Concurrency Unlocked

transactional memory
for
composable concurrency

Adapted by BCP
from slides by Simon Peyton Jones
Based on work by Tim Harris, Maurice Herlihy, Simon Marlow, and Simon Peyton Jones

Concurrent programming is hard

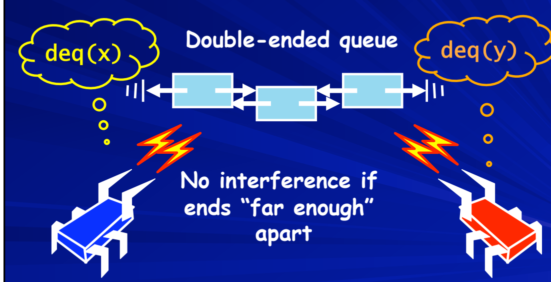
- Programmers have to think about all possible interleavings (and they aren't good at it)
- Testing is horrible: too many interleavings, lack of control
- Bugs are irreproducible

One solution: language support for concurrency abstractions
Market leader: locks / synchronised methods

Locks are broken

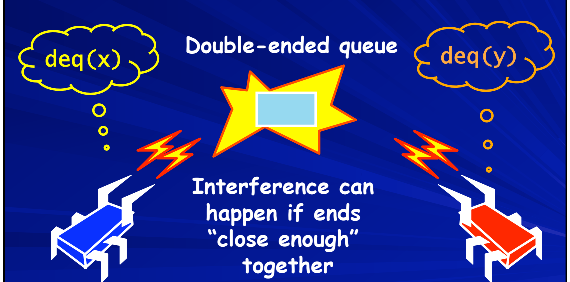
- **Races**: due to forgotten locks
- **Deadlock**: locks acquired in "wrong" order.
- **Lost wakeups**: forgotten notify to condition variable
- **Error recovery tricky**: need to restore invariants and release locks in exception handlers
- **Simplicity vs scalability tension**

Sadistic Homework Assignment



More sadism

[Michael & Scott, PODC 1996]



Locks are broken

- Races: due to forgotten locks
- Deadlock: locks acquired in "wrong" order.
- Lost wakeups: forgotten notify to condition variable
- Error recovery tricky: need to restore invariants and release locks in exception handlers
- Simplicity vs scalability tension
- ...but worst of all...

Locks do not compose

You cannot build a big working program from small working pieces

- A.withdraw(3): withdraw $3 from account A. Easy; use a synchronised method
- A.withdraw(3); B.deposit(3)
  Uh oh: an observer could see a state in which the money was in neither account

Loss of composition

- Expose the locking
  A.lock(); B.lock(); A.withdraw(3);
  B.deposit(3); A.unlock(); B.unlock()
- Uh oh. Danger of deadlock

```
if A<B then
    A.lock(); B.lock()
else
    B.lock(); A.lock()
end if
```

- Now transfer money from A's deposit account if A doesn't have enough money....

Composition of alternatives

- Method **m1** does a WaitAny(h1,h2) on two WaitHandles h1, h2. Ditto **m2**
- Can we WaitAny(**m1**,**m2**). No way!
- Instead, we break the abstraction and bubble up the WaitHandles we want to wait on to a top-level WaitAny, and then dispatch back to the handler code
- Same in Unix (select)

This way lies madness

Our main weapon in controlling program complexity is modular decomposition: build a big program by gluing together smaller ones

Locks eviscerate our main weapon

IDEA!

- **Software Transactional Memory**: Herlihy/Moss ISCA 1993
- STM in Java: Harris/Fraser OOPSLA 2003

Transactional memory

```
atomic
    A.withdraw(3)
    B.deposit(3)
end
```

- Steal ideas from the database folk
- **atomic** does what it says on the tin
- Directly supports what the programmer is trying to do: an atomic transaction against memory
- "Write the simple sequential code, and wrap **atomic** around it". (Recall sadistic homework.)

Transactional memory

- **No races**: no locks, so you can't forget to take one
- **No lock-induced deadlock**, because no locks
- **No lost wake-ups**, because no wake-up calls to forget [needs **retry**; wait a few slides]
- **Error recovery trivial**: an exception inside atomic aborts the transaction
- **Simple code is scalable**

How does it work?

Optimistic concurrency

```
atomic
   <body>
end
```

- Execute <body> without taking any locks
- Each read and write in <body> is logged to a thread-local transaction log
- Writes go to the log only, not to memory
- At the end, the transaction tries to **commit** to memory
- Commit may fail; then transaction is re-run

Transaction logs

Thread 1

```
atomic
→  v = read(bal)
   write( bal, v+1)
end
```

bal: 6

Thread 2

```
atomic
→  v = read(bal)
   write( bal, v-3)
end
```

Transaction log (Thread 1)

What	Value read	Value written
bal		

Transaction log (Thread 2)

What	Value read	Value written
bal		

Transaction logs

Thread 1

```
atomic
→  v = read(bal)
   write( bal, v+1)
end
```

bal: 6

Thread 2

```
atomic
   v = read(bal)
→  write( bal, v-3)
end
```

Transaction log (Thread 1)

What	Value read	Value written
bal		

Transaction log (Thread 2)

What	Value read	Value written
bal	6	

Transaction logs

Thread 1

```
atomic
   v = read(bal)
→  write( bal, v+1)
end
```

bal: 6

Thread 2

```
atomic
   v = read(bal)
→  write( bal, v-3)
end
```

Transaction log (Thread 1)

What	Value read	Value written
bal	6	

Transaction log (Thread 2)

What	Value read	Value written
bal	6	

Transaction logs

Thread 1

```
atomic
   v = read(bal)
   write( bal, v+1)
→ end
```

bal: 6

Thread 2

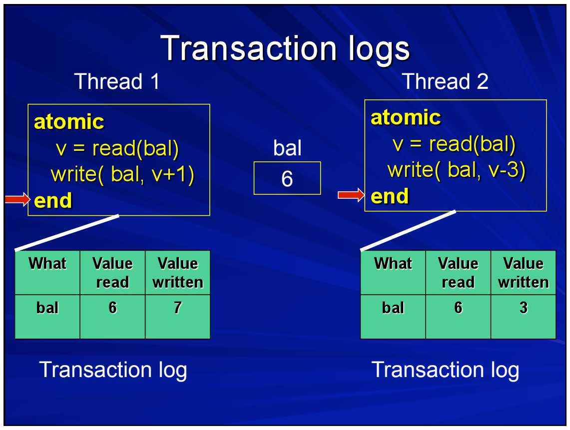
```
atomic
   v = read(bal)
→  write( bal, v-3)
end
```

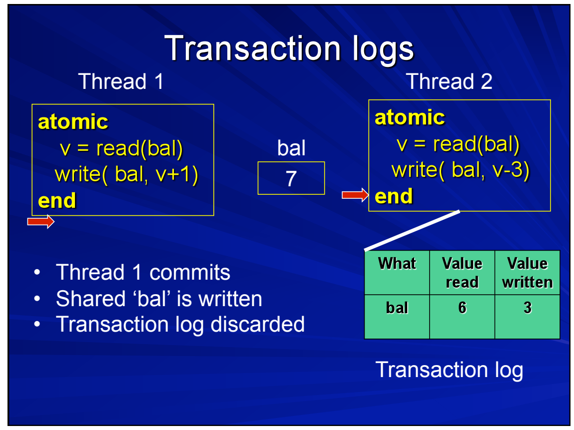
Transaction log (Thread 1)

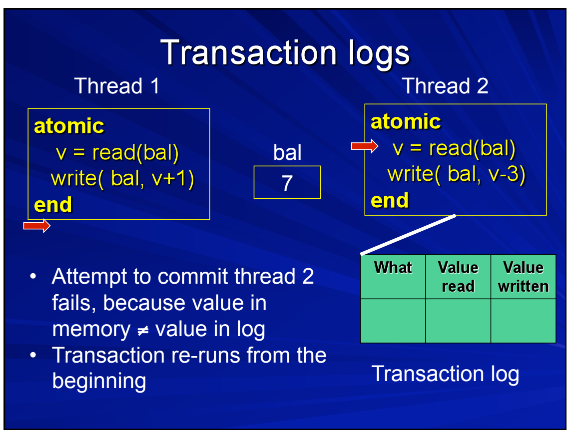
What	Value read	Value written
bal	6	7

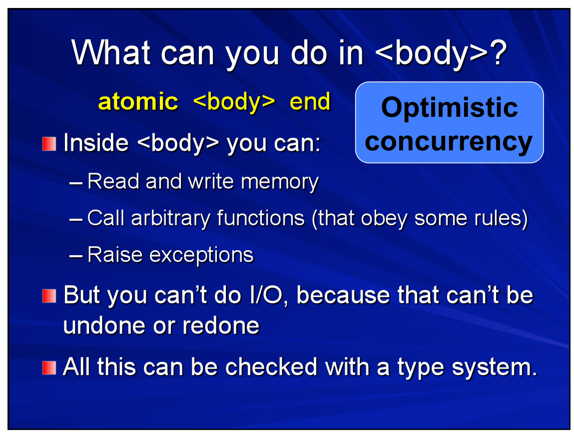
Transaction log (Thread 2)

What	Value read	Value written
bal	6	

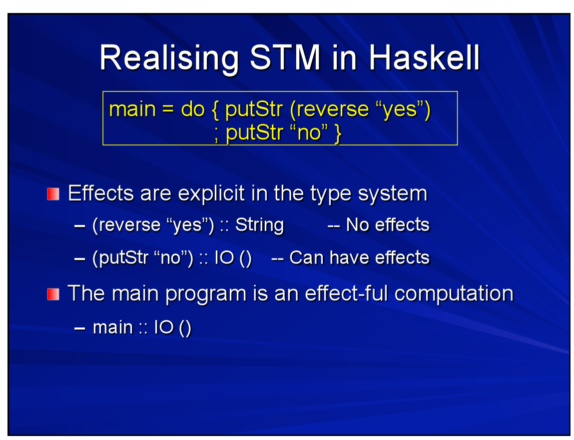

Transaction logs
Thread 1
Thread 2
atomic
v = read(bal)
write(bal, v+1)
end
bal
6
atomic
v = read(bal)
write(bal, v-3)
end
What | Value read | Value written
bal | 6 | 7
What | Value read | Value written
bal | 6 | 3
Transaction log
Transaction log


Transaction logs
Thread 1
Thread 2
atomic
v = read(bal)
write(bal, v+1)
end
bal
7
atomic
v = read(bal)
write(bal, v-3)
end
• Thread 1 commits
• Shared 'bal' is written
• Transaction log discarded
What | Value read | Value written
bal | 6 | 3
Transaction log


Transaction logs
Thread 1
Thread 2
atomic
v = read(bal)
write(bal, v+1)
end
bal
7
atomic
v = read(bal)
write(bal, v-3)
end
• Attempt to commit thread 2 fails, because value in memory ≠ value in log
• Transaction re-runs from the beginning
What | Value read | Value written
Transaction log


What can you do in <body>?
atomic <body> end
Optimistic concurrency
Inside <body> you can:
– Read and write memory
– Call arbitrary functions (that obey some rules)
– Raise exceptions
But you can't do I/O, because that can't be undone or redone
All this can be checked with a type system.


Realising STM in Haskell


Realising STM in Haskell
main = do { putStr (reverse "yes")
; putStr "no" }
Effects are explicit in the type system
– (reverse "yes") :: String -- No effects
– (putStr "no") :: IO () -- Can have effects
The main program is an effect-ful computation
– main :: IO ()

References

```
main = do { r <- newRef 0
          ; incR r
          ; s <- readRef r
          ; print s }

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

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

- Reads and writes are 100% explicit! You can't say (r + 6), because r::Ref Int
- Refs are totally non thread-safe (e.g. two concurrent calls to incR may step on each other).

Effectful computations are first-class

```
ntimes :: Int -> IO () -> IO ()
ntimes 0 action = return ()
ntimes n action = do { action; ntimes (n-1) action }

main :: IO ()
main = ntimes 10 (print "yes")
```

- fork spawns a thread; takes an action as its argument

  fork :: IO a -> IO ThreadId

```
main = do { r <- newRef 0
          ; fork (incR r)
          ; incR r
          ; ... }
```

NB: r is a free variable of the forked thread

STM in Haskell

- Key idea: **atomic :: IO a -> IO a**

```
main = do { r <- newRef 0
          ; fork (atomic (incR r))
          ; atomic (incR r)
          ; ... }
```

- atomic is a function, not a syntactic construct
- A worry: what stops you doing incR outside atomic?

STM in Haskell

- Better idea:

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

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

main = do { r <- atomic (newTVar 0)
          ; fork (atomic (incR r))
          ; atomic (incR r)
          ; ... }
```

STM in Haskell

```
atomic :: 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 inside atomic block [sad, but also good]

Two new ideas

Idea 1: modular blocking

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

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

- **retry** means "abort the current transaction and re-execute it from the beginning".
- Implementation avoids needless repetition by using reads in the transaction log (i.e. acc) to wait until something has changed

No condition variables

- No condition variables!
- Retrying thread is woken up automatically when acc is changed. No lost wake-ups!
- No danger of forgetting to test everything again when woken up; the transaction runs again from the beginning.
  e.g. atomic (do { withdraw a1 3
  ; withdraw a2 7 })

How is this "modular"?

- Because **retry** can appear anywhere inside an atomic block, including nested deep within a call.
- Contrast standard idiom:
  atomic (n > 0) { ...stuff... }
  which breaks the abstraction inside "...stuff..."
- Difficult to do that in a lock-based world, because you must release locks before blocking; but which locks?
- With STM, no locks => no danger of blocking while holding locks. This is a very strong property.

Farsite project (Jon Howell MSR)

"Your idea of using the writes from one transaction to wake up sleepy transactions is wonderful. We wanted to report on the effect your paper draft has *already had* on our project.

"...I told JD that I'd try to hack the Harris-and-company unblocking scheme into our stuff, but that he should slap me around if it ended up taking too long. We decided to check in after three days, and abandon after five. It took a day and a half....

"...In summary, using your composable blocking model is wonderful: it rips out a big chunk of our control flow related to liveness, and takes with it a whole class of potential bugs."

Idea 2: Choice

```
atomic (do {
   withdraw a1 3
     `orelse`
   withdraw a2 3
;  deposit b 3 })
```

Try this

...and if it retries, try this

...and then (in either case) 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
end
```

```
atomic
   (transfer a1 a2 b
       `orElse`
    transfer a3 a4 b)
```

- transfer has an orElse, but calls to transfer can still be composed with orElse

Algebra

- Nice equations:
  - orElse is associative (but not commutative)
  - retry `orElse` s = s
  - s `orElse` retry = s
- [For monad hackers] STM is an instance of MonadPlus.

Exceptions

- STM monad supports exceptions:

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

- In the call (atomic s), if s throws an exception, the transaction is aborted with no effect; and the exception is propagated into the IO monad
- No need to restore invariants or release locks!

But what does it all mean?

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

But what does it all mean?

- Small-step transition rules, in a similar (but not identical!) style to the Awkward Squad semantics

Thread soup	P, Q	$::=$	$M_t \mid (P \mid Q)$
Heap	Θ	$::=$	$r \hookrightarrow M$

I/O transitions $\quad P; \Theta \xrightarrow{a} Q; \Theta'$

thread states
shared heap (n.b.!)
observable event
New thread states and heap

Thread soup	P, Q	$::=$	$M_t \mid (P \mid Q)$
Heap	Θ	$::=$	$r \hookrightarrow M$
Allocations	Δ	$::=$	$r \hookrightarrow M$
Evaluation contexts	$\mathbb{E}$	$::=$	$[\cdot] \mid \mathbb{E}$ `>>=` $M \mid$ `catch` $\mathbb{E}\ M$
	$\mathbb{P}$	$::=$	$\mathbb{E}_t \mid (\mathbb{P} \mid P) \mid (P \mid \mathbb{P})$
Action	a	$::=$	$!c \mid ?c \mid \epsilon$

allocation records (for later)
two flavors of evaluation contexts

I/O transitions $\quad P; \Theta \xrightarrow{a} Q; \Theta'$

$$\mathbb{P}[\texttt{putChar}\ c];\ \Theta \xrightarrow{!c} \mathbb{P}[\texttt{return ()}];\ \Theta \qquad (PUTC)$$

$$\mathbb{P}[\texttt{getChar}];\ \Theta \xrightarrow{?c} \mathbb{P}[\texttt{return}\ c];\ \Theta \qquad (GETC)$$

$$\mathbb{P}[\texttt{forkIO}\ M];\ \Theta \rightarrow (\mathbb{P}[\texttt{return}\ t] \mid M_t);\ \Theta \quad t \notin \mathbb{P}, \Theta, M \qquad (FORK)$$

no need for restriction, since this rule can see *all* currently allocated thread ids

Administrative steps

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

Administrative transitions $\quad M \rightarrow N$

$$M \rightarrow V \quad \text{if } \mathcal{V}[\![M]\!] = V \text{ and } M \not\equiv V$$

$$\texttt{return}\ N \texttt{ >>= } M \rightarrow M\ N$$

$$\texttt{throw}\ N \texttt{ >>= } M \rightarrow \texttt{throw}\ N$$

$$\texttt{retry >>= } M \rightarrow \texttt{retry}$$

$$\texttt{catch (throw}\ M\texttt{)}\ N \rightarrow N\ M$$

$$\texttt{catch (return}\ M\texttt{)}\ N \rightarrow \texttt{return}\ M$$

Transactions

$$\begin{array}{rrcl} \text{Thread soup} & P,Q & ::= & M_t \mid (P \mid Q) \\ \text{Heap} & \Theta & ::= & r \hookrightarrow M \\ \text{Allocations} & \Delta & ::= & r \hookrightarrow M \\ \\ \text{Evaluation} & \mathbb{E} & ::= & [\cdot] \mid \mathbb{E} \mathtt{>>=} M \mid \mathtt{catch}\ \mathbb{E}\ M \\ \text{contexts} & \mathbb{P} & ::= & \mathbb{E}_t \mid (\mathbb{P} \mid P) \mid (P \mid \mathbb{P}) \\ \text{Action} & a & ::= & !c \mid ?c \mid \epsilon \end{array}$$

- atomic turns many STM steps (=>*) into one IO step (->):

$$\frac{M;\ \Theta, \{\}\ \overset{*}{\Rightarrow}\ \mathtt{return}\ N;\ \Theta', \Delta'}{\mathbb{P}[\mathtt{atomic}\ M];\ \Theta\ \rightarrow\ \mathbb{P}[\mathtt{return}\ N];\ \Theta'}\ (ARET)$$

- So what are the STM steps?

STM transitions

$$\begin{array}{rrcl} \text{Thread soup} & P,Q & ::= & M_t \mid (P \mid Q) \\ \text{Heap} & \Theta & ::= & r \hookrightarrow M \\ \text{Allocations} & \Delta & ::= & r \hookrightarrow M \\ \\ \text{Evaluation} & \mathbb{E} & ::= & [\cdot] \mid \mathbb{E} \mathtt{>>=} M \mid \mathtt{catch}\ \mathbb{E}\ M \\ \text{contexts} & \mathbb{P} & ::= & \mathbb{E}_t \mid (\mathbb{P} \mid P) \mid (P \mid \mathbb{P}) \\ \text{Action} & a & ::= & !c \mid ?c \mid \epsilon \end{array}$$

- Easy ones:

allocation record remembers which TVars have been allocated in this transaction

STM transitions $M; \Theta, \Delta \Rightarrow N; \Theta', \Delta'$

$$\begin{array}{rcll} \mathbb{E}[\mathtt{readTVar}\ r];\ \Theta, \Delta & \Rightarrow & \mathbb{E}[\mathtt{return}\ \Theta(r)];\ \Theta, \Delta & \text{if } r \in dom(\Theta) \\ \mathbb{E}[\mathtt{writeTVar}\ r\ M];\ \Theta, \Delta & \Rightarrow & \mathbb{E}[\mathtt{return}\ \ ()];\ \Theta[r \mapsto M], \Delta & \text{if } r \in dom(\Theta) \\ \mathbb{E}[\mathtt{newTVar}\ M];\ \Theta, \Delta & \Rightarrow & \mathbb{E}[\mathtt{return}\ r];\ \Theta[r \mapsto M], \Delta[r \mapsto M] & \text{if } r \notin dom(\Theta) \end{array}$$

add the new TVar to *both* the heap and the allocation record

choose a globally fresh address for the new TVar

Retry

$$\begin{array}{rrcl} \text{Thread soup} & P,Q & ::= & M_t \mid (P \mid Q) \\ \text{Heap} & \Theta & ::= & r \hookrightarrow M \\ \text{Allocations} & \Delta & ::= & r \hookrightarrow M \\ \\ \text{Evaluation} & \mathbb{E} & ::= & [\cdot] \mid \mathbb{E} \mathtt{>>=} M \mid \mathtt{catch}\ \mathbb{E}\ M \\ \text{contexts} & \mathbb{P} & ::= & \mathbb{E}_t \mid (\mathbb{P} \mid P) \mid (P \mid \mathbb{P}) \\ \text{Action} & a & ::= & !c \mid ?c \mid \epsilon \end{array}$$

- Here are the rules for retry:

Retry

$$\begin{array}{rrcl} \text{Thread soup} & P,Q & ::= & M_t \mid (P \mid Q) \\ \text{Heap} & \Theta & ::= & r \hookrightarrow M \\ \text{Allocations} & \Delta & ::= & r \hookrightarrow M \\ \\ \text{Evaluation} & \mathbb{E} & ::= & [\cdot] \mid \mathbb{E} \mathtt{>>=} M \mid \mathtt{catch}\ \mathbb{E}\ M \\ \text{contexts} & \mathbb{P} & ::= & \mathbb{E}_t \mid (\mathbb{P} \mid P) \mid (P \mid \mathbb{P}) \\ \text{Action} & a & ::= & !c \mid ?c \mid \epsilon \end{array}$$

- Here are the rules for retry:

  ...there are none!
  (apart from an admin transition)...

- In particular, no rule for

  P[atomic retry], Θ -> ...

orElse

$$\begin{array}{rrcl} \text{Thread soup} & P,Q & ::= & M_t \mid (P \mid Q) \\ \text{Heap} & \Theta & ::= & r \hookrightarrow M \\ \text{Allocations} & \Delta & ::= & r \hookrightarrow M \\ \\ \text{Evaluation} & \mathbb{E} & ::= & [\cdot] \mid \mathbb{E} \mathtt{>>=} M \mid \mathtt{catch}\ \mathbb{E}\ M \\ \text{contexts} & \mathbb{P} & ::= & \mathbb{E}_t \mid (\mathbb{P} \mid P) \mid (P \mid \mathbb{P}) \\ \text{Action} & a & ::= & !c \mid ?c \mid \epsilon \end{array}$$

$$\frac{M_1;\ \Theta, \Delta\ \overset{*}{\Rightarrow}\ \mathtt{return}\ N;\ \Theta', \Delta'}{\mathbb{E}[M_1\ \text{`}\mathtt{orElse}\text{`}\ M_2];\ \Theta, \Delta\ \Rightarrow\ \mathbb{E}[\mathtt{return}\ N];\ \Theta', \Delta'}\ (OR1)$$

First branch succeeds

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

First branch retries

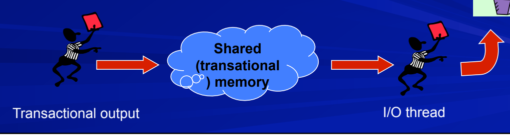
$$\frac{M_1;\ \Theta, \Delta\ \overset{*}{\Rightarrow}\ \mathtt{throw}\ N;\ \Theta', \Delta'}{\mathbb{E}[M_1\ \text{`}\mathtt{orElse}\text{`}\ M_2];\ \Theta, \Delta\ \Rightarrow\ \mathbb{E}[\mathtt{throw}\ N];\ \Theta', \Delta'}\ (OR2)$$

First branch raises exception

Odds and ends

Input/output

- You can't do I/O in a memory transaction (because there's no general way to undo it)
- The STM monad ensures you don't make a mistake about this
- To support transactional I/O:



Transactional input

- Same plan as for output, where input request size is known
- Variable-sized input is harder, because if there is not enough data in the buffer, the transaction may block (as it should), but has no observable effect.
- So the I/O thread doesn't know to get more data :-(
- Still thinking about what to do about this...not sure it matters that much

Progress

- A worry: could the system "thrash" by continually colliding and re-executing?
- No: one transaction can be forced to re-execute only if another succeeds in committing. That gives a strong progress guarantee.
- But a particular thread could perhaps starve.

Is this all a pipe dream?

Surely it's impractical to log every read and write?

1. Do you want working programs or not?
2. Tim built an implementation of TM for Java that showed a 2x perf hit. Things can only improve!
3. We only need to log reads and writes to **persistent** variables (ones outside the transaction); many variables are not.
4. Caches already do much of this stuff; maybe we could get hardware support.
5. ...but in truth this is an open question

No silver bullet

- Transactional memory is fantastic
- But you can still write buggy programs
- But it's like using a high-level language instead of assembly code: whole classes of low-level errors are eliminated
- It's a classic abstraction: a simple interface hides a complex and subtle implementation

What we have now

- A complete implementation of transactional memory in Concurrent Haskell [in GHC 6.4]. Try it!
  http://haskell.org/ghc
- A C# transactional-memory library. A bit clunky, and few checks, but works with unchanged C# [Marurice Herlihy]
- PPoPP'05 paper
  http://research.microsoft.com/~simonpj

Open questions

- Are the claims that transactional memory supports "higher-level programming" validated by practice?
- You can't do I/O within a transaction, because it can't be undone. How inconvenient is that?
- Can performance be made good enough?
- Starvation: a long-running transaction may be repeatedly "bumped" by short transactions that commit. How bad is this?

CML

- CML, a fine design, is the nearest competitor
  - receive :: Chan a -> Event a
  - guard :: IO (Event a) -> Event a
  - wrap :: Event a -> (a->IO b) -> Event b
  - choose :: [Event a] -> Event a
  - sync :: Event a -> IO a
- A lot of the program gets stuffed inside the events => somewhat inside-out structure

CML

- No way to wait for complex conditions
- No atomicity guarantees
- An event is a little bit like a transaction: it happens or it doesn't; but explicit user undo:
  - wrapAbort :: Event a -> IO () -> Event a
- Events have a single "commit point". Non compositional:
  - ??? :: Event a -> Event b -> Event (a,b)