CONCURRENCY BY DEFAULT

Sven Stork†*
Paulo Marques* and Jonathan Aldrich†

†Carnegie Mellon University
*University of Coimbra

October 28, 2009
Motivation

Era of Concurrency is upon us!

- Concurrency is (almost) everywhere nowadays
- Major paradigm shift in technology caused by physical limitations
- **BUT** we still reason about **sequential ordering** when writing programs

**Experiment:**

- Can we use reasoning about **dependencies** instead of **ordering**?
- Lead this to **more concurrent** and **more correct** code?
Explicit/Implicit Concurrency Models

Explicit

- **default** in **mainstream** (object oriented) programming languages
- "manual" management of concurrent (e.g., threads, locks, ...)
- **complex** and **error prone** reasoning about ordering constraints
- concurrent issues **additionally** to sequential problems

Implicit

- tell the system **what to do** and **not how to do it**
- pure functional programming languages (**no state**) (no state)
- works well for certain domains (e.g., **data parallel** languages)
  - does not work for general purpose tasks (e.g., text editor)
Can you eat the cake and have it?

Object Oriented Programming Language

- State
- Object Abstraction

Implicit Concurrency Model

- Automatically extraction of concurrency information.

Additional Features

- Protection against common concurrency issues (e.g., race conditions).
Concurrency By Default

- Paradigm shift in technology
  \( \leadsto \) Paradigm shift in programming.

- Current Approach: \textit{sequential by default} ;
  - Think about \textit{sequential order} of statements.
  - Think about \textit{ordering constraints} of concurrent entities.

- New Approach: \textit{concurrent by default} ;
  - \textit{Sequentiality} must be \textit{explicit specified}.
  - Thinking about \textit{dependencies/requirements}.
  - Let the \textit{system} worry about \textit{correct execution order}.

\textit{Æminium} new programming language based on concurrency by default paradigm
Approach

Why are current approaches insufficient?

- **Problem**: lack of **exact dependency information** caused by **aliasing**

- **Remedy**: use **access permissions**
  1. provide **aliasing** and **data access** information
  2. calculate **more precise dependencies**
  3. infer **concurrency information** obeying dependencies (**dataflow graph**)
  4. check against **correct usage**
The Access Permissions we Use

**unique**
- there is only one reference to the object
- exclusive access
- no synchronization required

**immutable**
- there might be several alias reference to the object, but all of them are immutable
- the object cannot be modified through an immutable reference
- no synchronization required

**shared**
- there might be several alias reference to the object, but all of them are shared
- the object can be modified through an shared reference
- access to shared objects requires synchronization
The Access Permissions we Use

unique
- there is only one reference to the object
- exclusive access
- no synchronization required

immutable
- there might be several alias reference to the object, but all of them are immutable
- the object cannot be modified through an immutable reference
- no synchronization required

shared
- there might be several alias reference to the object, but all of them are shared
- the object can be modified through a shared reference
- access to shared objects requires synchronization
  synchronization ENFORCED
How to infer concurrency with permissions?

- **automatically splitting/joining** of permissions
  
e.g., unique \(\iff\) immutable \(\otimes\) immutable
  
e.g., unique \(\iff\) shared \(\otimes\) shared
  
e.g., shared \(\iff\) shared \(\otimes\) shared

- Use **linear logic** and **fractions** for management access permissions

- "**reverse**" this approach and **infer** concurrency from permission flow
  
  1. Infer **permission flow** base on **lexical order**
  2. **DEFINE** that **operations** can **run concurrently** iff they depend on:
     
     - immutable permissions \(\leadsto\) only read operations
     - shared permissions \(\leadsto\) access must synchronized
  3. **Generate** **dataflow graph**

- For **enforcing** additional dependencies, we propose the usage of **data groups** (see paper for more details)
Producer/Consumer Main Program

```c
void main() {
    Queue q = createQueue()
    producer(q)
    consumer(q)
    disposeQueue(q)
}
```
Producer/Consumer Main Program

```c
void main() {
    Queue q = createQueue()
    producer(q)
    consumer(q)
    disposeQueue(q)
}
```

**Question**

- Is there concurrency in this program?
- Note the missing semicolon
Producer/Consumer Permission Annotation

```java
class Queue {
    void push(Object o) : shared(this), unique(o) ⇒ shared(this)
    { atomic { ... } }

    Object pop() : shared(this) ⇒ shared(this), unique(result)
    { atomic { ... } }
}

Queue createQueue() : unit ⇒ unique(result)

void disposeQueue(Queue q) : unique(q) ⇒ unit

void producer(Queue q) : shared(q) ⇒ shared(q)
{ q.push(new Object()) ... }

void consumer(Queue q) : shared(q) ⇒ shared(q)
{ Object o = q.pop() ... }
```

Permission Information

Syntax:

**INPUT**

**OUTPUT**

- `createQueue` returns a unique permission
- `disposeQueue` consumes a unique permission
- `producer` and `consumer` shared queue

Synchronization only shared permission to receiver ⇒ synchronize

Transfer of Ownership

- `push` method consumes a unique permission
- `pop` returns a unique permission
Producer/Consumer Permission Annotation

Permission Information

- **Syntax:** \( INPUT \Rightarrow OUTPUT \)
  - createQueue returns a unique permission
  - disposeQueue consumes a unique permission
  - producer and consumer shared queue

```java
class Queue {
    void push(Object o) : shared(this), unique(o) {
        atomic {
            ... 
        }
    }

    Object pop() : shared(this) Z ⇒ shared(this), unique(result) {
        atomic {
            ... 
        }
    }
}

Queue createQueue() : unit Z ⇒ unique(result)

void disposeQueue(Queue q) : unique(q) ⇒ unit

void producer(Queue q) : shared(q) ⇒ shared(q)
{ q.push(new Object()) ... }

void consumer(Queue q) : shared(q) ⇒ shared(q)
{ Object o = q.pop() ... }
```
Producer/Consumer Permission Annotation

class Queue {
  void push(Object o) {
    atomic {
      ... 
    }
  }

  Object pop() {
    atomic {
      ... 
    }
  }

  Queue createQueue() : unit ⇒ unique(result)

  void disposeQueue(Queue q) : unique(q) ⇒ unit

  void producer(Queue q) : shared(q) ⇒ shared(q)
  { q.push(new Object()) ... }

  void consumer(Queue q) : shared(q) ⇒ shared(q)
  { Object o = q.pop() ... }

Permission Information

- **Syntax**: INPUT ⇒ OUTPUT
- createQueue returns a unique permission
- disposeQueue consumes a unique permission
- producer and consumer shared queue

Synchronization only shared permission to receiver → synchronize

Transfer of Ownership
- push method consumes a unique permission
- pop returns a unique permission
Producer/Consumer Permission Annotation

```java
class Queue {
    void push(Object o) {
        Z ⇒ shared(this), unique(o)
    }

    Object pop() {
        atomic {
            ...
        }
    }

    Queue createQueue() : unit ⇒ unique(result)

    void disposeQueue(Queue q) : unique(q) ⇒ unit

    void producer(Queue q) : shared(q) ⇒ shared(q)
    { q.push(new Object()) ... }

    void consumer(Queue q) : shared(q) ⇒ shared(q)
    { Object o = q.pop() ... }
}
```

Permission Information

- **Syntax:** `INPUT ⇒ OUTPUT`
- `createQueue` returns a `unique` permission
- `disposeQueue` consumes a `unique` permission
- `producer` and `consumer` shared queue
- Synchronization only shared permission to receiver
- Transfer of Ownership: `push` method consumes a `unique` permission, `pop` returns a `unique` permission
Producer/Consumer Permission Annotation

Permission Information

- **Syntax:** $INPUT \Rightarrow OUTPUT$
- `createQueue` returns a `unique` permission
- `disposeQueue` consumes a `unique` permission
- `producer` and `consumer` shared queue

```java
class Queue {
    void push(Object o) {
        shared(this), unique(o) Z \Rightarrow shared(this)
    }
    Object pop() {
        atomic {
            ...
        }
    atomic {
        ...
    }
}

Queue createQueue() : unit \Rightarrow unique(result)

void disposeQueue(Queue q) : unique(q) \Rightarrow unit

void producer(Queue q) : shared(q) \Rightarrow shared(q)
{ q.push(new Object()) ... }

void consumer(Queue q) : shared(q) \Rightarrow shared(q)
{ Object o = q.pop() ... }
```
Producer/Consumer Permission Annotation

class Queue {
    void push(Object o) : shared(this), unique(o) ⇒ shared(this)
    { atomic { ... } }

    Object pop() : shared(this) ⇒ shared(this), unique(result)
    { atomic { ... } }
}

Queue createQueue() : unit ⇒ unique(result)

void disposeQueue(Queue q) : unique(q) ⇒ unit

void producer(Queue q) {
    q.push(new Object() ... )
}

void consumer(Queue q) : shared(q) ⇒ shared(q)
{ Object o = q.pop() ... }

Synchronization

- only shared permission to receiver → synchronize
### Producer/Consumer Permission Annotation

```java
class Queue {
    void push(Object o) : shared(this), unique(o) \Rightarrow shared(this)
    {
    atomic { ... }
    }

    Object pop() : shared(this) \Rightarrow shared(this), unique(result)
    {
    atomic { ... }
    }
}

Queue createQueue() : unit \Rightarrow unique(result)

void disposeQueue(Queue q) : unique(q) \Rightarrow unit

void producer(Queue q) : shared(q) \Rightarrow shared(q)
    {
    q.push(new Object() ...)
    }

void consumer(Queue q) : shared(q) \Rightarrow shared(q)
    {
    Object o = q.pop() ...
    }
```

**Synchronization**

- only shared permission to receiver \(\rightarrow\) synchronize

**Permission Information**

- `createQueue` returns a unique permission
- `disposeQueue` consumes a unique permission
- `producer` and `consumer` shared queue

**Synchronization only shared permission to receiver**
Producer/Consumer Permission Annotation

class Queue {
    void push(Object o) : \textit{shared}(\texttt{this}), \textit{unique}(\texttt{o}) \Rightarrow \textit{shared}(\texttt{this})
    { atomic { ... } }

    Object pop() : \textit{shared}(\texttt{this}) \Rightarrow \textit{shared}(\texttt{this}), \textit{unique}(\texttt{result})
    { atomic { ... } }
}

Queue createQueue() : \texttt{unit} \Rightarrow \texttt{unique}(\texttt{result})

void disposeQueue(Queue q) : \texttt{unique}(\texttt{q}) \Rightarrow \texttt{unit}

void producer(Queue q) : \texttt{shared}(\texttt{q}) \Rightarrow \texttt{shared}(\texttt{q})
{ q.push(\texttt{new Object()}) ... }

void consumer(Queue q) : \texttt{shared}(\texttt{q}) \Rightarrow \texttt{shared}(\texttt{q})
{ Object o = q.pop() ... }
Producer/Consumer Permission Annotation

```java
class Queue {
    void push(Object o) : shared( this ), unique( o ) ⇒ shared(this)
    {
        atomic { ... }
    }
    Object pop() : shared( this ) ⇒ shared(this), unique( result )
    {
        atomic { ... }
    }
}
Queue createQueue() : unit ⇒ unique( result )
void disposeQueue(Queue q) : unique( q ) ⇒ unit
void producer(Queue q) : shared( q ) ⇒ shared(q)
    {
        q.push( new Object() ) ...
    }
void consumer(Queue q) : shared( q ) ⇒ shared(q)
    {
        Object o = q.pop() ...
    }
```

Transfer of Ownership

- push method consumes a unique permission
- pop returns a unique permission
Producer/Consumer Permission Annotation

```java
class Queue {
    void push(Object o) : shared( this ), unique( o ) \Rightarrow shared(this)
    {
        atomic { ... }
    }

    Object pop() : shared( this ) \Rightarrow shared(this), unique( result )
    {
        atomic { ... }
    }
}

Queue createQueue() : unit \Rightarrow unique( result )

void disposeQueue(Queue q) : unique( q ) \Rightarrow unit

void producer(Queue q) : shared( q ) \Rightarrow shared( q )
{
    q.push( new Object() ) ...
}

void consumer(Queue q) : shared( q ) \Rightarrow shared( q )
{
    Object o = q.pop() ...
}
```

Transfer of Ownership:
- push method consumes a unique permission
- pop returns a unique permission
Producer/Consumer Permission Annotation

class Queue {
    void push(Object o) : shared( this ), unique( o ) \Rightarrow shared(this)
    { atomic { ... } }

    Object pop() : shared( this ) \Rightarrow shared(this), unique( result )
    { atomic { ... } }
}

Queue createQueue() : unit \Rightarrow unique( result )

void disposeQueue(Queue q) : unique( q ) \Rightarrow unit

void producer(Queue q) : shared( q ) \Rightarrow shared(q)
{ q.push( new Object() ) ... }

void consumer(Queue q) : shared( q ) \Rightarrow shared(q)
{ Object o = q.pop() ... }

Transfer of Ownership

- push method consumes a unique permission
- pop returns a unique permission
Producer/Consumer Permission Flow

```java
void main()
    Queue q = createQueue()  //unit ⇔ unique(q)

producer(q)  //shared(q) ⇔ shared(q)

consumer(q)  //shared(q) ⇔ shared(q)

disposeQueue(q)  //unique(q) ⇔ unit
}
```
Producer/Consumer Permission Flow

```java
void main()
    Queue q = createQueue()  // unit ⇒ unique(q)

producer(q)  // shared(q) ⇒ shared(q)

consumer(q)  // shared(q) ⇒ shared(q)

disposeQueue(q)  // unique(q) ⇒ unit
```

Motivation

Approach

Related Work

Conclusion
Producer/Consumer Permission Flow

```c
void main()
    Queue q = createQueue()  //unit ⇒ unique(q)

⇒ producer(q)              //shared(q) ⇒ shared(q)

consumer(q)                 //shared(q) ⇒ shared(q)

disposeQueue(q)              //unique(q) ⇒ unit
```

---

Producer/Consumer Permission Flow

```java
void main()
    Queue q = createQueue() //unit ⇒ unique(q)

⇒ producer(q) //shared(q) ⇒ shared(q)

consumer(q) //shared(q) ⇒ shared(q)

disposeQueue(q) //unique(q) ⇒ unit
```

Producer/Consumer Permission Flow

```c
void main()
{
    Queue q = createQueue(); // unit \Rightarrow unique(q)

    producer(q) // shared(q) \Rightarrow shared(q)
    consumer(q)  // shared(q) \Rightarrow shared(q)
    disposeQueue(q) // unique(q) \Rightarrow unit
}
```
Producer/Consumer Permission Flow

```plaintext
void main()
{
    Queue q = createQueue()  //unit ⇒ unique(q)
    split
    shared(q) ⇒ producer(q)  //shared(q) ⇒ shared(q)
    consumer(q)               //shared(q) ⇒ shared(q)
    disposeQueue(q)           //unique(q) ⇒ unit
}
```
Producer/Consumer Permission Flow

```java
void main()
{
    Queue q = createQueue(); //unit ⇒ unique(q)

    split
    shared(q)

    producer(q) //shared(q) ⇒ shared(q)

    ⇒ consumer(q) //shared(q) ⇒ shared(q)

    disposeQueue(q) //unique(q) ⇒ unit
}
```
void main()

Queue q = createQueue() //unit \Rightarrow unique(q)

producer(q) //shared(q) \Rightarrow shared(q)

consumer(q) //shared(q) \Rightarrow shared(q)

disposeQueue(q) //unique(q) \Rightarrow unit
Producer/Consumer Permission Flow

```java
void main()
{
    Queue q = createQueue();  // unit ⇒ unique(q)

    split
    {
        unique(q)

        split
        {
            shared(q)
            {
                producer(q)  // shared(q) ⇒ shared(q)

                split
                {
                    shared(q)
                    {
                        consumer(q)  // shared(q) ⇒ shared(q)
                    }
                }
            }
        }
    }

    disposeQueue(q)  // unique(q) ⇒ unit
}
```
Producer/Consumer Permission Flow

```java
void main()
    Queue q = createQueue() //unit ⇒ unique(q)
    split
    shared(q)
    producer(q) //shared(q) ⇒ shared(q)
    split
    shared(q)
    consumer(q) //shared(q) ⇒ shared(q)
    ⇒ disposeQueue(q) //unique(q) ⇒ unit
```

```c
unique(q) = Queue q = createQueue() //unit ⇒ unique(q)
```

```c
split
shared(q)
producer(q) //shared(q) ⇒ shared(q)
```

```c
split
shared(q)
consumer(q) //shared(q) ⇒ shared(q)
```

```c
⇒ disposeQueue(q) //unique(q) ⇒ unit
```
void main()

Queue q = createQueue()  // unit ⇒ unique(q)

split

shared(q)

producer(q)  // shared(q) ⇒ shared(q)

split

shared(q)

consumer(q)  // shared(q) ⇒ shared(q)

join

unique(q)

⇒ disposeQueue(q)  // unique(q) ⇒ unit

}
Producer/Consumer Permission Flow

```java
void main()
{
  Queue q = createQueue(); //unit ⇒ unique(q)
  split
  shared(q)
  shared(q)
  producer(q) //shared(q) ⇒ shared(q)
  split
  shared(q)
  shared(q)
  consumer(q) //shared(q) ⇒ shared(q)
  join
  unique(q)
  disposeQueue(q) //unique(q) ⇒ unit
}
producer/consumer permission dataflow graph

1. `createQueue` provides
2. split unique into shared
3. producer/consumer executed concurrently
4. access to shared state must be protected
5. after completion producer/consumer return shared permission
6. recover unique permissions
7. execute disposeQueue

User Aids
IDE can provide this or similar visualization to the user

Concurrency By Default
Producer/Consumer Permission Dataflow Graph

**createQueue**

1. createQueue provides
2. split unique into shared
3. producer/consumer executed concurrently
4. access to shared state must be protected
5. after completion producer/consumer return shared permission
6. recover unique permissions
7. execute disposeQueue
Producer/Consumer Permission Dataflow Graph

1. `createQueue` provides
2. `split` unique into shared
3. Producer/consumer executed concurrently
4. Access to shared state must be protected
5. After completion producer/consumer return shared permission
6. Recover unique permissions
7. Execute disposeQueue

User Aids
IDE can provide this or similar visualization to the user.
Producer/Consumer Permission Dataflow Graph

1. `createQueue` provides unique permissions.
2. `split` unique into shared.
3. Producer/consumer executed concurrently.
4. Access to shared state must be protected.
5. After completion, producer/consumer return shared permission.
6. Recover unique permissions.
7. Execute disposeQueue.
### Producer/Consumer Permission Dataflow Graph

1. **createQueue** provides
2. **split** unique into shared
3. **producer/consumer** executed concurrently
4. **access** to shared state must be protected
5. after completion **producer/consumer** return shared permission
6. **recover** unique permissions
7. execute **disposeQueue**
Producer/Consumer Permission Dataflow Graph

1. `createQueue` provides
2. `split` unique into shared
3. producer/consumer executed concurrently
4. access to shared state must be protected
5. after completion producer/consumer return shared permission
6. recover unique permissions
7. execute `disposeQueue`
Producer/Consumer Permission Dataflow Graph

1. `createQueue` provides
2. `split` unique into shared
3. `producer/consumer` executed concurrently
4. `access` to shared state must be protected
5. After completion, `producer/consumer` return shared permission
6. `recover` unique permissions
7. `execute disposeQueue`
Producer/Consumer Permission Dataflow Graph

1. `createQueue` provides
2. `split` unique into shared
3. producer/consumer executed concurrently
4. access to shared state must be protected
5. after completion producer/consumer return shared permission
6. recover unique permissions
7. execute `disposeQueue`
Producer/Consumer Permission Dataflow Graph

1. `createQueue` provides unique(q)
2. `split` unique into shared(q)
3. `producer`/`consumer` executed concurrently
4. access to shared state must be protected
5. after completion `producer`/`consumer` return `shared` permission
6. `recover` unique permissions
7. `execute` `disposeQueue`

User Aids

IDE can provide this or similar **visualization** to the user
Related Work With Permissions

Boyland
- Verification of concurrent programs
  (simple system with just unique/immutable) [SAS’03]
- Verification of correct lock usage
  (nested permissions) [ASWEC’09]

Bierhoff
- Verification of API protocol conformance
  [OOPSLA’07]

Beckman
- Verification of correct usage of atomic blocks
  [ECOOP’08]
- Optimizations of Software Transactional Memory system [IWACO’09]

Terauchi
- Verification of correct usage of locks [PLDI’08]
Concurrent Programming Languages

Fortress
- concurrent by default semantics for loops and tuples
- no check for correct synchronization

Axum
- programs as dataflow graph
- manual generation of graph

ML, etc.
- exact dependency information
- no state

NESL, etc.
- implicit concurrency model
- largely limited to domain specific areas
Conclusion

ÆMINIUM
IN A NUTSHELL
A Desktop Quick Reference

O’REILLY

Sven Stork
Conclusion

- New programming paradigm: **Concurrency by Default**

- **ÆMINIUM** a new programming language based on concurrency by default
  - use access permissions and data groups to specify dependencies and extract concurrency information

- Future Work:
  - Complete formal system.
  - Implement runtime system.
  - Performance evaluation and user studies
  - Merge with Plaid/Typestate (see talk tomorrow 3:50pm)
Thanks for the attention!

Questions ????
What Does ÆMINIUM Mean?

ÆMINIUM was the ancient roman city on which Coimbra was established.
Data Groups

- associate all shared objects to one data groups
- data groups have 3 possible states
  - atomic
  - concurrent
  - protected
- programmer must manually split and join
  - the split block converts atomic → concurrent groups
  - the atomic block converts concurrent → protected groups