Manchester University Transactions for Scala

Salman Khan
salman.khan@cs.man.ac.uk

MMNet 2011

Transactional Memory

- Alternative to locks for handling concurrency
- Locks
  - Prevent all other threads from accessing shared variables (pessimistic protection)
- Transactional Memory
  - Hope that there will be no collisions (optimistic protection)
  - Records sufficient information to rollback changes in the event of a collision
  - Manages individual transactions so that to the program they appear to happen instantaneously or not happen at all

Software Transactional Memory

- Mechanism requirements
  - Buffer state changes
  - Detect conflicts
  - Resolve conflicts

The Scala Programming Language

- Object Oriented
  - All values are objects
  - Classes, traits
  - Mixin based composition replaces multiple inheritance
- Functional
  - Functions are values
  - Anonymous functions, higher-order functions, the nesting of functions, and support for currying
  - Side effects possible

Why MUTS

Existing STMs do not allow transactions to be added to code without restructuring the code.

We want:

- No difference between transaction syntax and other language constructs
- No restrictions on transaction granularity
- Works with legacy code
- Maintainability

Why MUTS

Existing STMs do not allow transactions to be added to code without restructuring the code.

- User added library calls (tinySTM)

```
for(int i = 0; i < INCREMENT; i++) {
      int tmp = this->value;
      tmp = tmp + 1;
      this->value = tmp;
}
```

Why MUTS

Existing STMs do not allow transactions to be added to code without restructuring the code.

- User added library calls (tinySTM)

```
sigjmp_buf *_e = stm_get_env();
stm_tx_attr_t _a = {0, 0};
sigsetjmp(*_e, 0);
stm_start(_e, &_a);
for(int i = 0; i < INCREMENT; i++) {
      int tmp = (int) stm_load((stm_word_t *)
                                              this->value);
      tmp = tmp + 1;
      stm_store((stm_word_t *) &this->value,

      stm_word_t)tmp);
}
stm_commit();
```

Why MUTS

- Libraries taking functions as first class variables (CCSTM)

```
class IntSet {
  private class Node(val e: Int, next0: Node) {
    val next = Ref(next0)
  }

  private val header = new Node(-1, null)

  def add(e: Int) { atomic { implicit t => loop(e,
   header) } }

  private def loop(e: Int, prev: Node)(implicit t:Txn) {
    val cur = prev.next()
    if (cur == null || cur.e > e)
      prev.next() = new Node(e, cur)
    else if (cur.e != e) loop(e, cur)
  }
}
```

MANCHESTER 1824

The University of Manchester

Why MUTS

- Libraries taking functions as first class variables (CCSTM)

```
class IntSet {
  private class Node(val e: Int, next0: Node) {
    val next = Ref(next0)
  }

  private val header = new Node(-1, null)

  private def loop(e: Int, prev: Node)(implicit t:Txn) {
    val cur = prev.next()
    if (cur == null || cur.e > e)
      prev.next() = new Node(e, cur)
    else if (cur.e != e) loop(e, cur)
  }

  def add(e: Int) { atomic { implicit t => loop(e,
   header) } }
}
```

Why MUTS

- Libraries taking functions as first class variables

```
class IntSet {
  private class Node(val e: Int, next0: Node) {
    val next = Ref(next0)
  }

  private val header = new Node(-1, null)

  private def loop(e: Int, prev: Node)(implicit t:Txn) {
    val cur = prev.next
    if (cur == null || cur.e > e)
      prev.next := new Node(e, cur)
    else if (cur.e != e) loop(e, cur)
  }

  def add(e: Int) { atomic { implicit t => loop(e,
   header) } }
}
```

Why MUTS

- Libraries using annotations (Deuce STM)

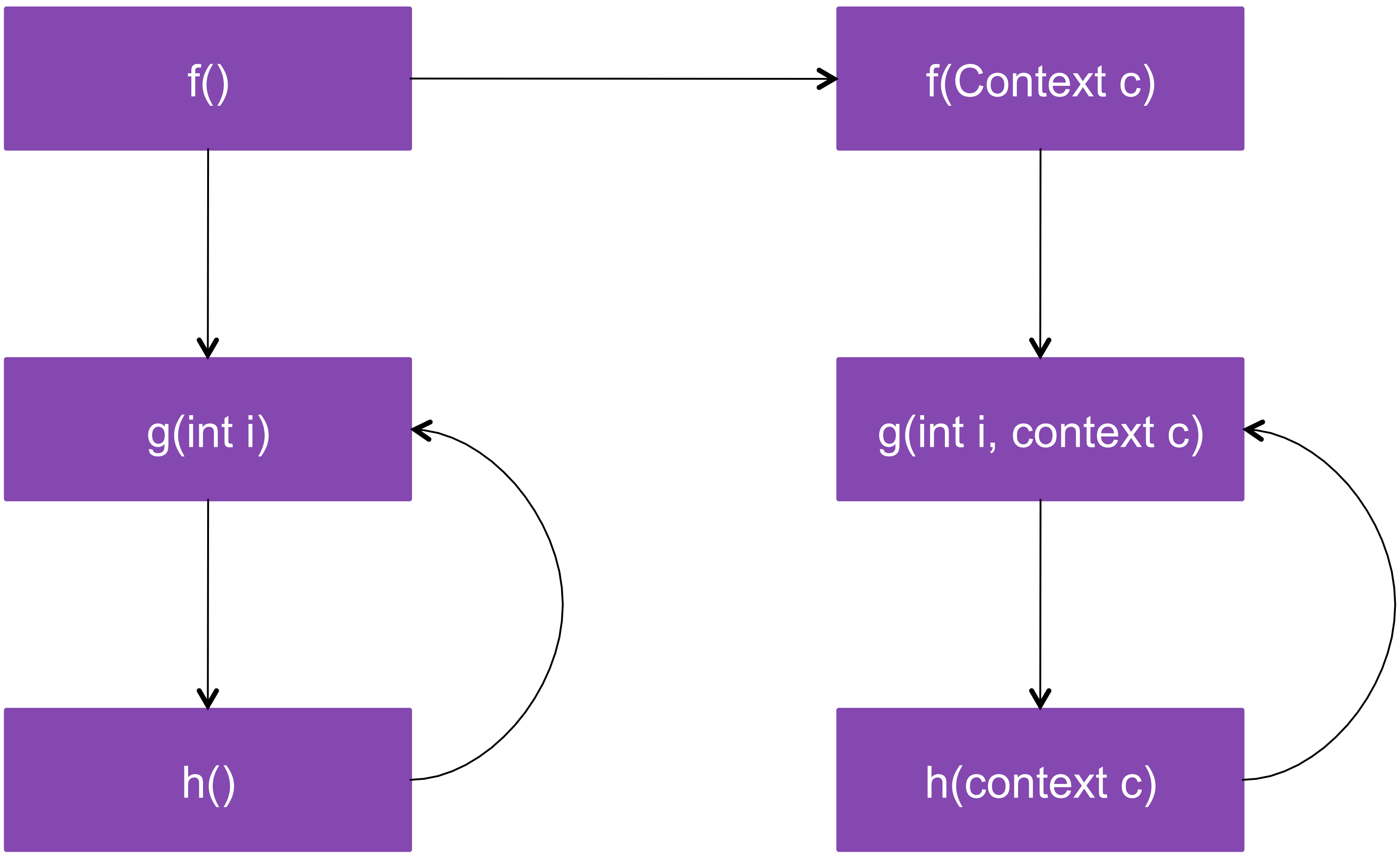
```
@Atomic
public void transfer
              (Account from, Account to, int amount) {
  from.withdraw(amount)
  to.deposit(amount)
}
```

- What happens when just a small part of a method needs to be transactional?
- What if that part of the method uses and or modifies many variables?

Deuce STM

- Implemented using a Java Agent to rewrite the Byte-Code at runtime
- Rewritten code has a duplicate of every method with:
  - A context as an extra method parameter
  - A call to this context for every field load and store
  - The context as an extra parameter to every method call
- Methods marked as @Atomic are replaced with methods that create a context and call the respective duplicate method

Deuce STM



MUTS Syntax

```
atomic {
    body
}
```

```
atomic {
    body
} retry;
```

```
atomic {
    body
} orElse {
    elseBody
}
```

```
atomic(test) {
    body
}
```

```
atomic(test) {
    body
} retry;
```

```
atomic(test) {
    body
} orElse {
    elseBody
}
```

MUTS Syntax

```
class IntSet {
  private class Node(val e: Int, next: Node)

  private val header = new Node(-1, null)

  private def loop(e: Int, prev: Node) {
    val cur = prev.next
    if (cur == null || cur.e > e)
      prev.next = new Node(e, cur)
    else if (cur.e != e) loop(e, cur)
  }

  def add(e: Int) { atomic {  loop(e, header) } }
}
```

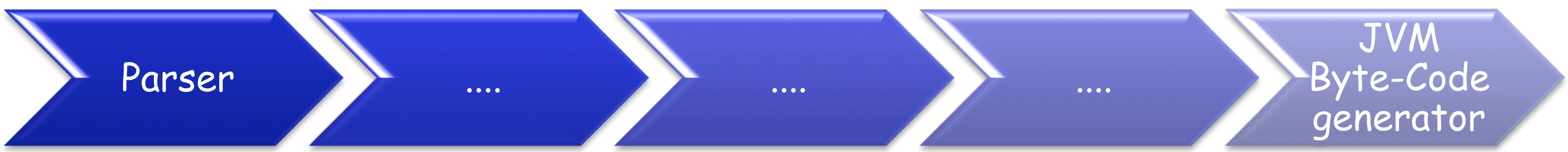
Why MUTS

- Libraries taking functions as first class variables (CCSTM)

```
class IntSet {
  private class Node(val e: Int, next0: Node) {
    val next = Ref(next0)
  }

  private val header = new Node(-1, null)

  private def loop(e: Int, prev: Node)(implicit t:Txn) {
    val cur = prev.next()
    if (cur == null || cur.e > e)
      prev.next() = new Node(e, cur)
    else if (cur.e != e) loop(e, cur)
  }

  def add(e: Int) { atomic { implicit t => loop(e,
   header) } }
}
```

Scala Compiler

- Scala compiler consists of 21 stage pipeline
- The Parser takes a file and returns an abstract syntax tree
- The remaining phases incrementally transform this tree until Byte-Code can be generated



- User constructed phases can be added, but...
  - The data structures are poorly documented
  - Multiple phases may need to be added
  - Adding phases makes the system very sensitive to change

Implementing MUTS

- Two phases
  - Modifications to the parser
  - Java Agent to instrument the Byte-Code

- Both of these work with well defined interfaces
  - Scala
  - JVM Byte-Code

Parser Modifications

- Add new keywords
- When an atomic section is detected:
  - Add the control logic using **existing** tree constructs
  - **Encase the transactional code with an try/catch**
    - Handle exceptions thrown by the body
    - **Mark the code that is transactional**
    - Allow transactions to abort
  - Create a context to store transaction data
  - Copy method variables so that active updates can be used on them by the body

Byte-Code Rewrite

Modifying the Deuce Java Agent

- Add Duplicate Methods
- Detect atomic sections of methods
  - Detect the location of the context
  - Instrument field accesses
  - Augment method calls
  - Remove the marker exception

MANCHESTER 1824

The University of Manchester

Protecting Against Code Reordering

- Compiler reordering means that transactional byte-code needs tagging
- Exception handlers scope is adjusted to reflect reordering
- Creating a special class of exception allows for the tracking of transactional code

Protecting Against Code Reordering

```
try {
  System.out.println("This is the start of the test");

  try{ foo(); }
  catch(IOException e) { System.out.println("IO exception"); }
  catch(NullPointerException e) { System.out.println("Null Pointer
      Exception"); }

  System.out.println("This is the end of the test");
}
catch(Exception e) {  System.out.println("The final Exception"); }
```

Exception table:

```
   from    to  target type
     8     11     14   Class java/io/IOException
     8     11     26   Class java/lang/NullPointerException
     0     43     46   Class java/lang/Exception
```

What Have We Gained?

User:

- Native Constructs
- No change of syntax
- No restrictions on the granularity of transactions
- No restrictions on the use of legacy code
- Interoperable with Java

Implementer:

- Working against well defined interfaces
- Native constructs with minimal changes to the compiler

The University of Manchester

MANCHESTER 1824

Conclusions

- MUTS is a much more intuitive and flexible STM for users
- Interoperable with Java
- Implemented with minimal changes to the parser, and no other changes to the compiler
- Exception handler overcomes code reordering
- This 2-phase approach can be applied to implementing other native constructs
- Part of a suite of Scala STM's at Manchester

Produced as part of the Teraflux Project http://www.teraflux.eu

Daniel.Goodman@Manchester.ac.uk

MANCHESTER
1824
The University of Manchester

Strong vs Weak Isolation

- MUTS, Deuce STM and CCSTM all provide weak isolation
- CCSTM uses the type system which enforces strong isolation **IF** objects are only ever accessed through reference objects
  - Forces all transactional variables to be wrapped in reference objects
  - Possible to use non transactional variables inside transactions
- Inference of transactional types would be better than using the type system