Software transactional memory (STM) is an approach to concurrent programming that replaces locking with atomic transactions which must have all their effects occur without interference from other threads. A key technique is to log each transactions effects, and undo them when interference is detected, then retry the transaction. STM is the subject of much current research, and it has the potential to become a mainstream technique within a few years.

“Transaction memory (TM) is to shared-memory concurrency as garbage collection (GC) is to memory management.” – Dan Grossman

Acknowledgment: thanks to Simon Peyton-Jones for these slides.
The Context

Multicore

Parallel programming essential

Task parallelism
- Explicit threads
- Synchronise via locks, messages

This unit

Data parallelism
Operate simultaneously on bulk data

Generally easier
The state of the art in concurrent programming is 30 years old: locks and monitors. (In Java/C#/F#: synchronised methods, lock statements, Monitor.wait.)

Locks/monitors are fundamentally flawed: it’s like building a sky-scraper out of bananas.

When building on a library, the locks that uses need to be taken into account to avoid deadlocks, etc.

Recent research on STM indicates that it can provide a better foundation: bricks and mortar instead of bananas.
What’s wrong with locks?

A 30-second review:

- **Races**: due to forgotten locks
- **Deadlock**: locks acquired in “wrong” order.
- **Lost wakeups**: forgotten notify to condition variable
- **Diabolical error recovery**: need to restore invariants and release locks in exception handlers

- These are serious problems. But even worse...
Locks are absurdly hard to get right

Scalable double-ended queue: one lock per cell

But watch out when the queue is 0, 1, or 2 elements long!
## Atomic memory transactions

<table>
<thead>
<tr>
<th>Coding style</th>
<th>Difficulty of concurrent queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential code</td>
<td>Undergraduate</td>
</tr>
<tr>
<td>Locks/monitors</td>
<td>Publishable result at international conference</td>
</tr>
<tr>
<td><strong>Atomic blocks</strong></td>
<td><strong>Undergraduate</strong></td>
</tr>
</tbody>
</table>
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. exception thrown inside the `get` code)
One possibility:

- Execute `<code>` without taking any locks
- Each read and write in `<code>` 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
Blocking

atomically <| stm {
    if n_items = 0  then retry()
    else ...remove from queue...
}

- **retry** says “abandon the current transaction and re-execute it from scratch”
- The implementation sees n_items in the log for the transaction, so it waits until n_items changes
- No possibility of lost wake-ups!
atomically <| stm{
    let! x = queue1.getItem()
    return! queue2.putItem(x)
}

- If either `getItem` or `putItem` retries, the whole transaction retries
- So the transaction waits until `queue1` is not empty AND `queue2` is not full
- No need to re-code `getItem` or `putItem`
- (Lock-based code does not compose)
atomically <| stm{
    let! x = queue1.getItem()
    return! orElse queue2.putItem(x)
    queue3.putItem(x)
}

- **orElse** tries two alternative paths
- If the first retries, it runs the second
- If both retry, the whole **orElse** retries.
atomically <| stm{
  let! x = queue1.getItem()
  return!orElse queue2.putItem(x)
               queue3.putItem(x)
}

- So the transaction waits until
  - queue1 is non-empty, AND
  - EITHER queue2 is not full OR queue3 is not full

without touching getItem or putItem
There’s a run-time cost for TM

- For mutable cells, `TVar<'a>` must be used instead of `‘a ref`, to log reads and writes and detect interference.

  ```haskell
  newTVar : ‘a -> TVar<'a>
  readTVar : TVar<'a> -> Stm<'a>
  writeTVar : TVar<'a> -> ‘a -> Stm unit
  ```

However:

- Compiler technology and hardware support can reduce the cost a lot
- A “faster” program that doesn’t work right is useless
- TM allows much finer-grain locking without losing correctness, so it scales better
  - performance may be **better** than when using locks
Results: concurrency control overhead

- Normalised execution time
- Sequential baseline (1.00x)
- Coarse-grained locking (1.13x)
- Fine-grained locking (2.57x)
- Traditional STM (5.69x)
- Direct-update STM (2.04x)
- Direct-update STM + compiler integration (1.46x)

Workload: operations on a binary search tree, 6:1:1 lookup:insert:delete mix with keys 0..65535

Scalable to multicore
Results: scalability

- Coarse-grained locking
- Fine-grained locking
- Traditional STM
- Direct-update STM + compiler integration

<table>
<thead>
<tr>
<th>#threads</th>
<th>Coarse-grained locking</th>
<th>Fine-grained locking</th>
<th>Traditional STM</th>
<th>Direct-update STM + compiler integration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

• Atomic blocks (atomic, retry, orElse) appear to be a real step forward in concurrent programming
• It’s like using a high-level language instead of a low-level one
  • whole classes of low-level errors are eliminated.
• Not a silver bullet:
  – you can still write buggy programs;
  – concurrent programs are still harder to write than sequential ones
  – aimed at shared memory (mostly)
• Very hot research area – expect developments
• An implementation for F# is available (link on the unit page)
• Other implementations: Haskell, C, C++, Java, C#,...
Backup slides
Starvation

• 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.
• No automatic solution can possibly be adequate
No I/O inside transactions

\[
\text{atomic \{ if (x>y) then launchMissiles \}}
\]

- The transaction might see \(x>y\) because it pauses between reading \(x\) and reading \(y\).
- So we must not call \text{launchMissiles} until the transaction commits.
- Simple story: no I/O inside transactions.
• Transactional output is easy:

• Input is a bit harder, because of the need to make sure the transactional input buffer is filled enough