CSE 230

Concurrency: STM

Slides due to: Kathleen Fisher, Simon Peyton Jones, Satnam Singh, Don Stewart

The Grand Challenge

How to properly use multi-cores?

Need new programming models!

Parallelism vs Concurrency

- A **parallel** program exploits real parallel computing resources to *run faster* while computing the *same answer*.
  - Expectation of genuinely simultaneous execution
  - Deterministic
- A **concurrent** program models independent agents that can communicate and synchronize.
  - Meaningful on a machine with one processor
  - Non-deterministic

Concurrent Programming

Essential For Multicore Performance

Concurrent Programming

State-of-the-art is 30 years old!

Locks and condition variables

Java: `synchronized`, `wait`, `notify`

Locks etc. Fundamentally Flawed

"Building a sky-scraper out of matchsticks"

What's Wrong With Locks?

Races

Forgotten locks lead to inconsistent views

Deadlock

Locks acquired in "wrong" order

Lost Wakeups

Forgotten notify to condition variables

Diabolical Error recovery

Even Worse! Locks Don't Compose

```
class Account{
  float balance;

  synchronized void deposit(float amt) {
    balance += amt;
  }

  synchronized void withdraw(float amt) {
    if (balance < amt)
      throw new OutOfMoneyError();
    balance -= amt;
  }
}
```

A Correct bank `Account` class

Write code to `transfer` funds between accounts

Even Worse! Locks Don't Compose

1st Attempt `transfer` = `withdraw` then `deposit`

```
class Account{
  float balance;
  synchronized void deposit(float amt) {
    balance += amt;
  }
  synchronized void withdraw(float amt) {
    if(balance < amt)
      throw new OutOfMoneyError();
    balance -= amt;
  }
  void transfer(Acct other, float amt) {
    other.withdraw(amt);
    this.deposit(amt);}
}
```

Even Worse! Locks Don’t Compose

1st Attempt **transfer = withdraw then deposit**

```
class Account{
  float balance;
  synchronized void deposit(float amt) {
    balance += amt;
  }
  synchronized void withdraw(float amt) {
    if(balance < amt)
      throw new OutOfMoneyError();
    balance -= amt;
  }
  void transfer(Acct other, float amt) {
    other.withdraw(amt);
    this.deposit(amt);}
}
```

Race Condition Wrong sum of balances

Even Worse! Locks Don’t Compose

2st Attempt: synchronized transfer

```
class Account{
  float balance;
  synchronized void deposit(float amt){
    balance += amt;
  }
  synchronized void withdraw(float amt){
    if(balance < amt)
      throw new OutOfMoneyError();
    balance -= amt;
  }
  synchronized void transfer(Acct other, float amt){
    other.withdraw(amt);
    this.deposit(amt);}
}
```


Even Worse! Locks Don’t Compose

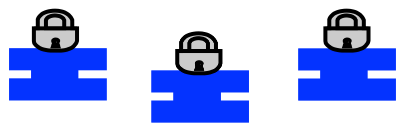
2st Attempt: synchronized transfer

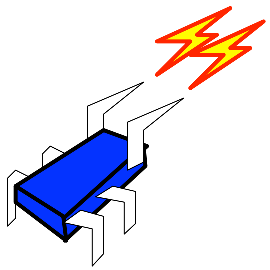
```
class Account{
  float balance;
  synchronized void deposit(float amt){
    balance += amt;
  }
  synchronized void withdraw(float amt){
    if(balance < amt)
      throw new OutOfMoneyError();
    balance -= amt;
  }
  synchronized void transfer(Acct other, float amt){
    other.withdraw(amt);
    this.deposit(amt);}
}
```

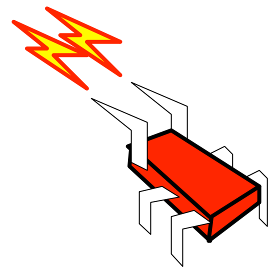
Deadlocks with Concurrent reverse transfer

Locks are absurdly hard to get right

Scalable double-ended queue: one lock per cell







No interference

If ends “far” apart

But watch out!

If queue is 0, 1, or 2 elements long

Locks are absurdly hard to get right

Coding Style	Difficulty of queue implementation
Sequential code	Undergraduate

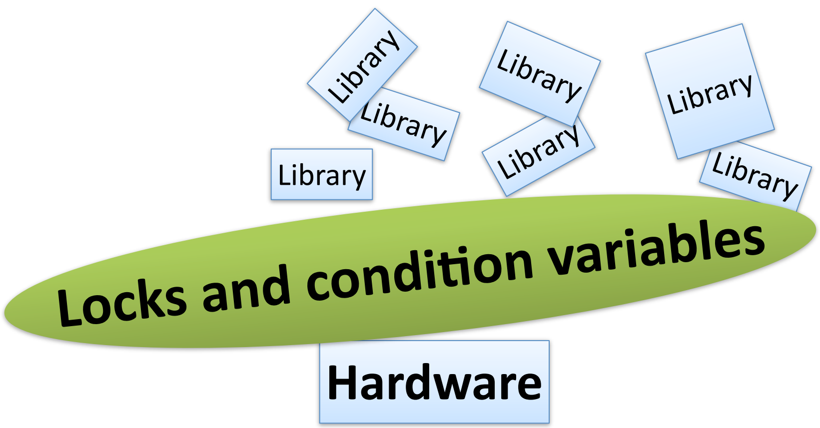
Locks are absurdly hard to get right

Coding Style	Difficulty of queue implementation
Sequential code	Undergraduate
Locks & Conditions	Major publishable result*

*Simple, fast, and practical non-blocking and blocking concurrent queue algorithms

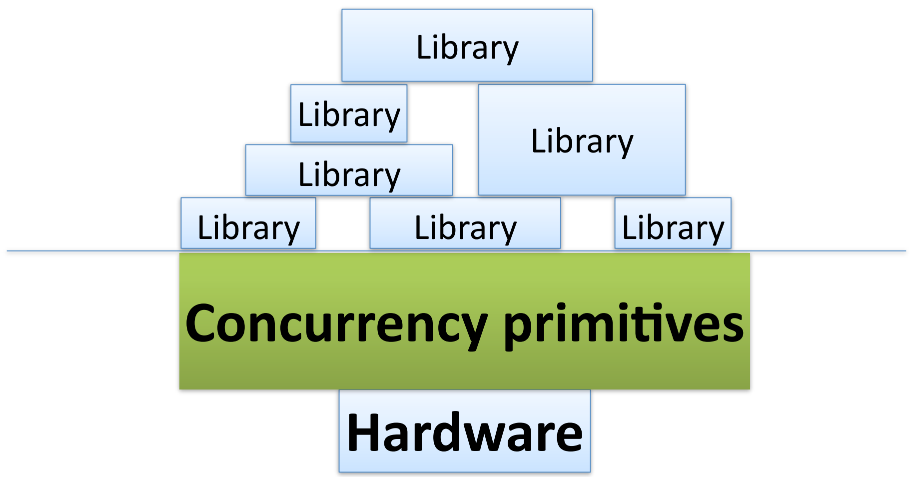
What we have

Locks and Conditions: Hard to use & Don't compose



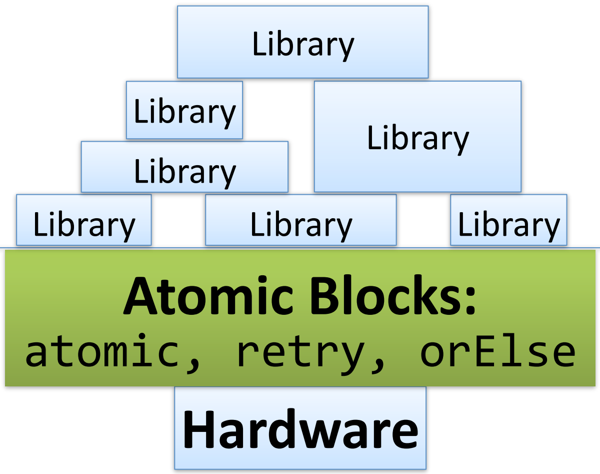
What we want

Libraries Build Layered Concurrency Abstractions



Idea: Replace locks with atomic blocks

Atomic Blocks/STM: Easy to use & Do compose



Locks are absurdly hard to get right

Coding Style	Difficulty of queue implementation
Sequential code	Undergraduate
Locks & Conditions	Major publishable result*
Atomic blocks(STM)	Undergraduate

[*Simple, fast, and practical non-blocking and blocking concurrent queue algorithms]

Atomic Memory Transactions

cf "**ACID**" database transactions

```
atomic {...sequential code...}
```

Wrap `atomic` around sequential code

All-or-nothing semantics: `atomic` commit

Atomic Memory Transactions

cf "**ACID**" database transactions

```
atomic {...sequential code...}
```

Atomic Block Executes in Isolation

No Data Race Conditions!

Atomic Memory Transactions

cf "**ACID**" database transactions

`atomic {...sequential code...}`

There Are No Locks

Hence, no deadlocks!

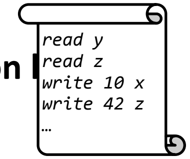
How it Works

`atomic {...sequential code...}`

Optimistic Concurrency

Execute code without any locks.

Record reads/writes in thread-local transaction



Writes go to the log only, not to memory.

At the end, transaction validates the log

If valid, *atomically commit* changes to memory

If invalid, *re-run* from start, discarding changes

Why it Doesn't Work...

`atomic {...sequential code...}`

Logging Memory Effects is Expensive

Huge slowdown on memory read/write

Cannot "Re-Run", Arbitrary Effects

How to "retract" email?

How to "un-launch" missile?

STM in Haskell

Haskell Fits the STM Shoe

Haskellers brutally trained from birth
to use memory/IO effects sparingly!

Issue: Logging Memory Is Expensive

Haskell already partitions world into
Immutable values (zillions and zillions)
Mutable locations (very few)

Solution: Only log mutable locations!

Issue: Logging Memory Is Expensive

Haskell already paid the bill!
Reading and Writing locations are
Expensive function calls

Logging Overhead
Lower than in imperative languages

Issue: Undoing Arbitrary IO

Types control where IO effects happen
Easy to keep them out of transactions

Monads Ideal For Building Transactions
Implicitly (invisibly) passing logs

Tracking Effects with Types

```
main = do { putStr (reverse "yes");
            putStr "no" }
```

```
(reverse "yes") :: String  -- No effects
(putStr  "no" ) :: IO ()   -- Effects okay
```

Main program is a computation with effects

```
main :: IO ()
```


1. **Mutable State**
2. **Concurrency**
3. **Synchronization**
4. **STM/Atomic Blocks**

Mutable State via the IO Monad

```
newRef   :: a -> IO (IORef a)
readRef  :: IORef a -> IO a
writeRef :: IORef a -> a -> IO ()
```

Reads and Writes are 100% Explicit

`(r+6)` is rejected as `r :: IORef Int`

Mutable State via the IO Monad

```
main = do r <- newIORef 0
          incR r
          s <- readIORef r
          print s

incR :: IORef Int -> IO ()
incR = do v <- readIORef r
          writeIORef r (v+1)
```

```
newRef   :: a -> IO (IORef a)
readRef  :: IORef a -> IO a
writeRef :: IORef a -> a -> IO ()
```

1. Mutable State
2. Concurrency
3. Synchronization
4. STM/Atomic Blocks

Concurrency in Haskell

forkIO function spawns a thread

Takes an IO action as argument

```
forkIO :: IO a -> IO ThreadId
```


Concurrency in Haskell

Data Race

```
main = do r <- newIORef 0
          forkIO $ incR r
          incR r
          print s

incR :: IORef Int -> IO ()
incR = do v <- readIORef r
          writeIORef r (v+1)
```

```
newRef   :: a -> IO (IORef a)
readRef  :: IORef a -> IO a
writeRef :: IORef a -> a -> IO ()
forkIO   :: IORef a -> IO ThreadId
```


1. Mutable State
2. Concurrency
3. Synchronization
4. STM/Atomic Blocks

Atomic Blocks in Haskell

goto code

1. **Mutable State**
2. **Concurrency**
3. **Synchronization**
4. **STM/Atomic Blocks**

Atomic Blocks in Haskell

```
atomically :: IO a -> IO a
```

`atomically act`

Executes `act` atomically

Atomic Blocks in Haskell

```
atomically :: IO a -> IO a
```

```
main = do r <- newRef 0
          forkIO $ atomically $ incR r
          atomically $ incR r
```

`atomic` Ensures No Data Races!

Atomic Blocks in Haskell

Data Race

```
main = do r <- newRef 0
          forkIO $ incR r
          atomically $ incR r
```

What if we use `incR` outside block?

Yikes! Races in code inside & outside!

A Better Type for Atomic

`STM` = Trans-actions

`Tvar` = Imperative transaction variables

```
atomic    :: STM a -> IO a
newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

Types ensure **`Tvar`** only touched in **`STM`** action

Type System Guarantees

You cannot forget `atomically`

Only way to execute `STM` action

```
incT :: TVar Int -> STM ()
incT r = do v <- readTVar r
            writeTVar r (v+1)
main   = do r <- atomically $ newTvar 0 TVar
            forkIO     $ atomically $ incT r
            atomically $ incT r
            ...
```


Type System Guarantees

Outside Atomic Block

Can't fiddle with `TVars`

Inside Atomic Block

Can't do IO , Can't manipulate imperative variables

```
atomic $ if x<y then launchMissiles
```

Type System Guarantees

Note: `atomically` is a function

not a special syntactic construct

...and, so, best of all...

(Unlike Locks) STM Actions Compose!

```
incT :: TVar Int -> STM ()
incT r = do v <- readTVar r
            writeTVar r (v+1)

incT2 :: TVar Int -> STM ()
incT2 r = do {incT r; incT r}

foo :: IO ()
foo = ...atomically $ incT2 r...
```

Glue STM Actions Arbitrarily

Wrap with atomic to get an IO action

Types ensure STM action is atomic

STM Type Supports Exceptions

```
throw :: Exception -> STM a
catch :: STM a ->(Exception->STM a)-> STM a
```

No need to restore invariants, or release locks!

In `atomically act` if `act` throws exception:

1. Transaction is aborted with no effect,
2. Exception is propagated to enclosing IO code*

[*Composable Memory Transactions]

Transaction Combinators

#1 retry: Compositional Blocking

```
retry :: STM ()
```

"Abort current transaction & re-execute from start"

```
withdraw :: TVar Int -> Int -> STM ()
withdraw acc n = do bal <- readTVar acc
                    if bal < n then retry
                    writeTVar acc (bal-n)
```


#1 retry: Compositional Blocking

```
retry :: STM ()
```

Implementation Avoids Busy Waiting

Uses logged reads to block till a read-var (eg. `acc`) changes

```
withdraw :: TVar Int -> Int -> STM ()
withdraw acc n = do bal <- readTVar acc
                    if bal < n then retry
                    writeTVar acc (bal-n)
```


#1 retry: Compositional Blocking

```
retry :: STM ()
```

No Condition Variables!

Uses logged reads to block till a read-var (eg. acc) changes

Retrying thread is woken on write, so no forgotten notifies

```
withdraw :: TVar Int -> Int -> STM ()
withdraw acc n = do bal <- readTVar acc
                    if bal < n then retry
                    writeTVar acc (bal-n)
```


#1 retry: Compositional Blocking

```
retry :: STM ()
```

No Condition Variables!

No danger of forgetting to test conditions

On waking as transaction runs from the start.

```
withdraw :: TVar Int -> Int -> STM ()
withdraw acc n = do bal <- readTVar acc
                    if bal < n then retry
                    writeTVar acc (bal-n)
```

Why is `retry` Compositional?

Can appear anywhere in an STM Transaction

Nested arbitrarily deeply inside a call

```
atomic $ do withdraw a1 3
            withdraw a2 7
```

Waits untill `a1>3` AND `a2>7`

Without changing/knowing `withdraw` code

Hoisting Guards Is Not Compositional

```
atomic (a1>3 && a2>7)
{   ...stuff...
}
```

Breaks abstraction of "...stuff..."

Need to know code to expose guards

#2 `orElse`: Choice

How to transfer $3 from a1 **or** a2 to b?

Try this...

...and if it retries, try this

```
atomically $ do withdraw a1 3`orelse` withdraw a2 3
                deposit b 3
```

...and and then do this

```
orElse :: STM a -> STM a -> STM a
```


Choice Is Composable Too!

```
transfer a1 a2 b = do withdraw a1 3`orElse` withdraw a2 3
                      deposit b 3
```

```
atomically $ transfer a1 a2 b
             `orElse`
             transfer a3 a4 b
```

`transfer` calls `orElse`

But calls to it can be composed with `orElse`

Ensuring Correctness of Concurrent Accesses?

e.g. account should never go below 0

Transaction Invariants

Assumed on Entry, Verified on Exit

Only Tested If Invariant's TVar changes

#3 always: Enforce Invariants

```
always :: STM Bool -> STM ()
```

```
checkBal   :: TVar Int -> STM Bool
checkBal v = do cts <- readTVar v
                return (v > 0)

newAccount :: STM (TVar Int)
newAccount = do v <- newTVar 0
                always $ checkBal v
                return v
```

An arbitrary
boolean valued
STM action

Every Transaction that touches acct will check invariant

If the check fails, the transaction restarts

#3 always: Enforce Invariants

```
always :: STM Bool -> STM ()
```

Adds a new invariant to a global pool

Conceptually, all invariants checked on all commits

Implementation Checks Relevant Invariants

That read TVars written by the transaction

Recap: Composing Transactions

A transaction is a value of type `STM a`

Transactions are first-class values

Big Tx By Composing Little Tx

sequence, choice, block ...

To Execute, Seal The Transaction

```
atomically :: STM a -> IO a
```


Complete Implementation in GHC6

Performance is similar to Shared-Var

Need more experience using STM in practice...

You can play with it*

Final will have some STM material ☺

[* Beautiful Concurrency]

STM in Mainstream Languages

Proposals for adding STM to Java etc.

```
class Account {
  float balance;
  void deposit(float amt) {
    atomic { balance += amt; }
  }
  void withdraw(float amt) {
    atomic {
      if(balance < amt) throw new OutOfMoneyError();
      balance -= amt;  }
  }
  void transfer(Acct other, float amt) {
    atomic {  // Can compose withdraw and deposit.
      other.withdraw(amt);
      this.deposit(amt); }
  }
}
```


Mainstream Types Don't Control Effects

So Code Inside Tx Can Conflict with Code Outside!

Weak Atomicity

Outside code sees **inconsistent** memory

Avoid by placing all shared mem access in Tx

Strong Atomicity

Outside code guaranteed **consistent** memory view

Causes big performance hit

A Monadic Skin

In C/Java, IO is Everywhere

No need for special type, all code is in "IO monad"

Haskell Gives You A Choice

When to be in IO monad vs when to be purely functional

Haskell Can Be Imperative BUT C/Java Cannot Be Pure!

Mainstream PLs lack a statically visible pure subset

The separation facilitates concurrent programming…

Conclusions

STM raises abstraction for concurrent programming

Think high-level language vs assembly code

Whole classes of low-level errors are eliminated.

But not a silver bullet!

Can still write buggy programs

Concurrent code still harder than sequential code

Only for shared memory, not message passing

There is a performance hit

But it seems acceptable, and things can only get better…

Mutable State via the IO Monad

```
main = do r <- newIORef 0
          incR r
          s <- readIORef r
          print s

incR :: IORef Int -> IO ()
incR = do v <- readIORef r
          writeIORef r (v+1)
```

```
newRef   :: a -> IO (IORef a)
readRef  :: IORef a -> IO a
writeRef :: IORef a -> a -> IO ()
```