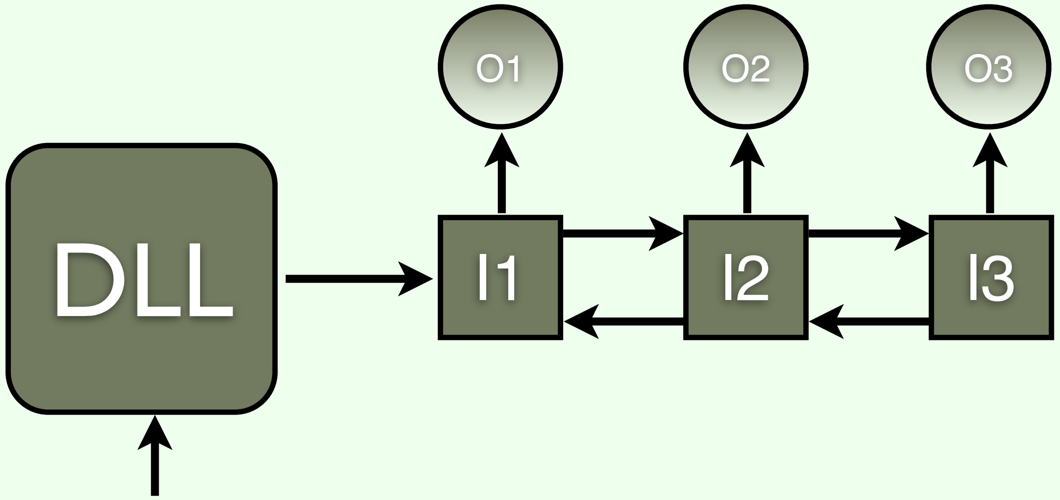
```
class DLLItem {
    public unique Object data;
    public shared DLLItem prev;
    public shared DLLItem next;
}

public class DLL {
    private shared DLLItem head;

    public void add(unique >> none Object data) : unique {
        shared DLLItem li = new DLLItem();
        this.head.prev = li;
        li.next = this.head;
        li.data = data;
        this.head = li;
    }
}
```



ERROR: Access shared data

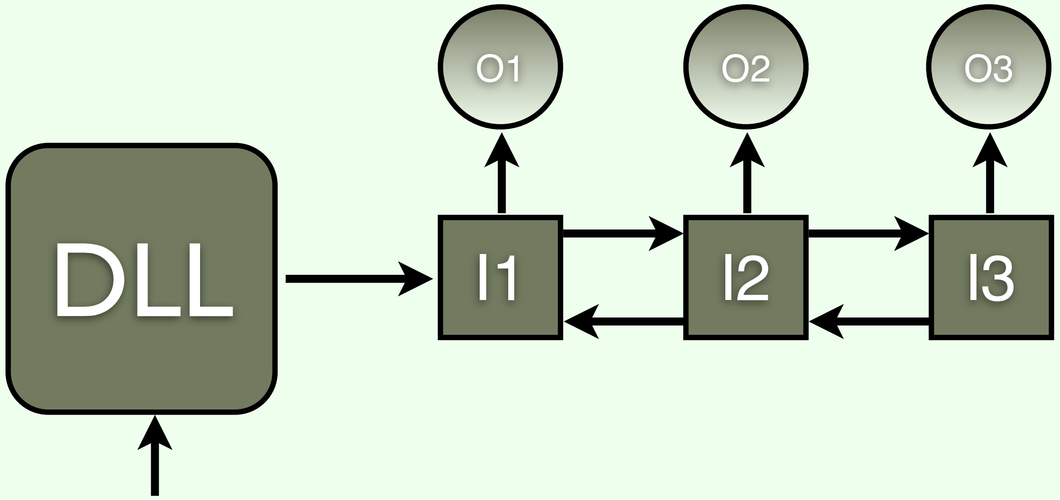
Data Group Example

```
class DLLItem {
    public unique Object data;
    public shared DLLItem prev;
    public shared DLLItem next;
}

public class DLL {
    private shared DLLItem head;

    public void add(unique >> none Object data) : unique {
        atomic {
          shared DLLItem li = new DLLItem();
          this.head.prev = li;
          li.next = this.head;
          li.data = data;
          this.head = li;
        }
    }
}
```



Data Group Example

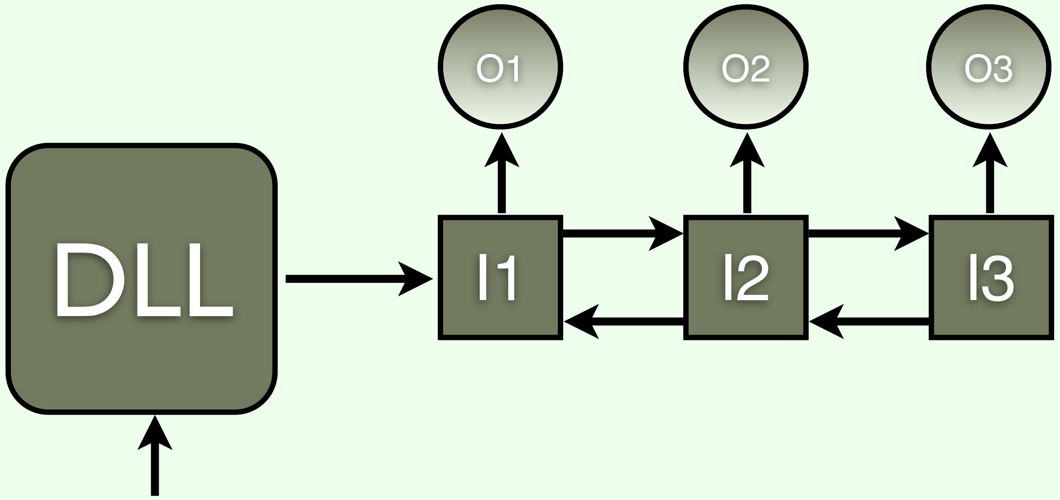
```
class DLLItem {
    public unique Object data;
    public shared DLLItem prev;
    public shared DLLItem next;
}

public class DLL {
    private shared DLLItem head;

    public void add(unique >> none Object data) : unique {
       atomic {
         shared DLLItem li = new DLLItem();
         this.head.prev = li;
         li.next = this.head;
         li.data = data;
         this.head = li;
       }
    }
}
```



Unique receiver means no aliases

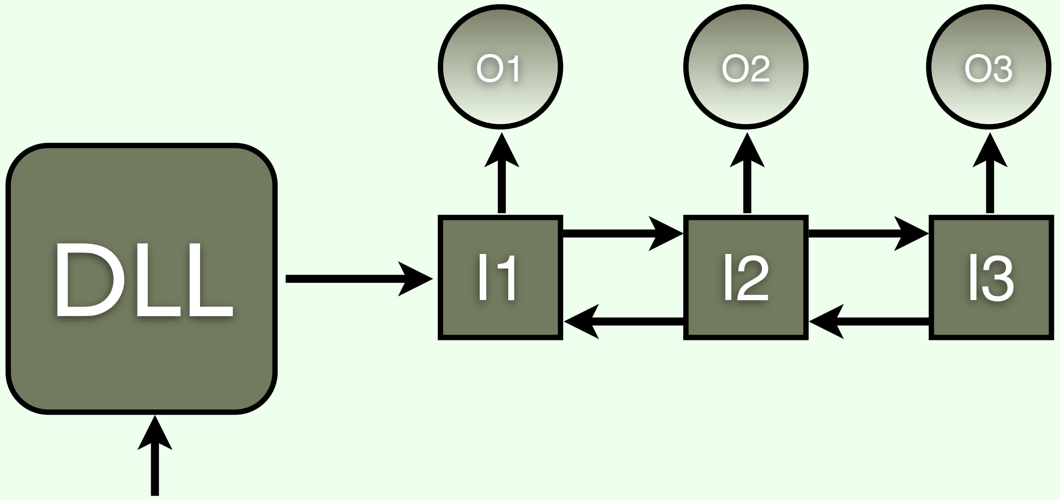
Data Group Example

```
class DLLItem    {
    public unique Object data;
    public shared DLLItem    prev;
    public shared DLLItem    next;
}

public class DLL {
    group nodes;
    private shared DLLItem head;

    public void add(unique >> none Object data) : unique {

        shared DLLItem        li = new DLLItem      ();
        this.head.prev = li;
        li.next = this.head;
        li.data = data;
        this.head = li;

    }
}
```



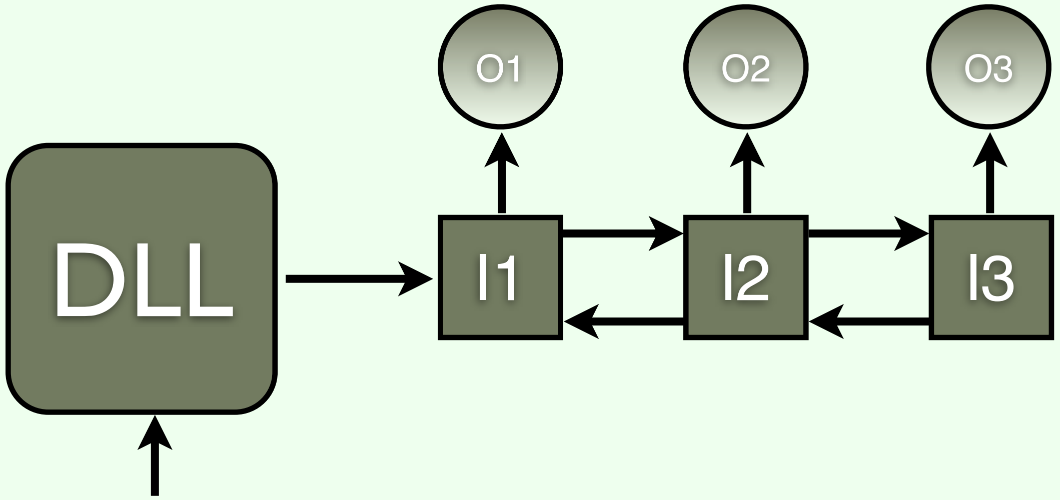
Data Group Example

```
class DLLItem<G> {
    public unique Object data;
    public shared DLLItem<G> prev;
    public shared DLLItem<G> next;
}

public class DLL {
    group nodes;
    private shared DLLItem head;

    public void add(unique >> none Object data) : unique {

        shared DLLItem        li = new DLLItem      ();
        this.head.prev = li;
        li.next = this.head;
        li.data = data;
        this.head = li;

    }
}
```



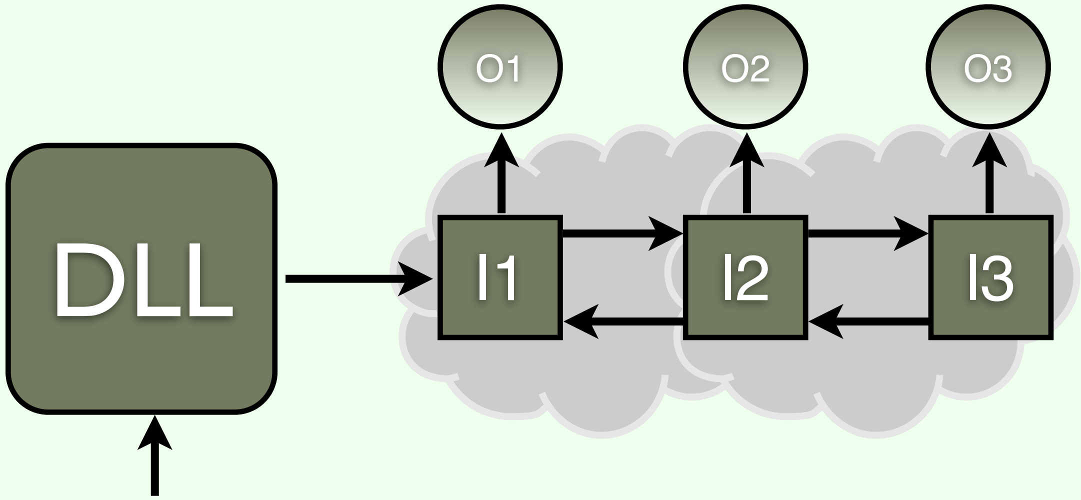
Data Group Example

```
class DLLItem<G> {
    public unique Object data;
    public shared DLLItem<G> prev;
    public shared DLLItem<G> next;
}

public class DLL {
    group nodes;
    private shared DLLItem head;

    public void add(unique >> none Object data) : unique {

        shared DLLItem<nodes> li = new DLLItem<nodes>();
        this.head.prev = li;
        li.next = this.head;
        li.data = data;
        this.head = li;

    }
}
```



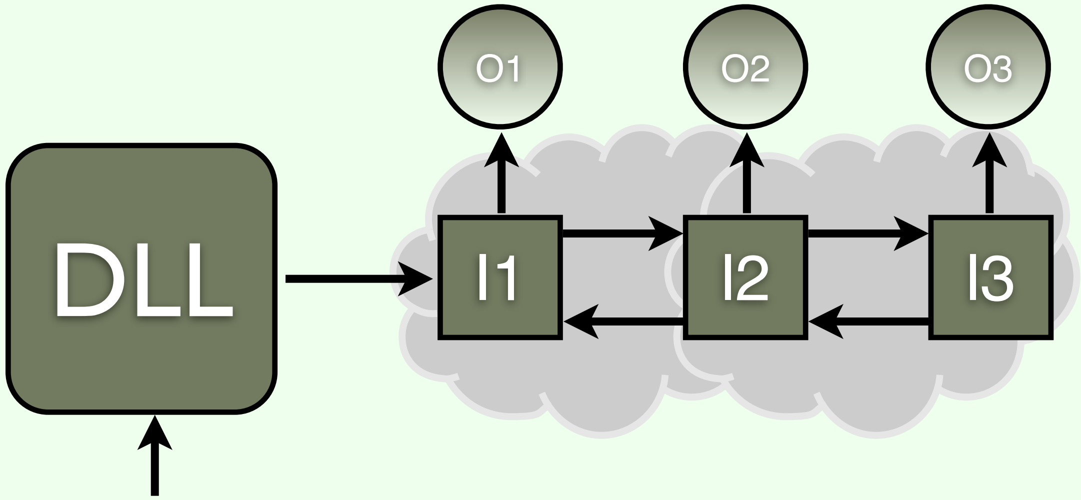
Data Group Example

```
class DLLItem<G> {
    public unique Object data;
    public shared DLLItem<G> prev;
    public shared DLLItem<G> next;
}

public class DLL {
    group nodes;
    private shared DLLItem head;

    public void add(unique >> none Object data) : unique {

        shared DLLItem<nodes> li = new DLLItem<nodes>();
        this.head.prev = li;
        li.next = this.head;
        li.data = data;
        this.head = li;

    }
}
```



Data Group Example

```
class DLLItem<G> {
    public unique Object data;
    public shared DLLItem<G> prev;
    public shared DLLItem<G> next;
}

public class DLL {
    group nodes;
    private shared DLLItem head;

    public void add(unique >> none Object data) : unique {
       unpack {
         shared DLLItem<nodes> li = new DLLItem<nodes>();
         this.head.prev = li;
         li.next = this.head;
         li.data = data;
         this.head = li;
       }
    }
}
```



Data Group Example

```
public void add(unique >> none Object data) : unique {
   unpack {
      ...
      li.data = data;
   }

}
```

Data Group Example

```
public void add(unique >> none Object data) : unique {
    // this: unique        data: unique
    unpack {

        ...

        li.data = data;

    }

}
```

Data Group Example

```
public void add(unique >> none Object data) : unique {

    unpack {
        // this: unique     this.nodes: exclusive      data: unique
        ...

        li.data = data;

    }


}
```

Data Group Example

```
public void add(unique >> none Object data) : unique {

  unpack {

    ...
    // this: unique    this.nodes: exclusive      data: unique
    li.data = data;

  }


}
```

Data Group Example

```
public void add(unique >> none Object data) : unique {

    unpack {

        ...

        li.data = data;
        // this: unique    this.nodes: exclusive
    }


}
```

Data Group Example

```
public void add(unique >> none Object data) : unique {
    unpack {
        ...
        li.data = data;
    }
    // this: unique
}
```

Data Group Example

```
public void add(unique >> none Object data) : unique {
   unpack {
      ...
      li.data = data;
   }

}
```

Progress so far

µÆMINIUM

- core-calculus based on group permissions
- **concurrent-by-default** type system
- soundness proof for absence of **race conditions** (cf. ‘safety’ hypothesis)

Dataflow Runtime

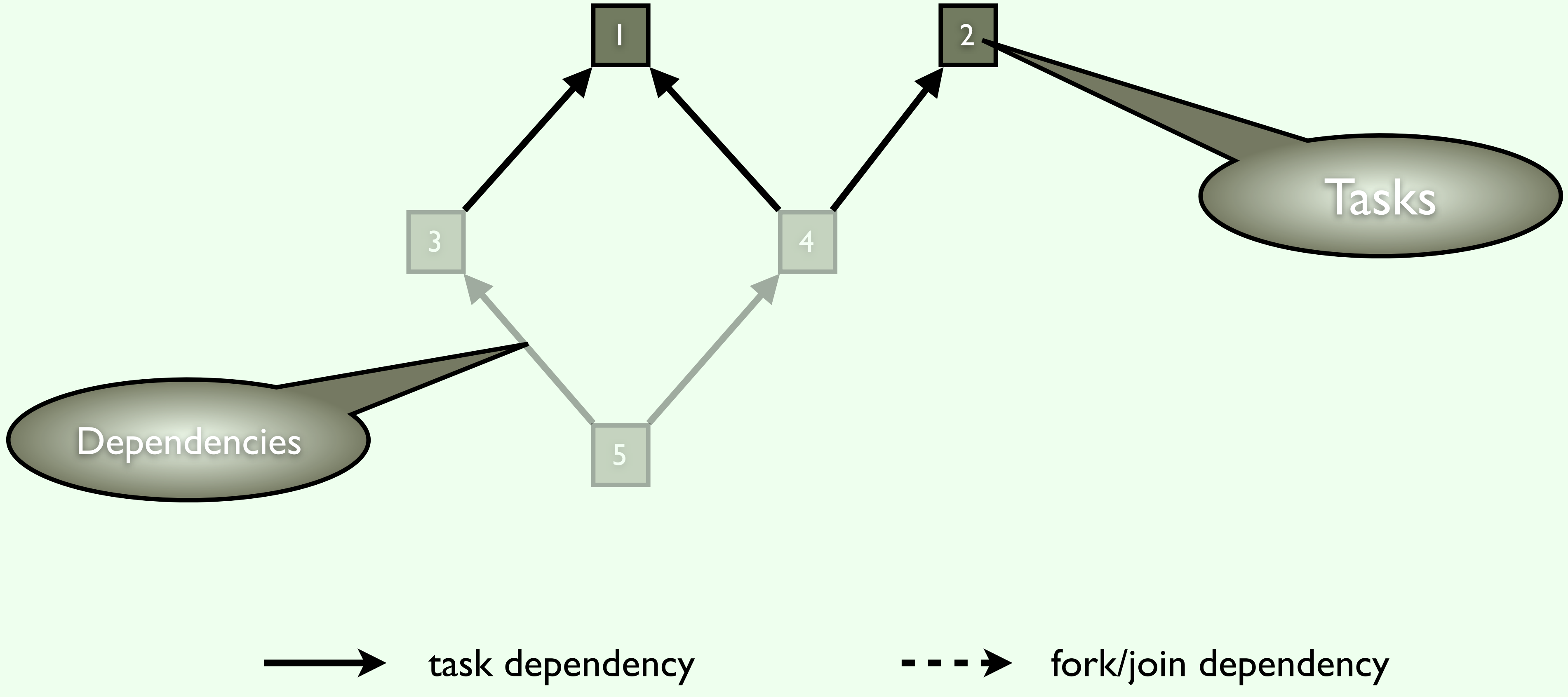
- data flow runtime system for ÆMINIUM
- **task** based runtime system for **dataflow** and **fork/join** parallelisms
- support for **locks** and **STM**
- **dynamic detection** of **deadlocks** (for the lock based approach)

Dataflow Runtime

- support for 3 kinds of tasks
- **Non-Blocking** -- computation intensive
- **Blocking** -- I/O tasks
- **Atomic** -- task that require protection

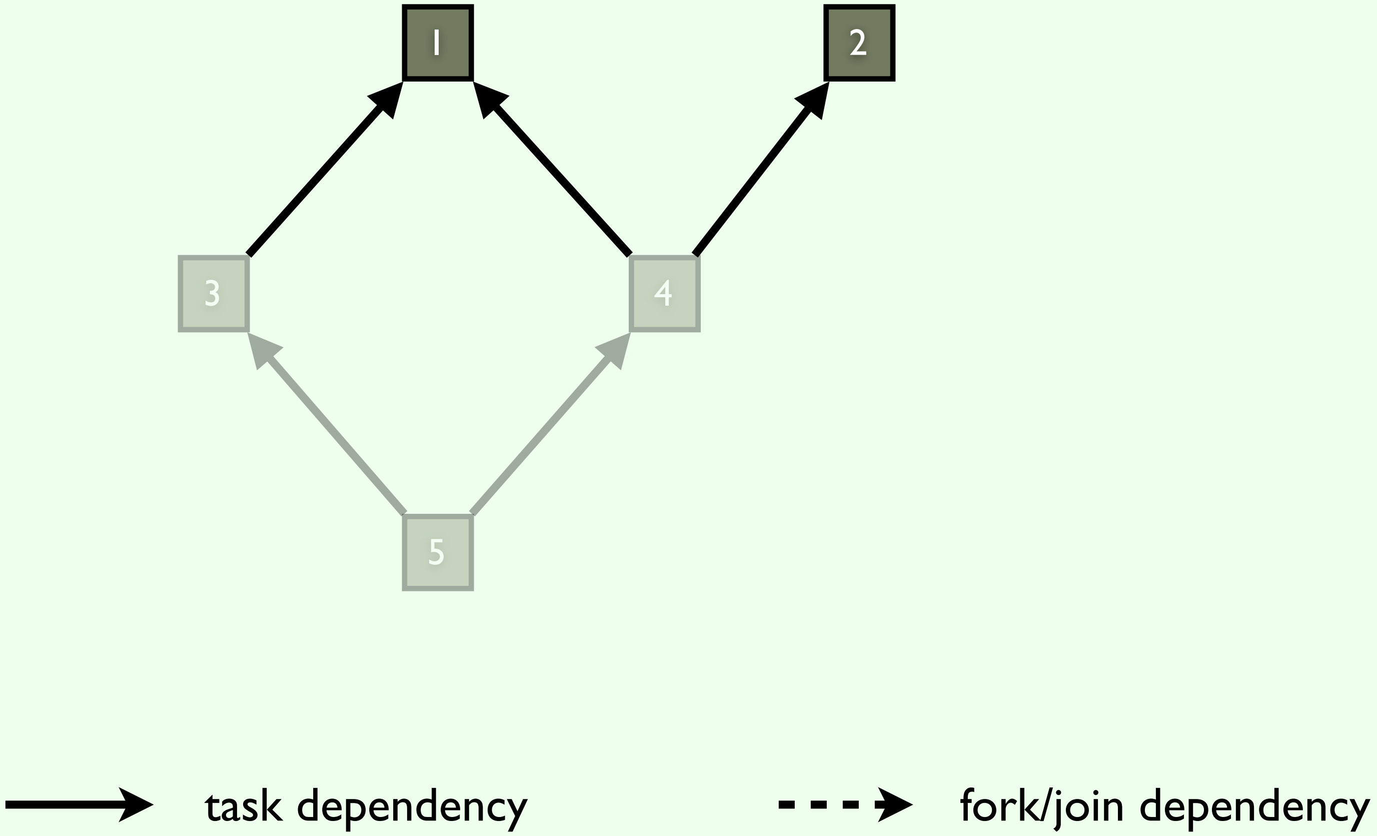
Dataflow Runtime

- data flow runtime system for ÆMINIUM
- task based support for **dataflow** and **fork/join** parallelisms



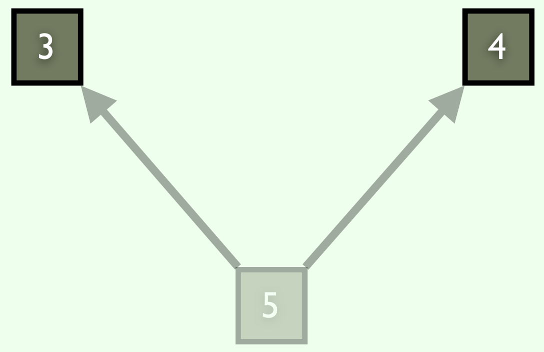
Dataflow Runtime

- data flow runtime system for ÆMINIUM
- task based support for **dataflow** and **fork/join** parallelisms



Dataflow Runtime

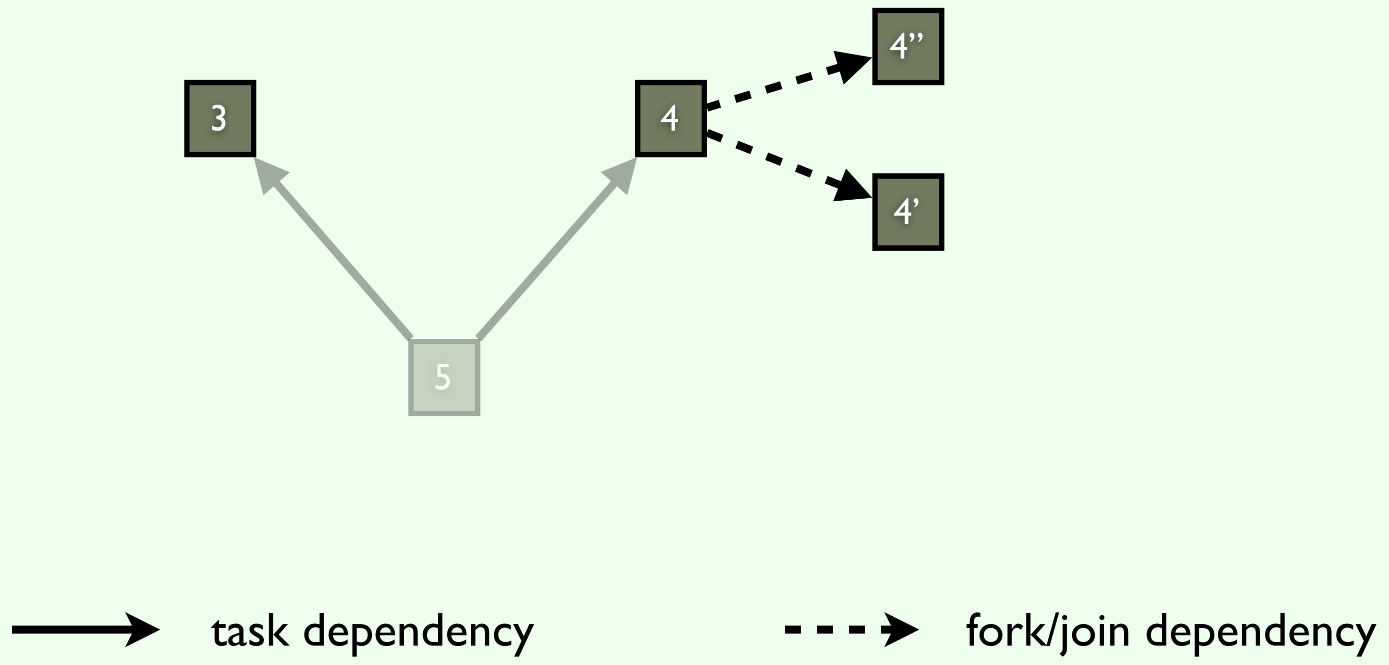
- data flow runtime system for ÆMINIUM
- task based support for **dataflow** and **fork/join** parallelisms



⟶ task dependency ⇢ fork/join dependency

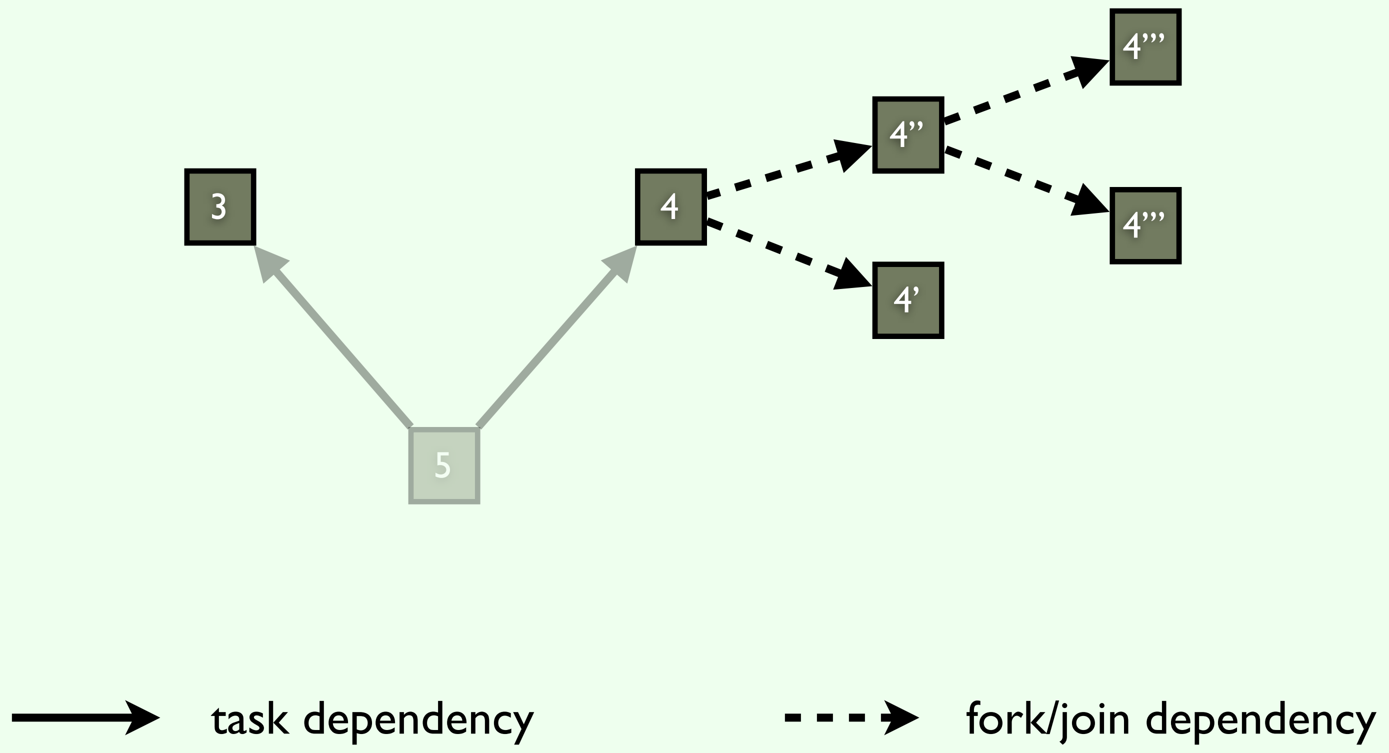
Dataflow Runtime

- data flow runtime system for ÆMINIUM
- task based support for **dataflow** and **fork/join** parallelisms



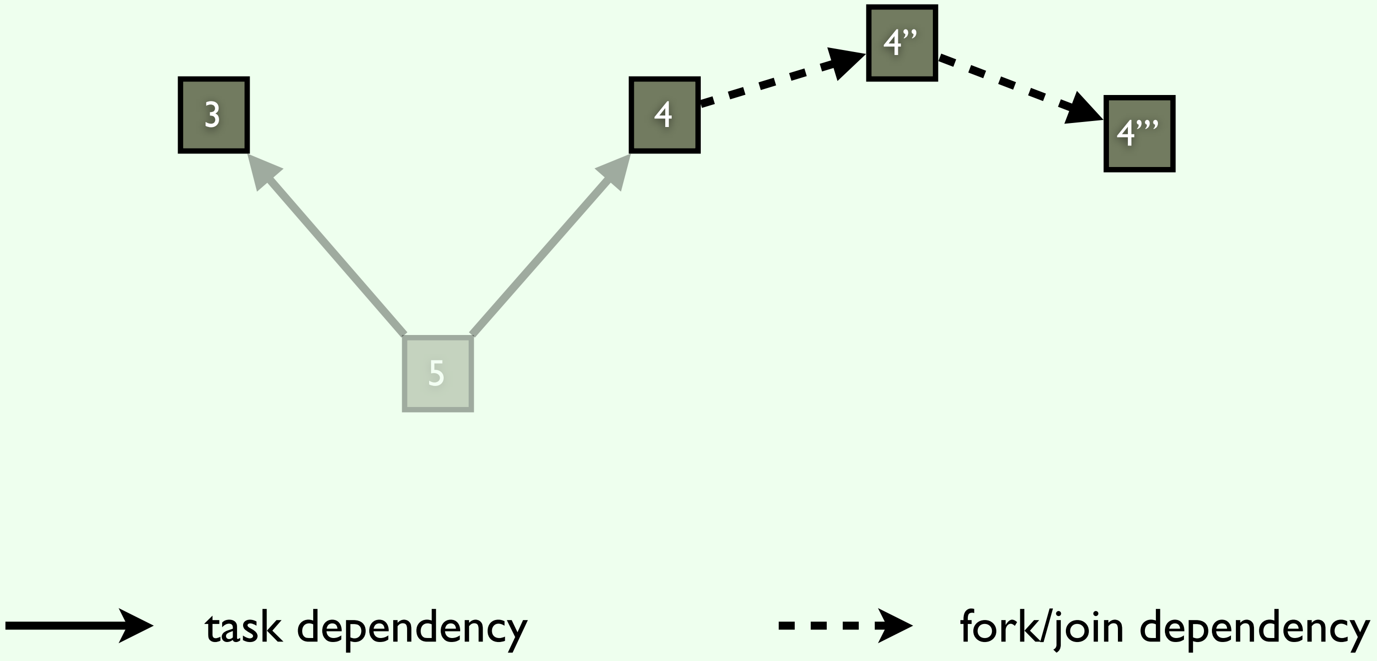
Dataflow Runtime

- data flow runtime system for ÆMINIUM
- task based support for **dataflow** and **fork/join** parallelisms



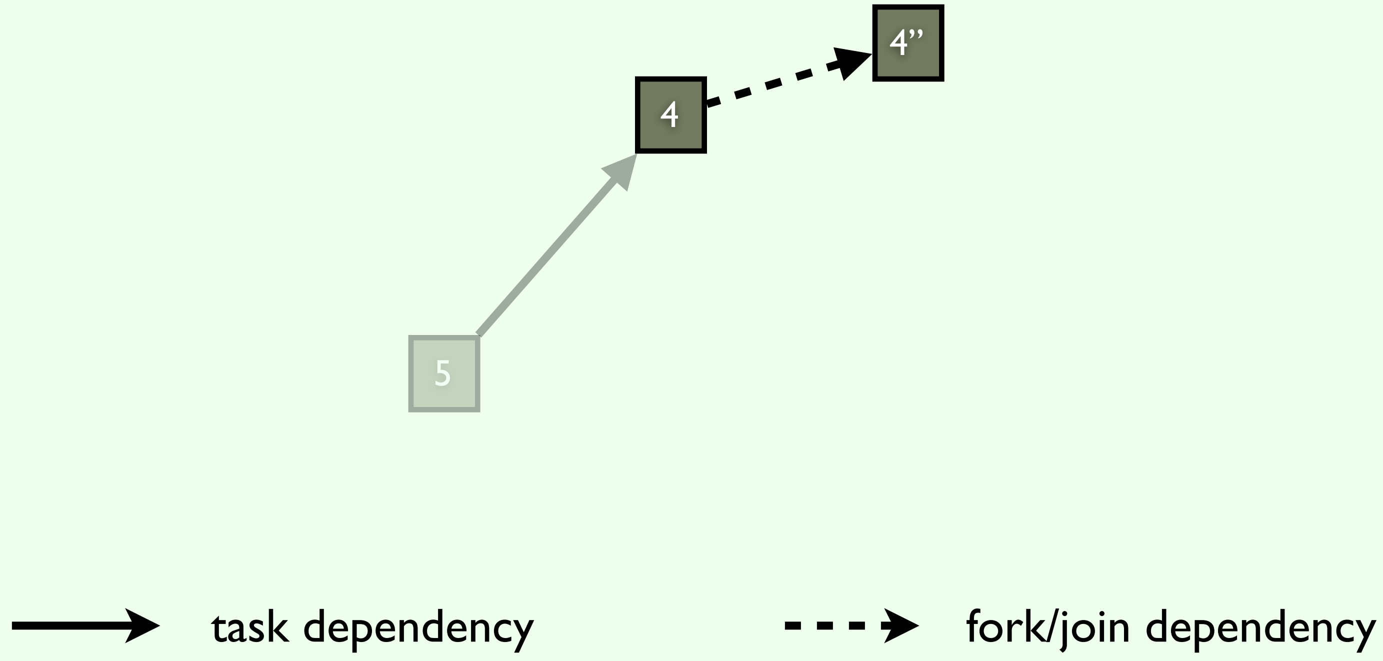
Dataflow Runtime

- data flow runtime system for ÆMINIUM
- task based support for **dataflow** and **fork/join** parallelisms



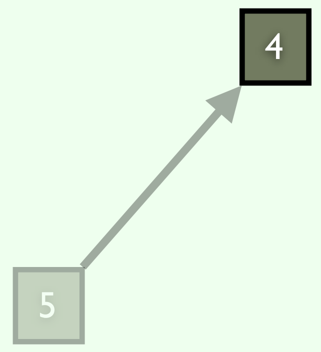
Dataflow Runtime

- data flow runtime system for ÆMINIUM
- task based support for **dataflow** and **fork/join** parallelisms



Dataflow Runtime

- data flow runtime system for ÆMINIUM
- task based support for **dataflow** and **fork/join** parallelisms



→ task dependency ⇢ fork/join dependency

Dataflow Runtime

- data flow runtime system for ÆMINIUM
- task based support for **dataflow** and **fork/join** parallelisms

Dataflow Runtime

- data flow runtime system for ÆMINIUM
- task based support for **dataflow** and **fork/join** parallelisms

⟶ task dependency ⇢ fork/join dependency

Dataflow Runtime Performance Evaluation

- compare performance to Java’s fork/join framework
    - run micro benchmarks used by the fork/join paper
    - ÆMINIUM runtime about 35% slower

Dataflow Runtime “Atomic” Evaluation

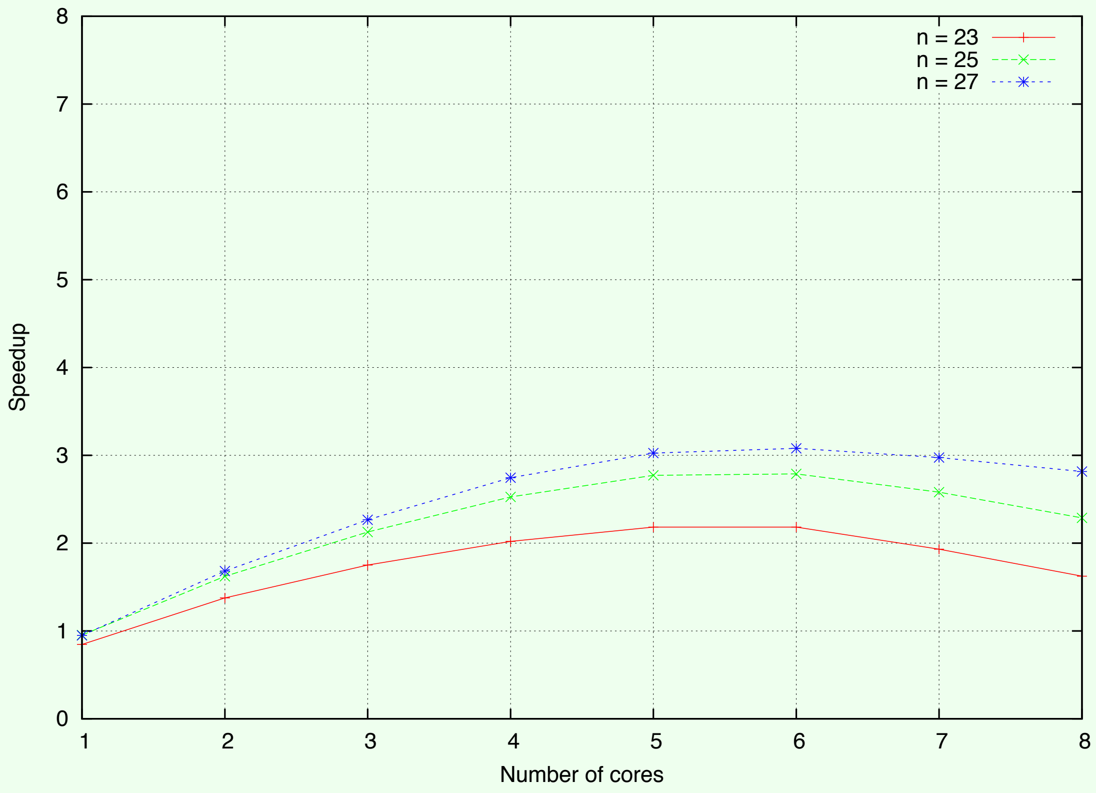
- compare worst, best and intermediate case
  - one global lock vs one lock per object
  - access single object vs multiple objects
  - read vs write
- the locking based implementation outperformed STM based implementation in almost all cases

Proof of Concept

- Master thesis of Manuel Mohr
- hand generated AST with type information
- each method call becomes a task
- showed principle feasibility

Proof of Concept

- performance improvements
- more optimize systems
- dynamic/static load balancing



Road ahead ...

Language Implementation

- implementing ÆMINIUM in Plaid
- Plaid has built-in support for permissions
- limited type checker for Plaid
  (lambda support is still missing)

Language Implementation

1. add ÆMINIUM to Plaid language/parser
2. extend Plaid typechecker with data groups
3. extend Plaid infrastructure to compute dataflow graph based on permission flow
4. extend Plaid code generator to produce parallel code

Approach

- 1st milestone
  - extend Plaid to compute permission flow and parallelize code (no data groups)
- 2nd milestone
  - extend Plaid with data groups
- Evaluate system

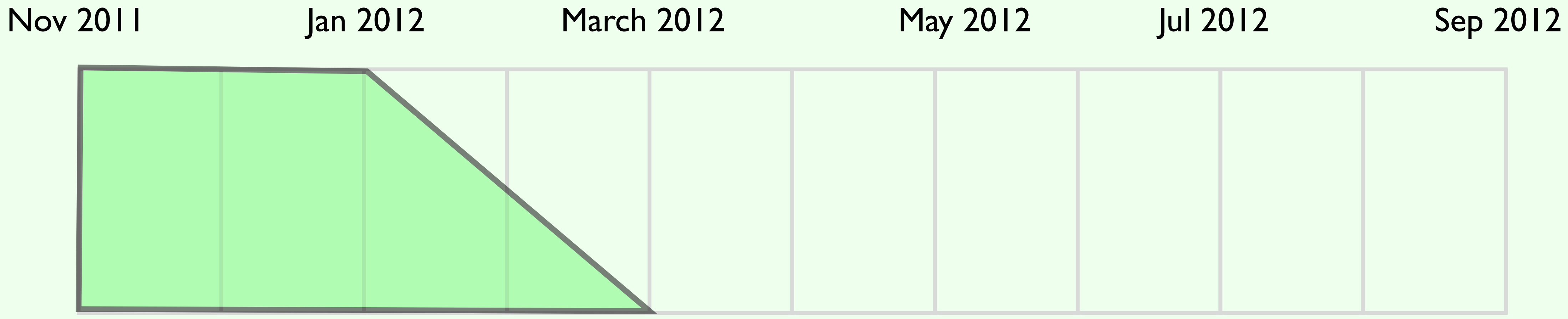
Evaluation

- conducting multiple case studies
  - evaluating performance (cf. **efficiency** hypothesis)
  - evaluating practicality (cf. **practical** hypothesis)

Evaluation

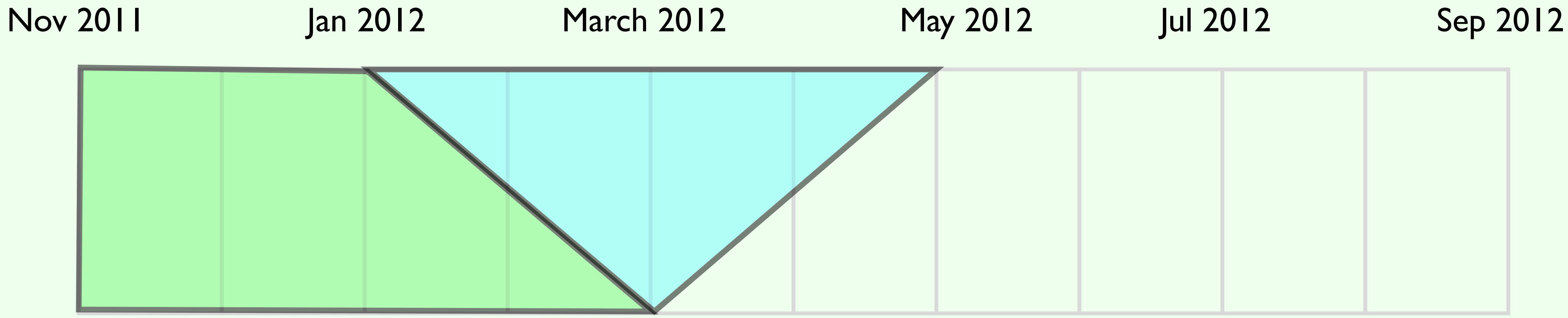
- selection of case studies
    - use applications with known parallel/ concurrency characteristics
    - use representative applications
        - existing real-world applications
        - existing benchmarks (SPLASH, SPEC, DaCapo, etc)
- rewrite applications in ÆMINIUM/Plaid

Time Line



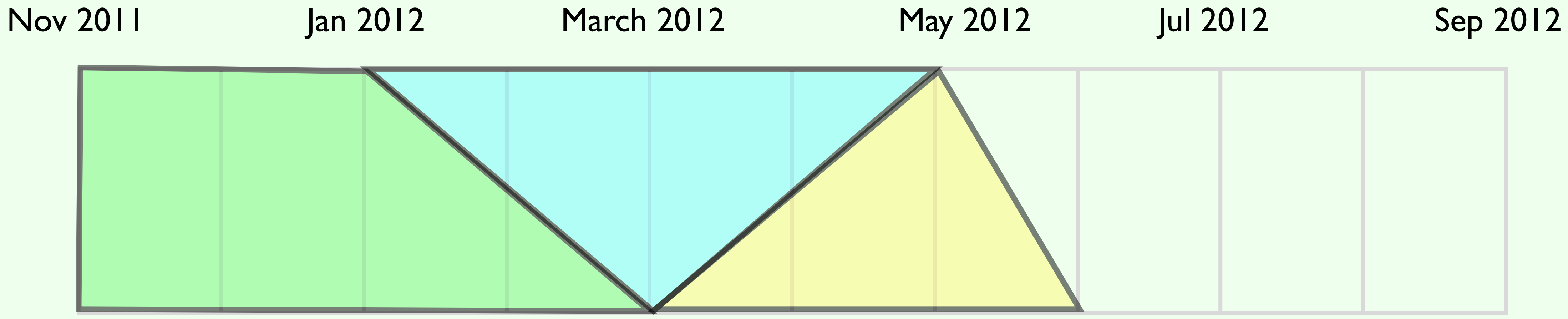
1st Milestone
permission only implementation

Time Line



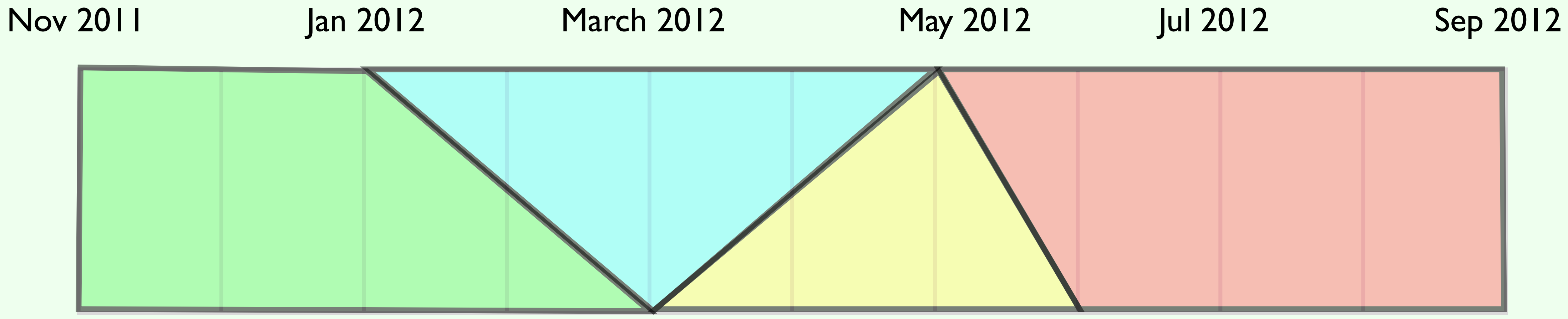
2nd Milestone
data group implementation

Time Line



Evaluation

Time Line



writing thesis

Risks

- Slow progress in Plaid
  - omit unnecessary features
  - parallelize/overlap work
  - 2 stage approach

Risks

- Granularity issues
  - implement optimization techniques (e.g., task merging, flattening, etc)
  - use dynamic load-balancing to avoid generation of "useless" tasks

Risks

- Lack of parallelism
  - no silver bullet
  - ensure that we do not pay extra in the case there is no parallelism

Thanks for the Attention!

Questions?