cs242

SOFTWARE TRANSACTIONAL MEMORY

Kathleen Fisher

Reading: "Beautiful Concurrency",
"The Transactional Memory / Garbage Collection Analogy"

Thanks to Simon Peyton Jones for these slides.

The Context

- Multi-cores are coming!
  - For 50 years, hardware designers delivered 40-50% increases per year in sequential program speed.
  - Around 2004, this pattern failed because power and cooling issues made it impossible to increase clock frequencies.
  - Now hardware designers are using the extra transistors that Moore's law is still delivering to put more processors on a single chip.

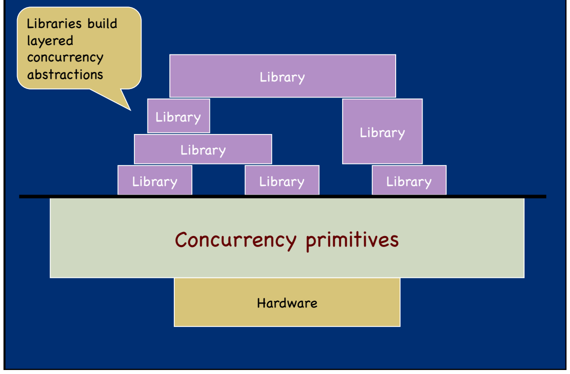
If we want to improve program speed, concurrent programs are no longer optional.

Concurrent Programming

- Concurrent programming is essential to improve performance on a multi-core.
- Yet the state of the art in concurrent programming is 30 years old: locks and condition variables. (In Java: **`synchronized`**, **`wait`**, and **`notify`**.)
- Locks and condition variables are fundamentally flawed: it's like building a sky-scraper out of bananas.

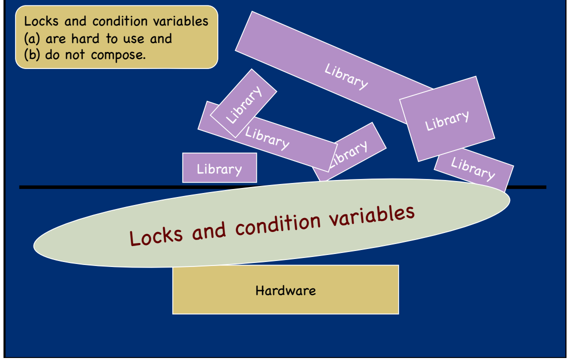
This lecture describes significant recent progress: bricks and mortar instead of bananas.

What we want



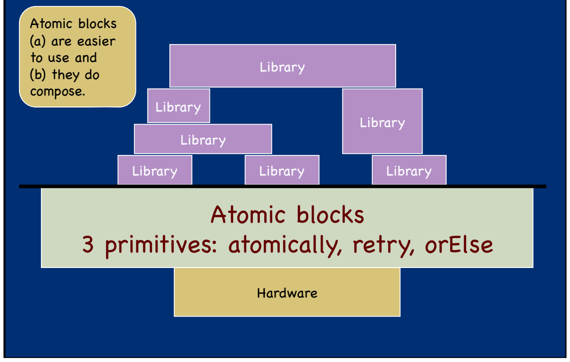
Libraries build layered concurrency abstractions

What we have



Locks and condition variables (a) are hard to use and (b) do not compose.

Idea: Replace locks with **Atomic Blocks**



Atomic blocks (a) are easier to use and (b) they do compose.

What's wrong with locks?

A 30-second review:

- **Races**: forgotten locks lead to inconsistent views
- **Deadlock**: locks acquired in "wrong" order
- **Lost wakeups**: forgotten notify to condition variables
- **Diabolical error recovery**: need to restore invariants and release locks in exception handlers

- These are serious problems. But even worse…

Locks are Non-Compositional

- Consider a (correct) Java bank `Account` class:

```
class Account{
  float balance;

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

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

- Now suppose we want to add the ability to transfer funds from one account to another.

Locks are Non-Compositional

- Simply calling `withdraw` and `deposit` to implement `transfer` causes a race condition:

```
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_wrong1(Acct other, float amt) {
    other.withdraw(amt);
    // race condition: wrong sum of balances
    this.deposit(amt);}
}
```

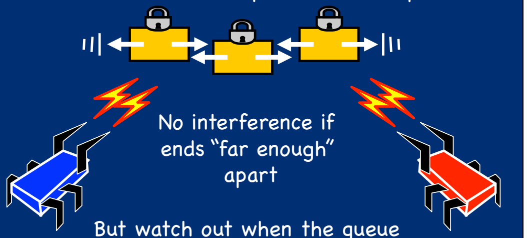
Locks are Non-Compositional

- Synchronizing `transfer` can cause deadlock:

```
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_wrong2(Acct other, float amt) {
    // can deadlock with parallel reverse-transfer
    this.deposit(amt);
    other.withdraw(amt);
  }
}
```

Locks are absurdly hard to get right

Scalable double-ended queue: one lock per cell

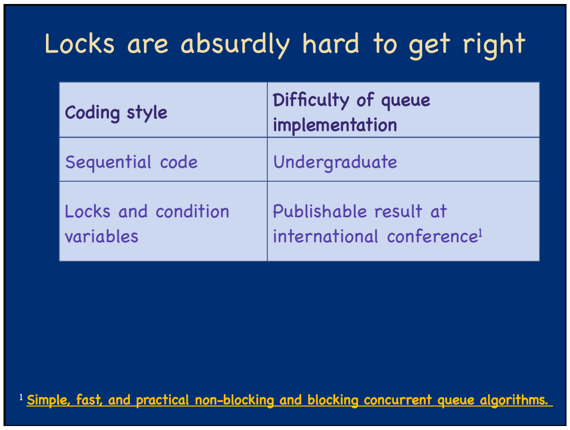


No interference if ends "far enough" apart

But watch out when the queue is 0, 1, or 2 elements long!

Locks are absurdly hard to get right

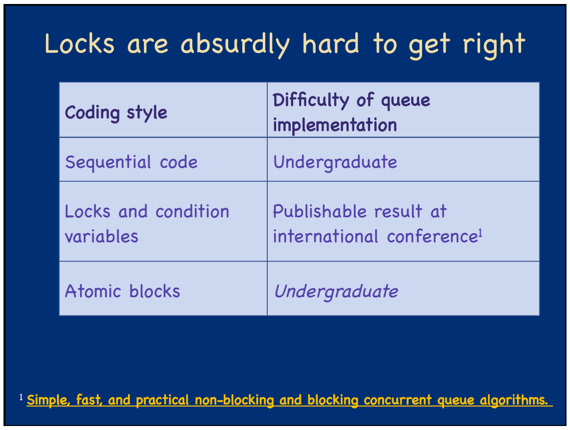
Coding style	Difficulty of queue implementation
Sequential code	Undergraduate



Locks are absurdly hard to get right

Coding style	Difficulty of queue implementation
Sequential code	Undergraduate
Locks and condition variables	Publishable result at international conference[1]

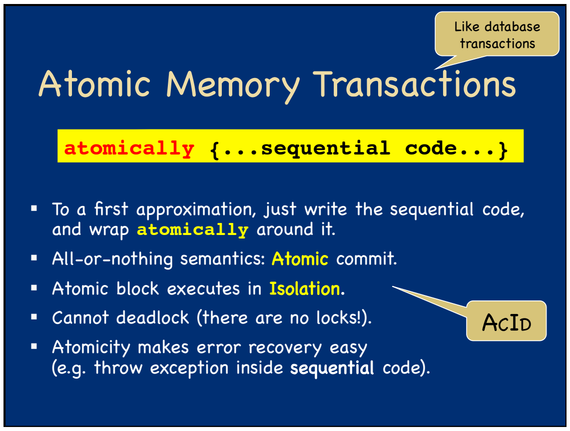
[1] Simple, fast, and practical non-blocking and blocking concurrent queue algorithms.



Locks are absurdly hard to get right

Coding style	Difficulty of queue implementation
Sequential code	Undergraduate
Locks and condition variables	Publishable result at international conference[1]
Atomic blocks	*Undergraduate*

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



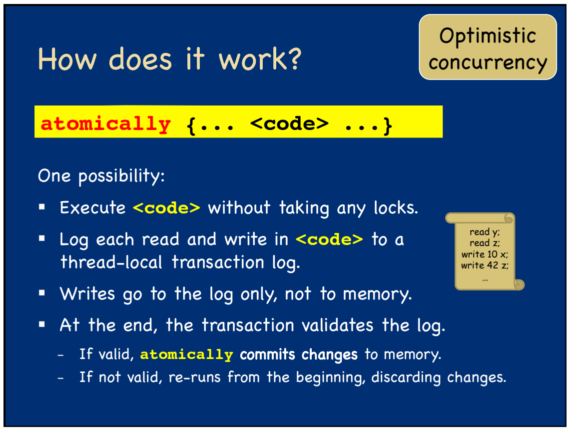
Like database transactions

Atomic Memory Transactions

`atomically {...sequential code...}`

- To a first approximation, just write the sequential code, and wrap **`atomically`** around it.
- All-or-nothing semantics: Atomic commit.
- Atomic block executes in Isolation.
- Cannot deadlock (there are no locks!).
- Atomicity makes error recovery easy (e.g. throw exception inside **sequential** code).

ACID



How does it work?

Optimistic concurrency

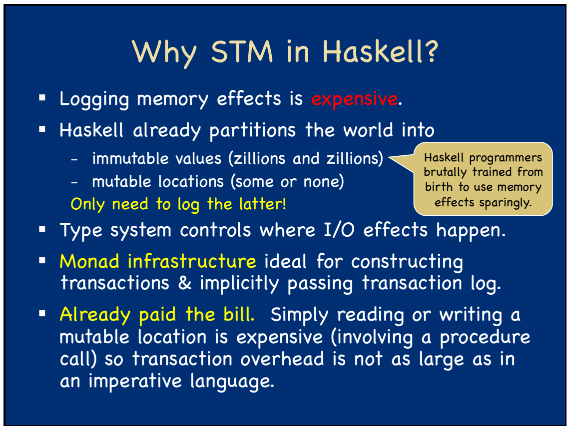
`atomically {... <code> ...}`

One possibility:

- Execute **`<code>`** without taking any locks.
- Log each read and write in **`<code>`** to a thread-local transaction log.
- Writes go to the log only, not to memory.
- At the end, the transaction validates the log.
  - If valid, **`atomically`** **commits changes** to memory.
  - If not valid, re-runs from the beginning, discarding changes.

read y;
read z;
write 10 x;
write 42 z;
…

Realizing STM in Haskell



Why STM in Haskell?

- Logging memory effects is expensive.
- Haskell already partitions the world into
  - immutable values (zillions and zillions)
  - mutable locations (some or none)

  Only need to log the latter!
- Type system controls where I/O effects happen.
- Monad infrastructure ideal for constructing transactions & implicitly passing transaction log.
- Already paid the bill. Simply reading or writing a mutable location is expensive (involving a procedure call) so transaction overhead is not as large as in an imperative language.

Haskell programmers brutally trained from birth to use memory effects sparingly.

Tracking Effects with Types

- Consider a simple Haskell program:

```
main = do { putStrLn (reverse “yes”);
            putStrLn “no” }
```

- Effects are explicit in the type system.

```
(reverse “yes”) :: String   – No effects
(putStr  “no” ) :: IO ()    – Effects okay
```

- Main program is a computation with effects.

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

Mutable State

```
newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

Recall that Haskell IO Monad functions `newIORef`, `readIORef`, and `writeIORef` manage mutable state.

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

incR :: IORef Int -> IO ()
incR r = do { v <- readIORef r;
              writeRef r (v+1) }
```

Reads and writes are 100% explicit. The type system disallows `(r + 6)` because `r :: IORef Int`.

Concurrency in Haskell

- The **`forkIO`** function spawns a thread.
- It takes an IO action as its argument.

```
forkIO :: IO () -> IO ThreadId
```

```
main = do { r <- newIORef 0;
            forkIO  (incR r);
            incR r;
            ... }

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

A race

Atomic Blocks in Haskell

- Idea: add a function **`atomically`** that executes its argument computation atomically.

```
atomically :: IO a -> IO a   – almost
```

```
main = do {
   r <- newIORef 0;
   forkIO (atomically (incR r));
   atomically (incR r);
   ... }
```

Worry: What prevents using **`incR`** outside **`atomically`**, which would allow data races between code inside atomic and outside?

A Better Type for **`Atomically`**

- Introduce a type for imperative transaction variables (**`TVar`**) and a new Monad (**`STM`**) to track transactions.
- Ensure **`TVars`** can only be modified in transactions.

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

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

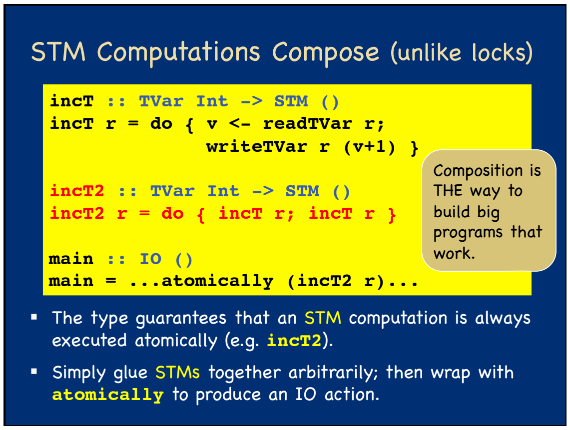
STM in Haskell

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

- Can’t fiddle with **`TVars`** outside atomic block. [good]
- Can’t do IO or manipulate regular imperative variables inside atomic block. [sad, but also good]

```
atomically (if x<y then launchMissiles)
```

- ...and, best of all...



STM Computations Compose (unlike locks)

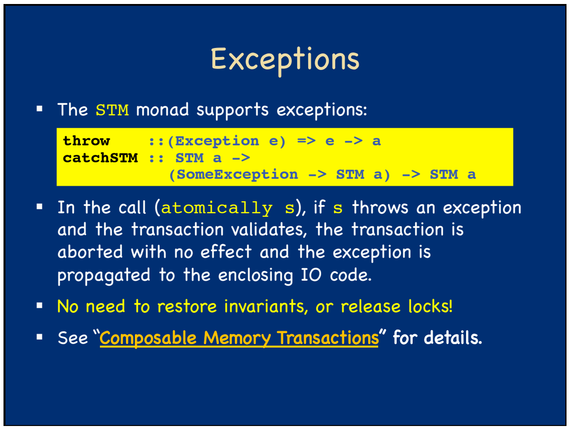
```
incT :: TVar Int -> STM ()
incT r = do { v <- readTVar r;
              writeTVar r (v+1) }

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

main :: IO ()
main = ...atomically (incT2 r)...
```

Composition is THE way to build big programs that work.

- The type guarantees that an STM computation is always executed atomically (e.g. **`incT2`**).
- Simply glue STMs together arbitrarily; then wrap with **`atomically`** to produce an IO action.



Exceptions

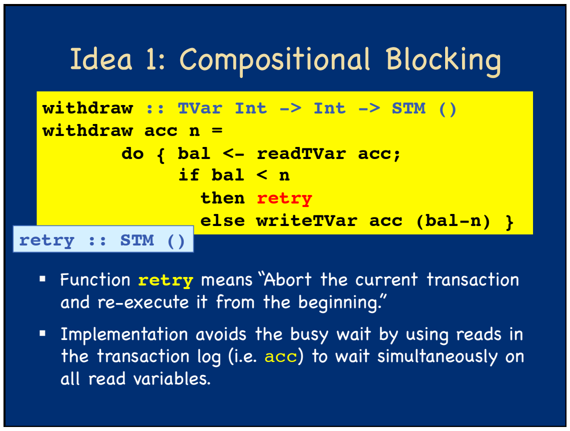
- The STM monad supports exceptions:

```
throw    ::(Exception e) => e -> a
catchSTM :: STM a ->
              (SomeException -> STM a) -> STM a
```

- In the call (`atomically s`), if `s` throws an exception and the transaction validates, the transaction is aborted with no effect and the exception is propagated to the enclosing IO code.
- No need to restore invariants, or release locks!
- See "Composable Memory Transactions" for details.

Three new ideas

`retry`
`orElse`
`always`

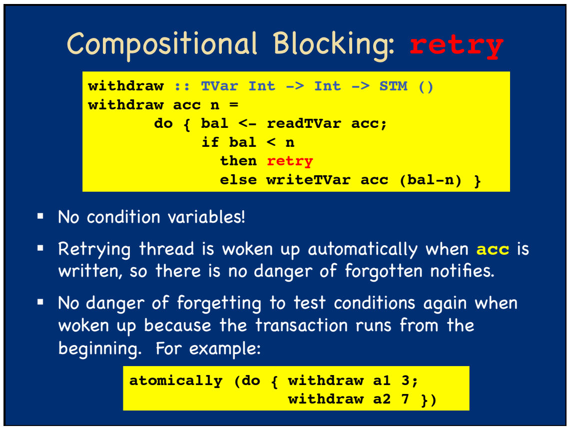


Idea 1: Compositional Blocking

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

`retry :: STM ()`

- Function **`retry`** means "Abort the current transaction and re-execute it from the beginning."
- Implementation avoids the busy wait by using reads in the transaction log (i.e. `acc`) to wait simultaneously on all read variables.

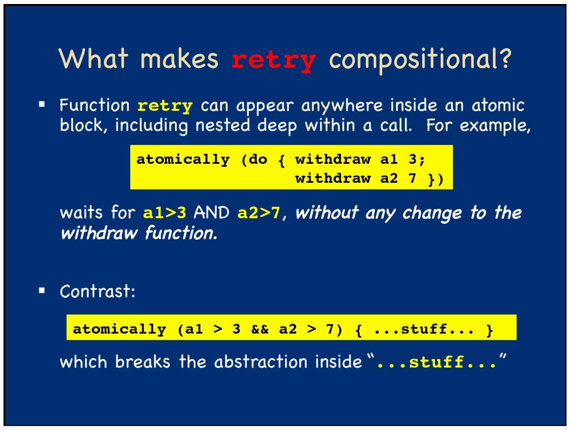


Compositional Blocking: `retry`

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

- No condition variables!
- Retrying thread is woken up automatically when **`acc`** is written, so there is no danger of forgotten notifies.
- No danger of forgetting to test conditions again when woken up because the transaction runs from the beginning. For example:

```
atomically (do { withdraw a1 3;
                 withdraw a2 7 })
```



What makes `retry` compositional?

- Function **`retry`** can appear anywhere inside an atomic block, including nested deep within a call. For example,

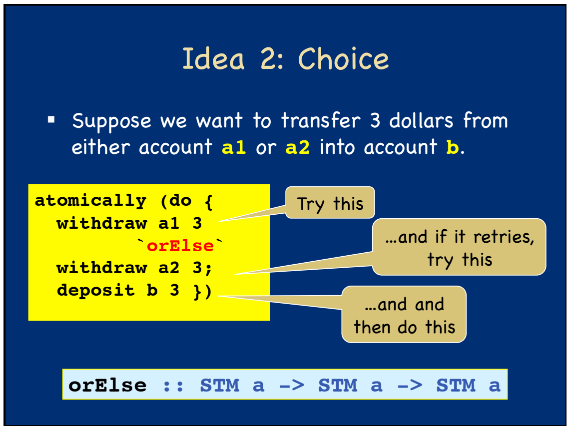
```
atomically (do { withdraw a1 3;
                 withdraw a2 7 })
```

waits for **`a1>3`** AND **`a2>7`**, ***without any change to the withdraw function.***

- Contrast:

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

which breaks the abstraction inside "**`...stuff...`**"



Idea 2: Choice

- Suppose we want to transfer 3 dollars from either account **a1** or **a2** into account **b**.

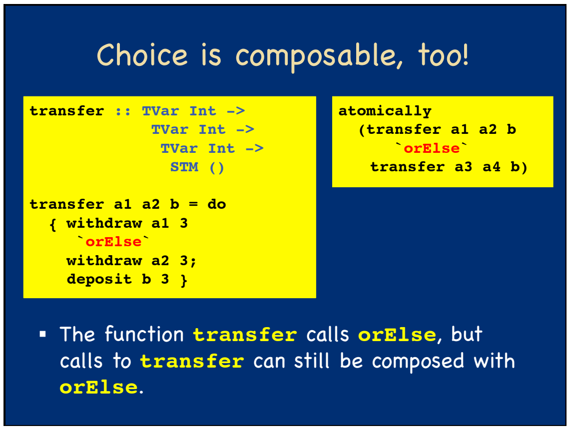
```
atomically (do {
  withdraw a1 3
         `orElse`
  withdraw a2 3;
  deposit b 3 })
```

Try this

…and if it retries, try this

…and and then do this

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



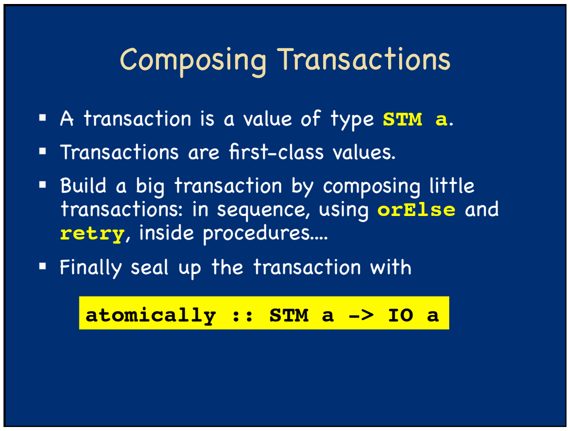
Choice is composable, too!

```
transfer :: TVar Int ->
             TVar Int ->
              TVar Int ->
               STM ()

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)
```

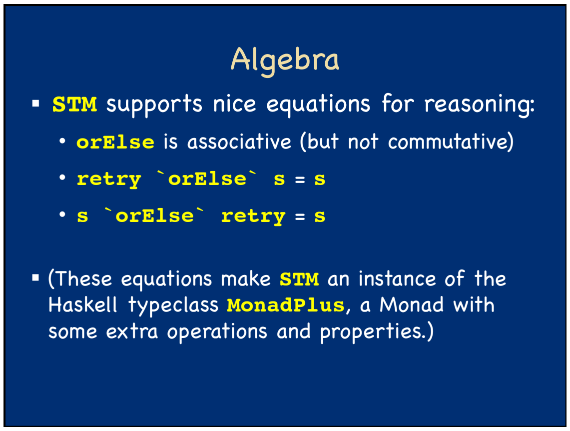
- The function **transfer** calls **orElse**, but calls to **transfer** can still be composed with **orElse**.



Composing Transactions

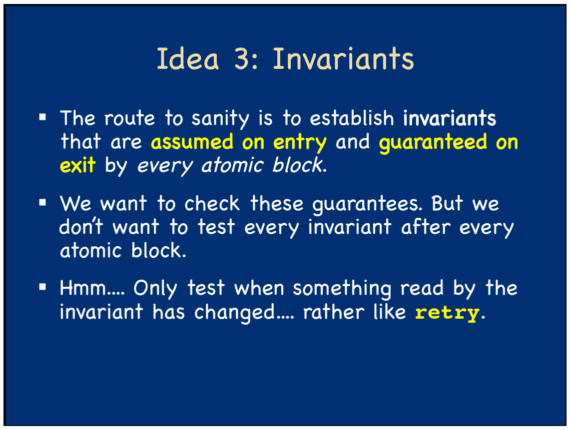
- A transaction is a value of type **STM a**.
- Transactions are first-class values.
- Build a big transaction by composing little transactions: in sequence, using **orElse** and **retry**, inside procedures….
- Finally seal up the transaction with

`atomically :: STM a -> IO a`



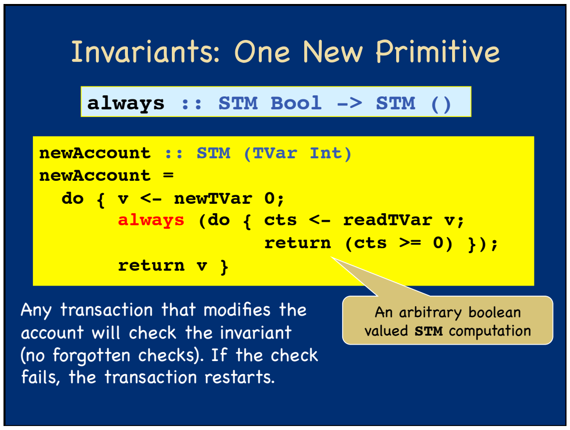
Algebra

- **STM** supports nice equations for reasoning:
  - **orElse** is associative (but not commutative)
  - **retry `orElse` s** = **s**
  - **s `orElse` retry** = **s**
- (These equations make **STM** an instance of the Haskell typeclass **MonadPlus**, a Monad with some extra operations and properties.)



Idea 3: Invariants

- The route to sanity is to establish **invariants** that are **assumed on entry** and **guaranteed on exit** by *every atomic block*.
- We want to check these guarantees. But we don't want to test every invariant after every atomic block.
- Hmm…. Only test when something read by the invariant has changed…. rather like **retry**.



Invariants: One New Primitive

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

```
newAccount :: STM (TVar Int)
newAccount =
  do { v <- newTVar 0;
       always (do { cts <- readTVar v;
                    return (cts >= 0) });
       return v }
```

An arbitrary boolean valued **STM** computation

Any transaction that modifies the account will check the invariant (no forgotten checks). If the check fails, the transaction restarts.

What **always** does

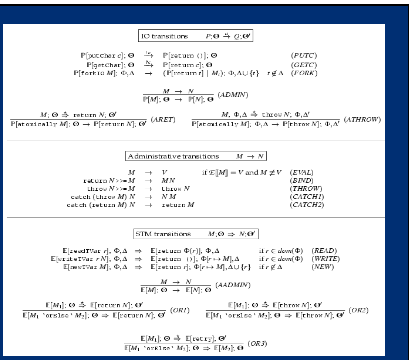
`always :: STM Bool -> STM ()`

- The function **always** adds a new invariant to a global pool of invariants.
- Conceptually, every invariant is checked as every transaction commits.
- But the implementation checks only invariants that read **TVars** that have been written by the transaction.
- …and garbage collects invariants that are checking dead **Tvars**.

What does it all mean?

- Everything so far is intuitive and arm-wavey.
- But what happens if it is raining, and you are inside an **orElse** and you throw an exception that contains a value that mentions…?
- We need a precise specification!

One exists



$\boxed{\text{IO transitions} \quad P;\Theta \xrightarrow{a} Q;\Theta'}$

$$\begin{array}{lcll} \mathbb{P}[\texttt{putChar}\ c];\ \Theta & \xrightarrow{!c} & \mathbb{P}[\texttt{return}\ ()];\ \Theta & (PUTC) \\ \mathbb{P}[\texttt{getChar}];\ \Theta & \xrightarrow{?c} & \mathbb{P}[\texttt{return}\ c];\ \Theta & (GETC) \\ \mathbb{P}[\texttt{forkIO}\ M];\ \Phi,\Delta & \rightarrow & (\mathbb{P}[\texttt{return}\ t]\ |\ M_t);\ \Phi,\Delta\cup\{t\} \quad t\notin\Delta & (FORK) \end{array}$$

$$\frac{M \ \rightarrow\ N}{\mathbb{P}[M];\ \Theta \ \rightarrow\ \mathbb{P}[N];\ \Theta}\ (ADMIN)$$

$$\frac{M;\ \Theta \overset{*}{\Rightarrow} \texttt{return}\ N;\ \Theta'}{\mathbb{P}[\texttt{atomically}\ M];\ \Theta \rightarrow \mathbb{P}[\texttt{return}\ N];\ \Theta'}\ (ARET) \qquad \frac{M;\ \Phi,\Delta \overset{*}{\Rightarrow} \texttt{throw}\ N;\ \Phi,\Delta'}{\mathbb{P}[\texttt{atomically}\ M];\ \Phi,\Delta \rightarrow \mathbb{P}[\texttt{throw}\ N];\ \Phi,\Delta'}\ (ATHROW)$$

$\boxed{\text{Administrative transitions} \quad M \ \rightarrow\ N}$

$$\begin{array}{rcll} M & \rightarrow & V & \text{if } \mathcal{E}[\![M]\!] = V \text{ and } M \not\equiv V \quad (EVAL) \\ \texttt{return}\ N \texttt{>>=} M & \rightarrow & M\ N & (BIND) \\ \texttt{throw}\ N \texttt{>>=} M & \rightarrow & \texttt{throw}\ N & (THROW) \\ \texttt{catch}\ (\texttt{throw}\ M)\ N & \rightarrow & N\ M & (CATCH1) \\ \texttt{catch}\ (\texttt{return}\ M)\ N & \rightarrow & \texttt{return}\ M & (CATCH2) \end{array}$$

$\boxed{\text{STM transitions} \quad M;\Theta \ \Rightarrow\ N;\Theta'}$

$$\begin{array}{rcll} \mathbb{E}[\texttt{readTVar}\ r];\ \Phi,\Delta & \Rightarrow & \mathbb{E}[\texttt{return}\ \Phi(r)];\ \Phi,\Delta & \text{if } r\in dom(\Phi) \quad (READ) \\ \mathbb{E}[\texttt{writeTVar}\ r\ N];\ \Phi,\Delta & \Rightarrow & \mathbb{E}[\texttt{return}\ ()];\ \Phi[r\mapsto M],\Delta & \text{if } r\in dom(\Phi) \quad (WRITE) \\ \mathbb{E}[\texttt{newTVar}\ M];\ \Phi,\Delta & \Rightarrow & \mathbb{E}[\texttt{return}\ r];\ \Phi[r\mapsto M],\Delta\cup\{r\} & \text{if } r\notin\Delta \quad (NEW) \end{array}$$

$$\frac{M \ \rightarrow\ N}{\mathbb{E}[M];\ \Theta \ \rightarrow\ \mathbb{E}[N];\ \Theta}\ (AADMIN)$$

$$\frac{\mathbb{E}[M_1];\ \Theta \overset{*}{\Rightarrow} \mathbb{E}[\texttt{return}\ N];\ \Theta'}{\mathbb{E}[M_1\ \texttt{`orElse`}\ M_2];\ \Theta \Rightarrow \mathbb{E}[\texttt{return}\ N];\ \Theta'}\ (OR1) \qquad \frac{\mathbb{E}[M_1];\ \Theta \overset{*}{\Rightarrow} \mathbb{E}[\texttt{throw}\ N];\ \Theta'}{\mathbb{E}[M_1\ \texttt{`orElse`}\ M_2];\ \Theta \Rightarrow \mathbb{E}[\texttt{throw}\ N];\ \Theta'}\ (OR2)$$

$$\frac{\mathbb{E}[M_1];\ \Theta \overset{*}{\Rightarrow} \mathbb{E}[\texttt{retry}];\ \Theta'}{\mathbb{E}[M_1\ \texttt{`orElse`}\ M_2];\ \Theta \Rightarrow \mathbb{E}[M_2];\ \Theta}\ (OR3)$$

See "Composable Memory Transactions" for details.

Haskell Implementation

- A complete, multiprocessor implementation of **STM** exists as of GHC 6.
- Experience to date: even for the most mutation-intensive program, the Haskell **STM** implementation is as fast as the previous **MVar** implementation.
  - The **MVar** version paid heavy costs for (usually unused) exception handlers.
- Need more experience using **STM** in practice, though!

STM in Mainstream Languages

- There are similar proposals for adding STM to Java and other mainstream languages.

```
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); }
  }
}
```

Weak vs Strong Atomicity

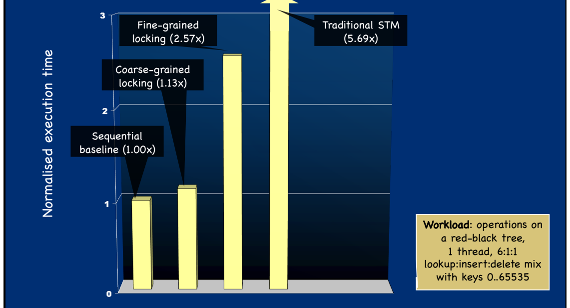
- Unlike Haskell, type systems in mainstream languages don't control where effects occur.
- What happens if code outside a transaction conflicts with code inside a transaction?
  - Weak Atomicity: Non-transactional code can see inconsistent memory states. Programmer should avoid such situations by placing all accesses to shared state in transaction.
  - Strong Atomicity: Non-transactional code is guaranteed to see a consistent view of shared state. This guarantee may cause a performance hit.

For more information: "Enforcing Isolation and Ordering in STM"

Performance

- At first, atomic blocks look insanely expensive. A naive implementation (c.f. databases):
  - Every load and store instruction logs information into a thread-local log.
  - A store instruction writes to the log only.
  - A load instruction consults the log first.
  - Run-time system (RTS) validates the log at the end of the atomic block.
    - If succeeds, the RTS atomically commits writes to shared memory.
    - If fails, the RTS restart the transaction.

State of the Art Circa 2003



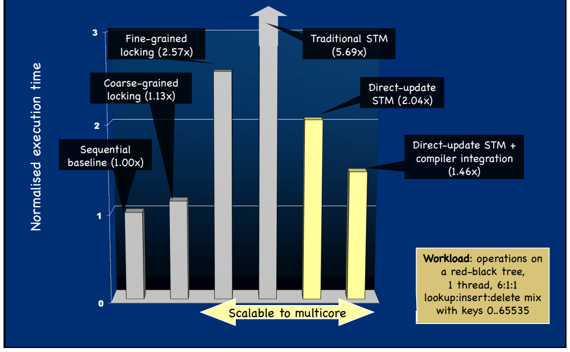
Workload: operations on a red-black tree, 1 thread, 6:1:1 lookup:insert:delete mix with keys 0..65535

See "Optimizing Memory Transactions" for more information.

New Implementation Techniques

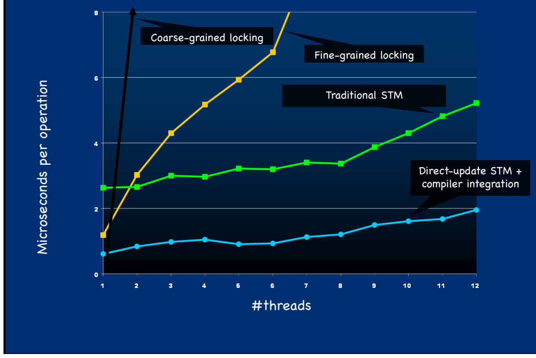
- Direct-update STM
  - Allows transactions to make updates in place in the heap.
  - Avoids reads needing to search the log to see earlier writes that the transaction has made.
  - Makes successful commit operations faster at the cost of extra work on contention or when a transaction aborts.
- Compiler integration
  - Decompose transactional memory operations into primitives.
  - Expose these primitives to compiler optimization (e.g. hoist concurrency control operations out of a loop).
- Runtime system integration
  - Integrates transactions with the garbage collector to scale to atomic blocks containing 100M memory accesses.

Results: Concurrency Overhead



Workload: operations on a red-black tree, 1 thread, 6:1:1 lookup:insert:delete mix with keys 0..65535

Results: Scalability



Performance, Summary

- Naïve STM implementation is hopelessly inefficient.
- There is a lot of research going on in the compiler and architecture communities to optimize STM.
- This work typically assumes transactions are smallish and have low contention. If these assumptions are wrong, performance can degrade drastically.
- We need more experience with "real" workloads and various optimizations before we will be able to say for sure that we can implement STM sufficiently efficiently to be useful.

Easier, But Not Easy.

- The essence of shared-memory concurrency is *deciding where critical sections should begin and end*. This is a hard problem.
  - Too small: application-specific data races (Eg, may see deposit but not withdraw if transfer is not atomic).
  - Too large: delay progress because deny other threads access to needed resources.

Still Not Easy, Example

- Consider the following Atomic Java program:

```
Initially, x = y = 0
```

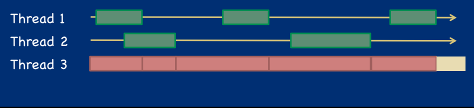
```
Thread 1
   atomic {                          //A0
     atomic { x = 1; }               //A1
     atomic { if (y==0) abort; }     //A2
   }
```

```
Thread 2
atomic {        //A3
  if (x==0) abort;
  y = 1;
}
```

- Successful completion requires **A3** to run after **A1** but before **A2**.
- So adding a critical section **A0** changes the behavior of the program (from terminating to non-terminating).

Starvation

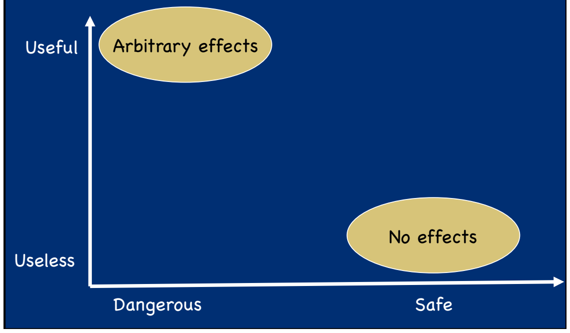
- Worry: Could the system "thrash" by transactions continually having conflicts and re-executing?
- No: A transaction can be forced to re-execute only if another succeeds in committing. That gives a strong *progress guarantee*.
- But: A particular thread could starve:



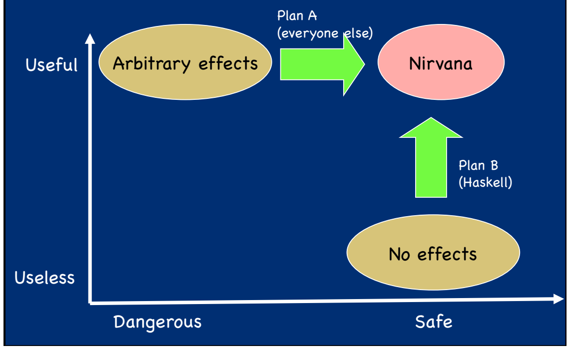
A Monadic Skin

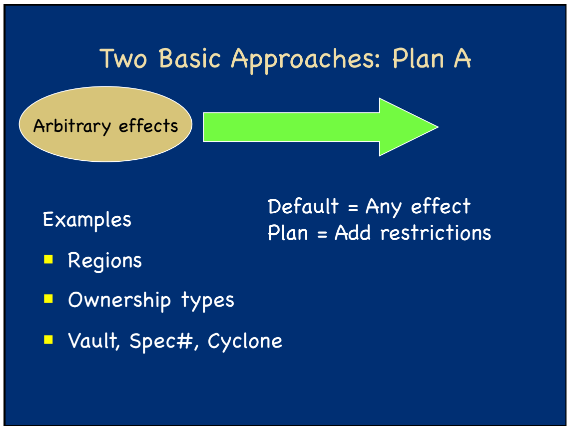
- In languages like ML or Java, the fact that the language is in the IO monad is baked in to the language. There is no need to mark anything in the type system because IO is everywhere.
- In Haskell, the programmer can choose when to live in the IO monad and when to live in the realm of pure functional programming.
- Interesting perspective: It is not Haskell that lacks imperative features, but rather the other languages that lack the ability to have a statically distinguishable pure subset.
- This separation facilitates concurrent programming.

The Central Challenge



The Challenge of Effects





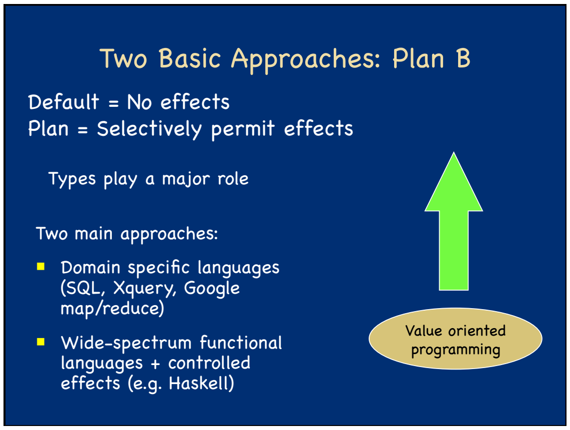
Two Basic Approaches: Plan A

Arbitrary effects

Default = Any effect
Plan = Add restrictions

Examples

- Regions
- Ownership types
- Vault, Spec#, Cyclone



Two Basic Approaches: Plan B

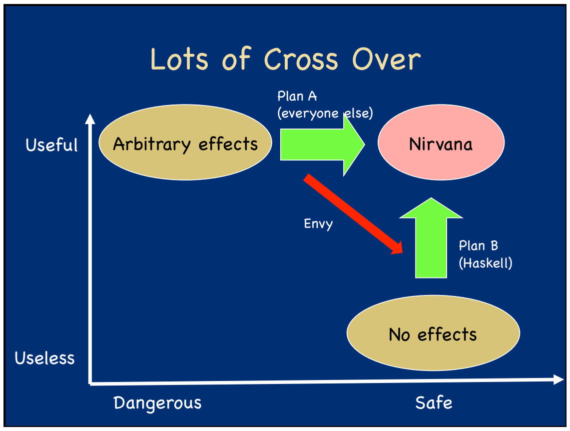
Default = No effects
Plan = Selectively permit effects

Types play a major role

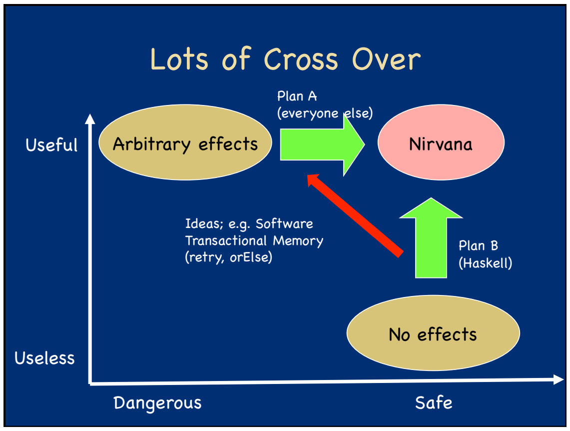
Two main approaches:

- Domain specific languages (SQL, Xquery, Google map/reduce)
- Wide-spectrum functional languages + controlled effects (e.g. Haskell)

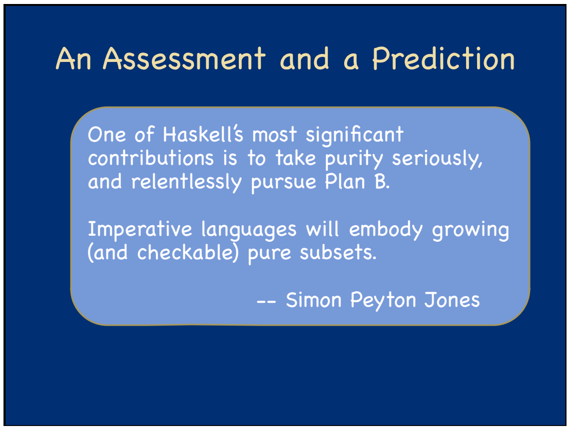
Value oriented programming



Lots of Cross Over



Lots of Cross Over

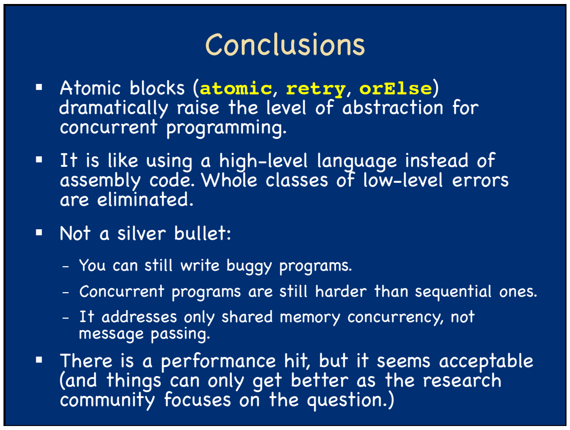


An Assessment and a Prediction

One of Haskell's most significant contributions is to take purity seriously, and relentlessly pursue Plan B.

Imperative languages will embody growing (and checkable) pure subsets.

-- Simon Peyton Jones



Conclusions

- Atomic blocks (**`atomic`**, **`retry`**, **`orElse`**) dramatically raise the level of abstraction for concurrent programming.
- It is like using a high-level language instead of assembly code. Whole classes of low-level errors are eliminated.
- Not a silver bullet:
  - You can still write buggy programs.
  - Concurrent programs are still harder than sequential ones.
  - It addresses only shared memory concurrency, not message passing.
- There is a performance hit, but it seems acceptable (and things can only get better as the research community focuses on the question.)