Concurrency by Default

Sven Stork
svens@cs.cmu.edu

May 8, 2009

Outline

Motivation

Motivation

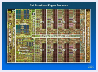
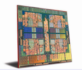
Why are we looking into concurrent programming ?

- concurrent programming is the future (no choice)
  - "The free lunch is over" (Herb Sutter)
  - several areas move towards concurrency (e.g., embedded systems, HPC, ...)
- concurrent programming was not mainstream in the last decades
  - there is no (good) support for parallel programming in current programming languages
  - only domain specific areas (e.g., HPC) have solutions for concurrent programming
  - concepts and approaches of domain specific areas are most of the time no suitable for general purpose programming
- concurrent programming is important and hard
  - all problems of sequential programming
  - + concurrent problems (e.g., race-conditions, deadlocks, ...)

Motivation

What can we learn ?

- Large scale concurrency is upon us
- Different kind of hardware support for concurrency


- Many different approaches to write efficient code

Real World Example

- Currently TAing 'High Performance Computing' class
- 1^{st} assignment was NVIDIA/CUDA
  - Optimal approach, use 1 one thread per matrix cell
- 2^{nd} assignment was Pthreads
  - Student blindly applied CUDA approach and create 1 thread computation element
  - 'Obviously' inefficient approach for OS threads

Motivation

How to program those systems ?

- **explicit**
  - user manually manage concurrency via low-level primitives (e.g., threads, locks, semaphores, ...)
  - user has to reason about
    - possible execution paths and correct synchronization
    - correct granularity of concurrency
    - how to deal with locality
- **implicit**
  - user specifies what should be computed and what the dependencies are
  - the runtime will handle
    - possible execution paths and correct synchronization
    - correct granularity of concurrency
    - how to deal with locality

Motivation

Our hypothesis

- **Implicit concurrency** is the more general and desirable approach.

Motivation

Is implicit concurrency a silver bullet ?

- NO

Motivation

Is implicit concurrency a silver bullet ?

- NO
- but in comparable situation to garbage collection versus manual memory management
  - automatic management solves many problems (e.g., dangling pointers)
  - automatic management reduces learning curve and increases productivity
  - there are cases where manual control is required (not the case for most applications)

Motivation

Is implicit concurrency a silver bullet ?

- NO
- but in comparable situation to garbage collection versus manual memory management
  - automatic management solves many problems (e.g., dangling pointers)
  - automatic management reduces learning curve and increases productivity
  - there are cases where manual control is required (not the case for most applications)
- and comparable to high-level versus assembly code
  - high-level abstracts from low-level details (write once, run everywhere)
  - high-level abstractions reduce learning curve and increases productivity
  - there are cases where low-level control is required (not the case for most applications)

State of the Art

What do we have currently ?

- programming languages with implicit parallelism
  - NESL, ZPL, ...
  - works well if it comes to data parallelism
  - have limitations when it comes to general purpose programs
- programming languages with explicit parallelism
  - Java, Erlang, Cilk, ...
  - requires explicit specification of concurrency

State of the Art

How about automatically parallisation ?

- works reasonably well for micro-parallelism
  - e.g., using vector units for computation

State of the Art

How about automatically parallisation ?

- works reasonably well for micro-parallelism
  - e.g., using vector units for computation
- works somewhat well regular problems
  - e.g., using OpenMP for loops, blocks, ...

State of the Art

How about automatically parallisation ?

- works reasonably well for micro-parallelism
  - e.g., using vector units for computation
- works somewhat well regular problems
  - e.g., using OpenMP for loops, blocks, ...
- works poorly for irregular problems
  - e.g., how to automatically parallelize a web server ?

State of the Art

How about automatically parallisation ?

- works reasonably well for micro-parallelism
  - e.g., using vector units for computation
- works somewhat well regular problems
  - e.g., using OpenMP for loops, blocks, ...
- works poorly for irregular problems
  - e.g., how to automatically parallelize a web server ?

Why is it so hard ?

- hard to extract concurrency with current programming languages:
  - code says how and not what to do
  - aliasing problems

Objective

What we intend to do ?

- reverse the situation
- everything is concurrent by default
- use access permissions to specify design intent
- use access permissions to extract data dependencies
- programmer only specifies data dependencies that cannot be inferred

Concurrency by Default

Automatic Concurrency

Explicit Concurrency

- So far users still need to explicit **think** and **code** for concurrency

Automatic Concurrency

Explicit Concurrency

- So far users still need to explicit **think** and **code** for concurrency

Solution

- **Access Permissions** specify how data is accesses or modified
  - **automatically splitting/joining**
    e.g., unique $\Lleftarrow\!\!\Rrightarrow$ immutable $\otimes$ immutable
    e.g., unique $\Lleftarrow\!\!\Rrightarrow$ shared $\otimes$ shared
  - use linear logic for management access permissions (e.g., $\multimap$)
- reverse this approach and infer which operations can be executed concurrently
- use these data dependencies and lexical order to create dataflow graphs

Automatic Concurrency

What kind of Access Permissions do we use ?

Type	Interpretation
Unique	• owner has exclusive access to the object • all previous read/write operations need to be finished
Immutable	• owner has only read permissions • the object is currently only referenced by immutable permissions • multiple reads can take palce concurrently • all previous write operations need to be finished
Shared	• multiple owners that have read/write permissions • owner needs to be inside an atomic-block to access the referenced object (shared → unique) • multiple shared accesses can take place concurrently

Automatic Concurrency

What kind of Access Permissions do we use ?

Atomic Block

- scoped block; body is executed in transactional way
- either everything executes or nothing
- creates the illusion that body has exclusive access to resources

```
atomic {
    ...
}
```

Type	Interpretation
Unique	…nished
Immuta…	…mutable … …ed
Shared	…ns … …cess the … referenced object (shared → unique) • multiple shared accesses can take place concurrently

Automatic Concurrency

What kind of Access Permissions do we use ?

Type	Interpretation
Unique	• owner has exclusive access to the object • all previous read/write operations need to be finished
Immutable	• owner has only read permissions • the object is currently only referenced by immutable permissions • multiple reads can take palce concurrently • all previous write operations need to be finished
Shared	• multiple owners that have read/write permissions • owner needs to be inside an atomic-block to access the referenced object (shared → unique) • multiple shared accesses can take place concurrently

1^{st} Example : Unique/Immutable

Automatic Concurrency

Program

```
void main() {
  Collection c = readData()
  printCollection(c)
  Statistics s = compStats(c)
  Dependencies d = compDeps(c)
  removeDuplicates(c)
  printCollection(c)
}
```

Automatic Concurrency

Program

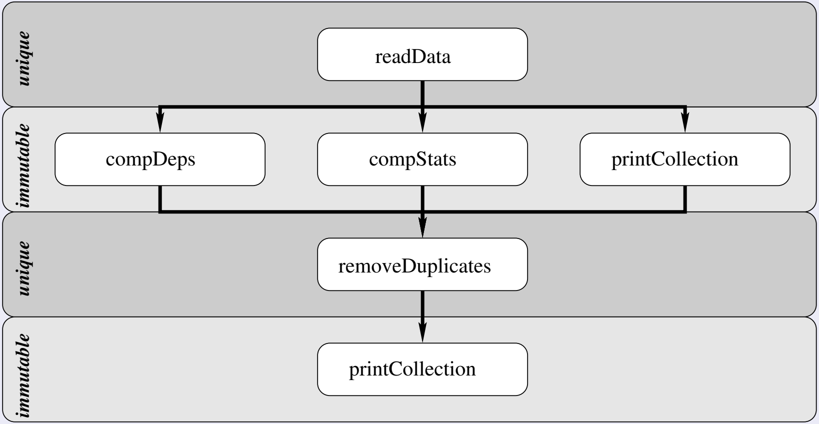
```
void main() {
  Collection c = readData()
  printCollection(c)
  Statistics s = compStats(c)
  Dependencies d = compDeps(c)
  removeDuplicates(c)
  printCollection(c)
}
```

Design Intent

- read data, non-modifying operations on data to extract information, modify data
- we would like to peform as many operations as possible concurrent

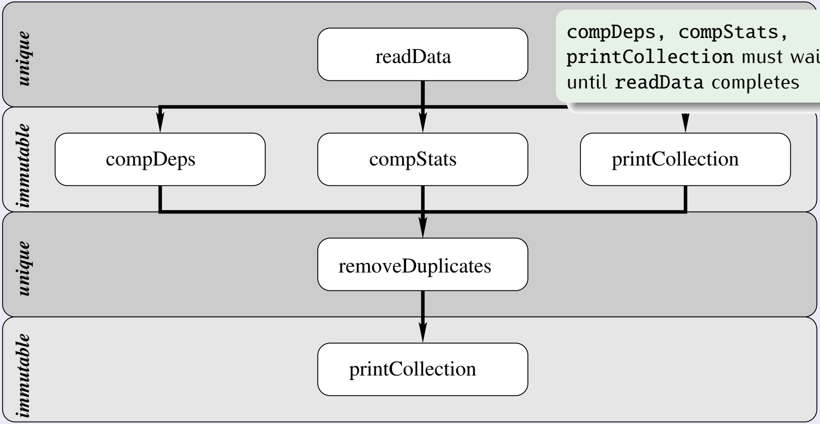
Automatic Concurrency

Dependency Graph



Automatic Concurrency

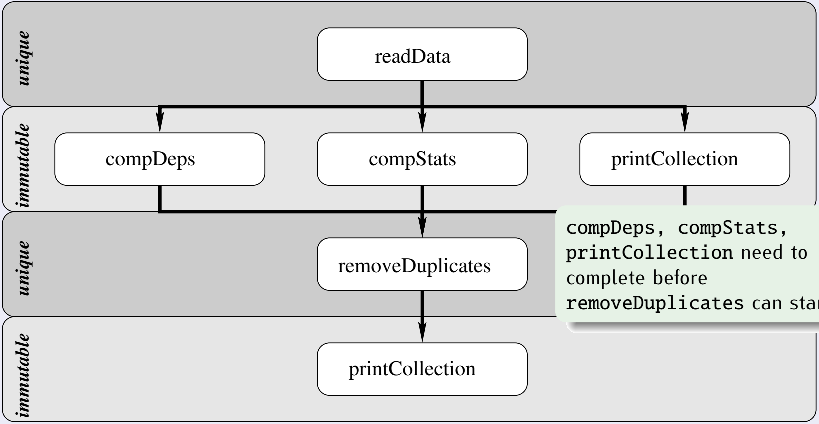
Dependency Graph



`compDeps`, `compStats`, `printCollection` must wait until `readData` completes

Automatic Concurrency

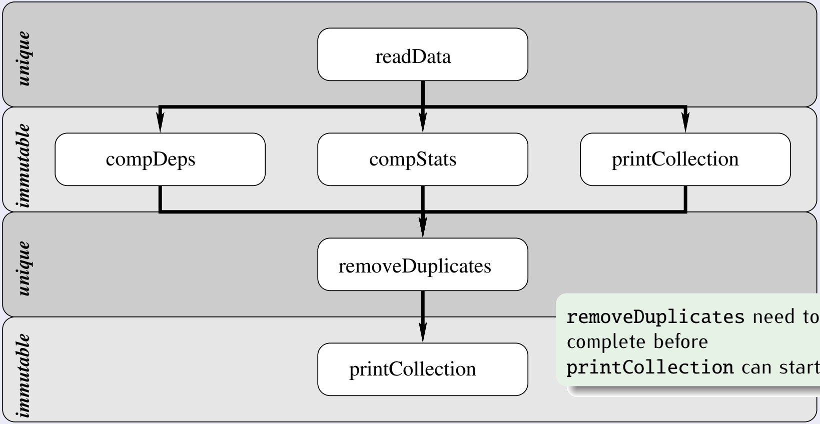
Dependency Graph



`compDeps`, `compStats`, `printCollection` need to complete before `removeDuplicates` can start

Automatic Concurrency

Dependency Graph



`removeDuplicates` need to complete before `printCollection` can start

Automatic concurrency

Interfaces

```
class Collection { ... }
class Dependencies { ... }
class Statistics { ... }

Collection readData()
 : unit —o unique(result)

void removeDuplicates(Collection c)
 : unique(c) —o unique(c)

void printCollection(Collection c)
 : immutable(c) —o immutable(c)

Dependencies compDeps(Connection c)
 : immutable(c) —o immutable(c),unique(result)

Statistics compStats(Connection c)
 : immutable(c) —o immutable(c),unique(result)
```

Automatic concurrency

Interfaces

```
class Collection { ... }
class Dependencies { ... }
class Statistics { ... }

Collection readData()
 : unit -o unique(result)

void removeDuplicates(Collection c)
 : unique(c) -o unique(c)

void printCollection(Collection c)
 : immutable(c) -o immutable(c)

Dependencies compDeps(Connection c)
 : immutable(c) -o immutable(c),unique(result)

Statistics compStats(Connection c)
 : immutable(c) -o immutable(c),unique(result)
```

`readData`, `removeDuplicates`
exclusive access to collection

Automatic concurrency

Interfaces

```
class Collection { ... }
class Dependencies { ... }
class Statistics { ... }

Collection readData()
 : unit -o unique(result)

void removeDuplicates(Collection c)
 : unique(c) -o unique(c)

void printCollection(Collection c)
 : immutable(c) -o immutable(c)

Dependencies compDeps(Connection c)
 : immutable(c) -o immutable(c),unique(result)

Statistics compStats(Connection c)
 : immutable(c) -o immutable(c),unique(result)
```

`compDeps`, `compStats`, `printCollection` **readonly access** to collection

2^{nd} Example : Unique/Immutable/Shared

Automatic Concurrency

Program

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

Automatic Concurrency

Program

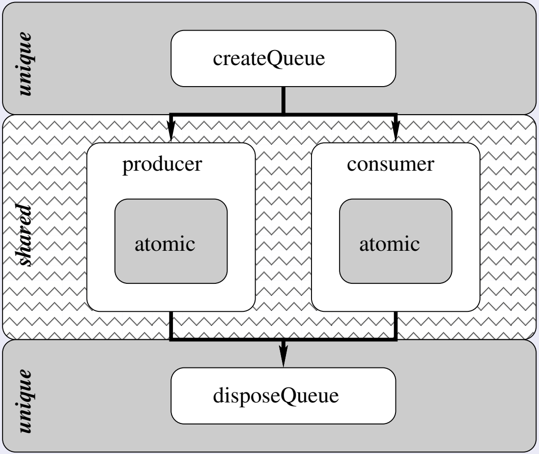
```
void main() {
    Queue q = createQueue()
    producer(q)
    consumer(q)
    disposeQueue(q)
}
```

Design Intent

- create queue, run producer/consumer concurrently, destroy queue
- consumer/producer must run concurrently to avoid a possible <u>deadlock</u> and allow <u>pipelining</u>

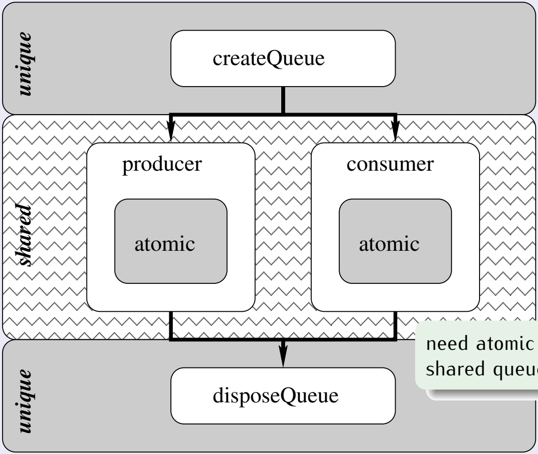
Automatic Concurrency

Dependency Graph



Automatic Concurrency

Dependency Graph



need atomic block to access shared queue

Automatic Concurrency

Interfaces

```
class Queue {
    void push(Object o)
      : unique(this), shared(o) −o unique(this)

    Object pop()
      : unique(this) −o unique(this), shared(result)
}

Queue createQueue() : unit −o unique(result)

void disposeQueue(Queue q) : unique(q) −o unit

void producer(Queue q) : shared(q) −o shared(q)
{
    atomic { q.add(...) }
}

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

Automatic Concurrency

Interfaces

```
class Queue {
    void push(Object o)
      : unique(this), shared(o) —o unique(this)

    Object pop()
      : unique(this) —o unique(this), shared(result)
}

Queue createQueue() : unit —o unique(result)

void disposeQueue(Queue q) : unique(q) —o unit

void producer(Queue q) : shared(q) —o shared(q)
{
    atomic { q.add(...) }
}

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

modifying the queue requires **exclusive access**

Automatic Concurrency

Interfaces

```
class Queue {
    void push(Object o)
      : unique(this), shared(o) -o unique(this)

    Object pop()
      : unique(this) -o unique(this), shared(result)
}

Queue createQueue() : unit -o unique(result)

void disposeQueue(Queue q) : unique(q) -o unit

void producer(Queue q) : shared(q) -o shared(q)
{
    atomic { q.add(...) }
}

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

creating/destroying requires **exclusive access**

Automatic Concurrency

Interfaces

```
class Queue {
    void push(Object o)
      : unique(this), shared(o) -o unique(this)

    Object pop()
      : unique(this) -o unique(this), shared(result)
}

Queue createQueue() : unit -o unique(result)

void disposeQueue(Queue q) : unique(q) -o unit

void producer(Queue q) : shared(q) -o shared(q)
{
    atomic { q.add(...) }
}

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

producer/consumer can run concurrently

3^{rd} Example : Unique/Immutable/Shared

Automatic Concurrency

Program

```
void main() {
    Subject sub = new Subject()
    Observer obs1 = new Observer(sub)
    Observer obs2 = new Observer(sub)
    update(sub)
    update(sub)
}
```

Automatic Concurrency

Program

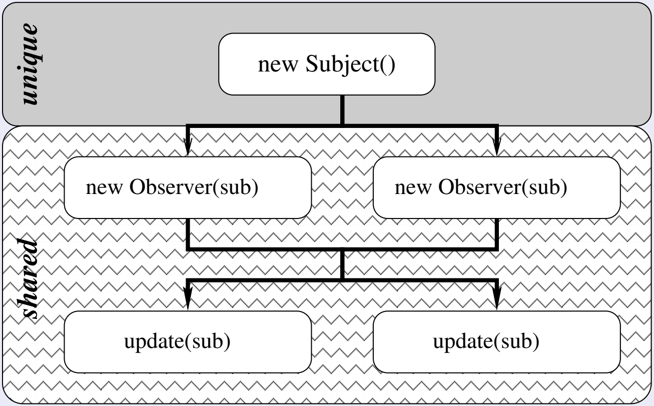
```
void main() {
    Subject sub = new Subject()
    Observer obs1 = new Observer(sub)
    Observer obs2 = new Observer(sub)
    update(sub)
    update(sub)
}
```

Design Intent

- observers should be created/subscribed in parallel
- updates should be performed concurrently
- observers need to attached before updates can be executed

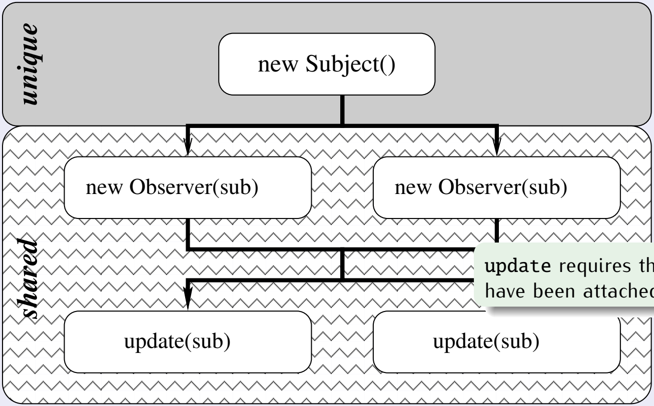
Automatic Concurrency

Dependency Graph



Automatic Concurrency

Dependency Graph



`update` requires that observers have been attached

Automatic Concurrency

Interfaces

```
class Subject {
  void add(Observer o)
    : shared(this), shared(o) -o shared(this)

  void update()
    : shared(this) -o shared(this)
}

class Observer {
  Observer(Subject s)
   : shared(s) -o shared(s), shared(result)
  { s.add(this); }

  void notify(Subject s)
   : shared(this), shared(s) -o shared(this), shared(s)
}

void update (Subject s) : shared(s) -o shared(s)
{
  s.update();
}
```

Automatic Concurrency

Interfaces

```
class Subject {
  void add(Observer o)
    : shared(this), shared(o) -o shared(this)

  void update()
    : shared(this) -o shared(this)
}

class Observer {
  Observer(Subject s)
   : shared(s) -o shared(s), shared(result)
  { s.add(this); }

  void notify(Subject s)
   : shared(this), shared(s) -o shared(this), shared(s)
}

void update (Subject s) : shared(s) -o shared(s)
{
  s.update();
}
```

subject shared amongst multiple entities → encourage sharing

Automatic Concurrency

Interfaces

```
class Subject {
  void add(Observer o)
    : shared(this), shared(o) -o shared(this)

  void update()
    : shared(this) -o shared(this)
}

class Observer {
  Observer(Subject s)
   : shared(s) -o shared(s), shared(result)
  { s.add(this); }

  void notify(Subject s)
   : shared(this), shared(s) -o shared(this), shared(s)
}

void update (Subject s) : shared(s) -o shared(s)
{
  s.update();
}
```

subscriptions should be performed concurrently

Automatic Concurrency

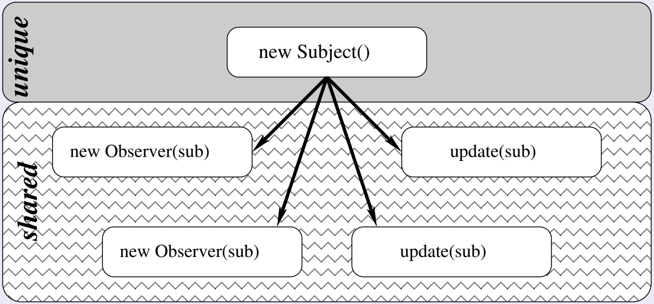
Interfaces

```
class Subject {
  void add(Observer o)
    : shared(this), shared(o) -o shared(this)

  void update()
    : shared(this) -o shared(this)
}

class Observer {
  Observer(Subject s)
   : shared(s) -o shared(s), shared(result)
  { s.add(this); }

  void notify(Subject s)
   : shared(this), shared(s) -o shared(this), shared(s)
}

void update (Subject s) : shared(s) -o shared(s)
{
  s.update();
}
```

updates should be performed concurrently

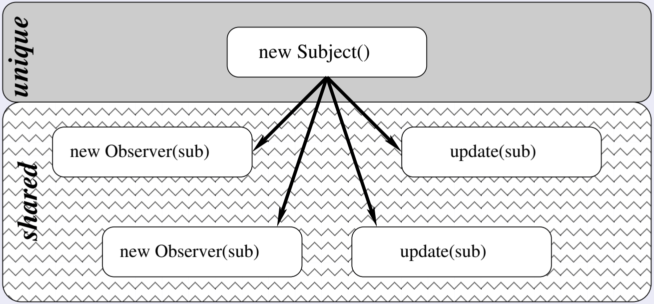
Automatic Concurrency

Dependency Graph



Automatic Concurrency

Dependency Graph



there is **no** data dependency between updates and subscriptions

Automatic Concurrency

Problem

- `update(sub)` and `new Observer(sub)` could run concurrently → **race-condition**
    - `Observer` might miss notify call (BAD)
    - `this` escapes constructor → `Observer` might be notified while not completely constructed (WORSE)

Automatic Concurrency

Problem

- `update(sub)` and `new Observer(sub)` could run concurrently → **race-condition**
  - `Observer` might miss notify call (BAD)
  - `this` escapes constructor → `Observer` might be notified while not completely constructed (WORSE)

Solution

- Allow the user to specify **application level dependencies**
- use **data groups** (Leino) to group objects

Automatic Concurrency

Data Groups

- every object is associated with exactly one data group
- there are 2 kind of permissions to data groups
  - **atomic** $\hat{=}$ unique
  - **concurrent** $\hat{=}$ shared
- data group permissions must **manually** be split/joined

Automatic Concurrency

Program

```
void main() {



    Subject sub = new Subject()
    Observer obs1 = new Observer(sub)
    Observer obs2 = new Observer(sub)



    update(sub)
    update(sub)

}
```

Automatic Concurrency

Program

```
void main() {



    Subject sub = new Subject()
    Observer obs1 = new Observer(sub)
    Observer obs2 = new Observer(sub)



    update(sub)
    update(sub)

}
```

`update` depends on the fact that observers are attached to the subject

Automatic Concurrency

Program

```
void main() {
    group<sg>


    Subject sub = new Subject()
    Observer obs1 = new Observer(sub)
    Observer obs2 = new Observer(sub)




    update(sub)
    update(sub)

}
```

add new data group

Automatic Concurrency

Program

```
void main() {
    group<sg>

    split(sg) {
        Subject sub = new<sg> Subject()
        Observer obs1 = new<sg> Observer(sub)
        Observer obs2 = new<sg> Observer(sub)
    }


    update(sub)
    update(sub)

}
```

- wrap dependent statements in split block
- associate objects with data group

Automatic Concurrency

Program

```
void main() {
    group<sg>

    split(sg) {
        Subject sub = new<sg> Subject()
        Observer obs1 = new<sg> Observer(sub)
        Observer obs2 = new<sg> Observer(sub)
    }

    split(sg) {
        update(sub)
        update(sub)
    }
}
```

wrap **update** inside split block

Automatic Concurrency

Program

```
void main() {
    group<sg>

    split(sg) {
        Subject sub = new<sg> Subject()
        Observer obs1 = new<sg> Observer(sub)
        Observer obs2 = new<sg> Observer(sub)
    }

    split(sg) {
        update(sub)
        update(sub)
    }
}
```

```
group<sg>
// atomic(sg)
```

Automatic Concurrency

Program

```
void main() {
    group<sg>

    split(sg) {
        Subject sub = new<sg> Subject()
        Observer obs1 = new<sg> Observer(sub)
        Observer obs2 = new<sg> Observer(sub)
    }

    split(sg) {
        update(sub)
        update(sub)
    }
}
```

```
// atomic(sg)
split(sg) {
    // concurrent(sg)
    Subject sub = new<sg> Subject()
    Observer obs1 = new<sg> Observer(sub)
    Observer obs2 = new<sg> Observer(sub);
}
// atomic(sg)
```

Automatic Concurrency

Program

```
void main() {
    group<sg>

    split(sg) {
        Subject sub = new<sg> Subject()
        Observer obs1 = new<sg> Observer(sub)
        Observer obs2 = new<sg> Observer(sub)
    }

    split(sg) {
        update(sub)
        update(sub)
    }
}
```

```
// atomic(sg)
split(sg) {
    // concurrent(sg)
    update(sub)
    update(sub)
}
// atomic(sg)
```

Automatic Concurrency

split statement

- use scoped `split` block to split atomic → concurrent permission

```
split (atomic DataGroup dg) {
    // provides arbitrary number of
    // concurrent permissions for dg
}
```

Automatic Concurrency

split statement

- use scoped `split` block to split atomic → concurrent permission

```
split (atomic DataGroup dg) {
    // provides arbitrary number of
    // concurrent permissions for dg
}
```

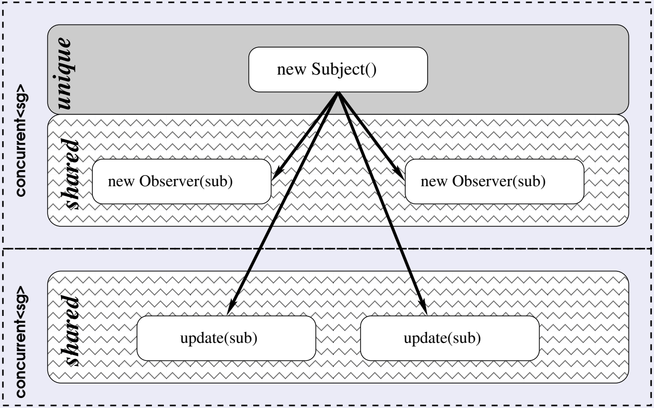
Extended Atomic Block

- extend atomic-block to refer to access data group(s)

```
atomic (concurrent DataGroup dg) {
    /* allows modifying access to object dg */
}
```

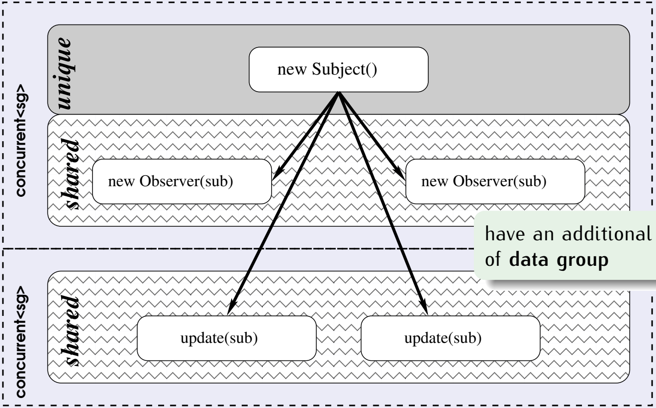
Automatic Concurrency

Dependency Graph



Automatic Concurrency

Dependency Graph



have an additional **dependency** of **data group**

Automatic Concurrency

Data Groups : Remarks

- if no data group is specified, a default data group is assumed (i.e., `world`)
- by default runs the whole program inside a `split(world)` block
- the more effort the user invest in specification, the better the results will be

Automatic Concurrency

Data Groups : Remarks

- if no data group is specified, a default data group is assumed (i.e., `world`)
- by default runs the whole program inside a `split(world)` block
- the more effort the user invest in specification, the better the results will be

World Split

```
split(world) {
    void main() { ... }
}
```

Automatic Concurrency

Data Groups : Remarks

- if no data group is specified, a default data group is assumed (i.e., `world`)
- by default runs the whole program inside a `split(world)` block
- the more effort the user invest in specification, the better the results will be

Data Groups: More Attributes

- data groups can be used to model ownership (Clark)
  - improves locality
  - allows better description of design intent
  - allows to reduce aliasing
- extended atomic-block allows possibility to optimize TM ?

Open Issues

Open Issues

Language

- Interoperability with legacy code (e.g. Java)
  - Scala has a nice approach
  - How to deal with code that has no annotations ?
- How much information needs to be preserved ?
  - Does the runtime needs more than just the dependency information ?
- Do we provide enough opportunity for concurrency ?
- Does the language encourage a concurrent programming style ?

Open Issues

Runtime

- Represent the data-dependencies after compilation ?
- How to implement the runtime in an efficient way ?
- Should the granularity be selected by the runtime ?

Conclusion

Conclusion

- Revolution vs Evolution
  - no incremental improvements based on existing approaches → solve solves near term future
  - aim for mid-long term future
  - fundamentally change the way we think and write programs
  - design and specify programs rather than code them
- new language that
  - encourage the user to specify his design intent
  - only minimal data dependencies specification by user (mainly inferred)
  - allows sophisticated error checking
  - allows a seamless extraction of concurrency

Thanks for the attention!
Questions ?