CS162 Week 6

Kyle Dewey

Overview

- STM: semantics and implementation
  - Will cover as much as possible

Software Transactional Memory

Recall STM Semantics

- Based on transactions programmatically specified with `atomic`
- Multiple transactions can occur in parallel
- Transactions are isolated from each other
  - Transaction `T1` cannot see the changes transaction `T2` is making until `T2` commits

At Transaction End

- If some value the transaction depended on has changed elsewhere since it was last used, the transaction fails (abort, rollback...)
- Otherwise, there are no conflicts, and the transaction can commit
  - Commits can be seen elsewhere

STM in miniJS

Mutability

- You are free to use mutability as you see fit
  - Including mutable data structures, `vars`, `while` loops, and so on
- But remember: you may need to synchronize these

Code Base

- Based off of `miniJS` with objects
- No overlap with previous assignments

Threads

testThread0.not

```
var thread in

  thread := (param) =>{
    output param };

tStart(thread, 42)
```

testThread0.not

```
var thread in

  thread := (param) =>{
    output param };

tStart(thread, 42)
```

Output:
42

testThread1.not

```
var a, b, c in
  b := 0;
  a := (param) => {
    output b
  };
  c := (param) => {
    b := b + 1;
    output "inside b"
  };
  tStart(a, {});
  tStart(c, {});
  tStart(a, {});
  tStart(c, {})
```

testThread1.not

```
var a, b, c in
  b := 0;
  a := (param) => {
    output b
  };
  c := (param) => {
    b := b + 1;
    output "inside b"
  };
  tStart(a, {});
  tStart(c, {});
  tStart(a, {});
  tStart(c, {})
```

Output:
???
(Depends on thread scheduling)

testThread2.not

```
var a, b, c in
  b := 0;
  a := (param) =>
    {output b};
  c := (param) =>
    {b := b + 1;
     tStart(a, {})};

  var d in
    d := 0;
    while (d < 20) {
    tStart(c, {});
    d := d + 1
  }
```

testThread2.not

```
var a, b, c in
  b := 0;
  a := (param) =>
    {output b};
  c := (param) =>
    {b := b + 1;
     tStart(a, {})};

  var d in
    d := 0;
    while (d < 20) {
    tStart(c, {});
    d := d + 1
  }
```

Output:
???
(Depends on thread scheduling)

Thread Implementation

- Uses Java's existing `Thread` class
- Different ways to do it
  - Can override `Thread`'s `run()` method
  - Can define a subclass of `Runnable`, which is passed to `Thread`'s constructor

atomic

testAtomic1.not

```
var d in
   d := 0;

   atomic{
      while (d < 40) {
         d := d + 1
      }
   };

   output "Final output is";
   output d
```

testAtomic1.not

```
var d in
   d := 0;

   atomic{
      while (d < 40) {
         d := d + 1
      }
   };

   output "Final output is";
   output d
```

Output:
40

testAtomic2.not

```
var d in
  d := {foo:0};

  atomic{
    while (d.foo < 40) {
      d.foo := d.foo + 1
    }
  };

  output "Final output is";
  output d.foo
```

testAtomic2.not

```
var d in
  d := {foo:0};

  atomic{
    while (d.foo < 40) {
      d.foo := d.foo + 1
    }
  };

  output "Final output is";
  output d.foo
```

Output:
40

testAtomic3.not

```
var d, a in
   d := {foo:0};

   atomic{
      a := {foo:d.foo}
   };

   output "Final output is";
   output a.foo
```

testAtomic3.not

```
var d, a in
   d := {foo:0};

   atomic{
      a := {foo:d.foo}
   };

   output "Final output is";
   output a.foo
```

Output:
0

Threads with `atomic`

prod-consumer.not

testCombine1.not

Implementation Details

- "Come up with a `Log` data structure that registers all the reads and writes that are done inside an atomic block. This data structure should also act as a local store for the atomic section."
- How to make this happen?
  - `interpreter.scala`

Making this Happen

- Could modify everything in the interpreter to use a store
  - This *store-passing style* is used in formal semantics
- Could check to see if we were given a `Log` or not
  - If so, use it. If not, use the global store.
  - RIP-style execution modes
- Many options

Initial Log Contents

- Could use the whole store
- Why is this not a great idea?

Initial `Log` Contents

- Could use whole store
  - Lots of extra memory used; semantically this copies the entire heap
    - Combining is difficult, since we only care about things that were manipulated in a transaction
- Other ideas?

Initial `Log` Contents

- Lazily allocate into the `Log`
  - If the address is in the `Log`, use it
    - If not, look at the global store
    - For new things allocated, put them into the `Log`
- What is wrong with this setup?

Issue

```
var a, thread1, thread2 in
   a := 0;
   thread1 := (param) => {
     atomic {a := 1}
   };
   thread2 := (param) => {
     atomic{
      if (a == 0) {
       a := a + 1
      }}};
  tStart(thread1, 0);
  tStart(thread2, 0);
  // assume both threads finish here
  output a
```

Issue

```
var a, thread1, thread2 in
   a := 0;
   thread1 := (param) => {
     atomic {a := 1}
   };
   thread2 := (param) => {
     atomic{
      if (a == 0) {
       a := a + 1
      }}};
tStart(thread1, 0);
tStart(thread2, 0);
// assume both threads finish here
output a
```

Output:
Either 1 or 2 if we always defer to the global store.

How can this be fixed?

Initial `Log` Contents

- Lazily allocate into the `Log`
  - If the address is in the `Log`, use it
  - If not, look at the global store, and put the address / value mapping from the global store into the `Log`
  - For new things allocated, put them into the `Log`

Commits

- "Modify the global store data structure to handle commits."
- What does this mean?

Commits

- "Modify the global store data structure to handle commits."
- Apply changes from the `Log` into the global store

Modifying Address

- "You may have to modify the `Address` value to ensure proper commits."
- Why?

Modifying Address

```
// the actual store; an Address is an index into this buffer
val store:Buffer[Storable] = Buffer()

def apply( a:Address ): Storable =
  if ( a.loc < store.length ) store( a.loc )
  else throw undefined

def update( a:Address, v:Storable ): Storable =
  if ( a.loc < store.length ) {
    store( a.loc ) = v
    UndefV()
  }
  else throw undefined
```

Modifying Address

```
var a, b, thread1, thread2 in
  thread1 := (param) => {
    atomic {
      a := {foo: 1}
    }
  };
  thread2 := (param) => {
    atomic {
      b := {bar: 2}
    }
  };
  tStart(thread1, 0);
  tStart(thread2, 0)
```

Modifying Address

```
var a, b, thread1, thread2 in
  thread1 := (param) => {
    atomic {
      a := {foo: 1}
    }
  };
  thread2 := (param) => {
    atomic {
      b := {bar: 2}
    }
  };
  tStart(thread1, 0);
  tStart(thread2, 0)
```

Same address, different objects

Alternative to Address Modification

- Use a global counter for addresses
- Each time a new address is needed, get it from this counter
  - `AtomicInteger` may come in handy
- Store becomes a map from `Address`es to `Storable`s in this setup
- You are free to experiment with whichever strategy you want

Synchronization

- "Make sure that the commit process is atomic (i.e no race condition) using thread synchronization techniques."
  - What if we try to commit two `Log`s to the same store at the same time?
  - What if the `Log`s conflict with each other? (i.e. different values for the same address)

Synchronization

- Easy way: use the `synchronized` construct
- Internally uses locks, but this is only a performance thing anyway

```
var a = 5
synchronized {
  a = a + 1
}
```

`synchronized`

- Really a prettified monitor
- `synchronized` is a synonym for `this.synchronized`
- Can also specify explicitly: `foo.synchronized`
  - `foo` is what we will lock on

synchronized

Example

```
var g = 0
val lock = new Object
class Foo {
  // locks on this instance of Foo
  synchronized {
    g = g + 1
  }
  // locks on lock
  lock.synchronized {
    g = g + 1
  }
}
```

Levels of Parallelism

Granularity

`atomic`-level

- Basic idea: only one transaction (atomic block) can execute at a time
- No actual parallelism
- Max score of ‘C’
- Trivial to implement

Commit-level

- Transactions can execute in parallel, but they must commit sequentially
- Parallelism up until commit points, which act as a bottleneck
- Max score of ‘B’
- A tad more complex than `atomic`-level

Address-level

- Transactions that do not conflict with each other can be executed in parallel
- If they do conflict, one will commit and the other will roll back
- Max score of ‘A+’
- Much more difficult

Address-level

- Will require locking individual `Address` objects

```
val toLock: Seq[Address] = ...
toLock(0).synchronized {
  toLock(1).synchronized {
    ...
    toLock(n).synchronized {
      // attempt the commit
    }
  }
}
```

Performing this Locking

- Problem: there can be an arbitrary number of addresses to lock
- We must nest some number of `synchronized` calls
- How to do this?

Performing this Locking

- Could use recursion
  - For each element, lock and recursively call
    - At list end, try the commit
- Could build function with all the `synchronize`s on the fly
  - `foldLeft` / `foldRight` to the rescue

Nested atomic

testAtomicAtomic1

```
var a, b in
  b := 5;

  atomic {
    a := b;
    atomic {
      b := 3;
      a := b
    }
  };

  output a;
  output b
```

testAtomicAtomic1

```
var a, b in
  b := 5;

  atomic {
    a := b;
    atomic {
      b := 3;
      a := b
    }
  };

  output a;
  output b
```

Output:

3

3

Nested `atomic` Implementation

- "When you exit an inner atomic section, commit the changes to the log of the enclosing atomic section."
  - Now `Log`s need to be handle commits in addition to the global store
  - Need to somehow record what to commit to (The global store? A `Log`? If a `Log`, which `Log`?)

tStart **Within** atomic

testThreadAtomic2.not

testAtomicThread1

```
var a, b, thread1 in
  a := 0;
  thread1 := (param) => {
    while (a < 5000) {
      a := a + 1
    }
  };
  atomic {
    tStart(thread1, 0)
  };
  output a
```

testAtomicThread1

```
var a, b, thread1 in
  a := 0;
  thread1 := (param) => {
    while (a < 5000) {
      a := a + 1
    }
  };
  atomic {
    tStart(thread1, 0)
  };
  output a
```

Output:
5000

testAtomicThread2_1.not

testAtomicThread2_1.not

- Output depends on thread schedule
- Final output is always 5000
- Other two values range anywhere from 0 to 5000

Implementing `tStart` within `atomic`

- "Make sure that all the threads within an atomic section complete their execution before performing a commit."
  - Completed means a thread is dead
  - Threads will die on their own when they complete their execution (assuming your `tStart` implementation works correctly)