Software Transactional Memory (STM)

4010-441

Principles of Concurrent System Design

State - Values and Change*

- Imperative Programming
  - Manipulates world (memory) directly.
  - Foundation in the old world of sequential execution (1 thread).
  - Mutexes, locks, state propagation in multi-core - YUCCH!
- Functional Programming
  - More mathematical, "pure" view: Functions take arguments and return values.
  - All values immutable, so concurrency is a non-issue.
- Enter the real world
  - Few programs are merely functions.
  - Need a notion of "state" - now it's this, now it's that.
  - States via immutable values and mutable identities.

*Adapted from Rick Hickey's Work: See http://clojure.org/state

State - Values and Change*

- Identities
  - Stable logical entity associated with different values over time.
  - Examples:
    - Boston Red Sox
    - U. S. citizens
- Identities have ***state*** - the associated ***value*** at a point in time.
- Values ***do not change*** - they are immutable:
  - 42 doesn't change.
  - July 4, 1776 doesn't change.
  - The set of persons on the Yankees in 1927 doesn't change.
    - That is, aggregates are also immutable values
    - The identity Yankees refers to different immutable person sets as trades occur, etc.
- Radically different from ***most*** OO approaches.

*Adapted from Rick Hickey's Work: See http://clojure.org/state

The Clojure / AKKA Approach

- Clearly separate identities from their values over time.
  - Identities are not states; identities have states.
  - And states are true (mathematical values).
  - Identities "appear" to change by assuming different states over time.
  - Program observations of identity's state is a snapshot of an unchanging value.
- Concurrency
  - Refs create identities associated with values.
  - Updating an identity must be done in an ACI transaction.
  - Changes proceed only if the initial snapshot is still valid at commit time.
  - Immutability => efficient creation of new values from old ones.
  - Especially important with composite values.

*Adapted from Rick Hickey's Work: See http://clojure.org/state

Transactions in Akka

```
import akka.stm.*;

final Ref<Integer> ref = new Ref<Integer>(0);

public int counter() {
    return new Atomic<Integer>() {
        public Integer atomically() {
            int inc = ref.get() + 1;
            ref.set(inc);
            return inc;
        }
    }.execute();
}

counter();
// -> 1

counter();
// -> 2
```

Transactions in Akka

```
import akka.stm.*;

final Ref<Integer> ref = new Ref<Integer>(0);

public int counter() {
    return new Atomic<Integer>() {
        public Integer atomically() {
            int inc = ref.get() + 1;
            ref.set(inc);
            return inc;
        }
    }.execute();
}

counter();
// -> 1

counter();
// -> 2
```

Transactions in Akka

```
import akka.stm.*;

final Ref<Integer> ref = new Ref<Integer>(0);

public int counter() {
    return new Atomic<Integer>() {
        public Integer atomically() {
            int inc = ref.get() + 1;
            ref.set(inc);
            return inc;
        }
    }.execute();
}

counter();
// -> 1

counter();
// -> 2
```

Transactions in Akka

```
import akka.stm.*;

final Ref<Integer> ref = new Ref<Integer>(0);

public int counter() {
    return new Atomic<Integer>() {
        public Integer atomically() {
            int inc = ref.get() + 1;
            ref.set(inc) ;
            return inc;
        }
    }.execute();
}

counter();
// -> 1

counter();
// -> 2
```

Transactions in Akka

```
import akka.stm.*;

final Ref<Integer> ref = new Ref<Integer>(0);

public int counter() {
    return new Atomic<Integer>() {
        public Integer atomically() {
            int inc = ref.get() + 1;
            ref.set(inc);
            return inc;
        }
    }.execute() ;
}

counter();
// -> 1

counter();
// -> 2
```

Another Example - Energy Source

(Solved previously using explicit locking in PCJVM Chpt 5)

```
public class UseEnergySource
...

for(int i = 0; i < 10; i++) {
      tasks.add(new Callable<Object>() {
        public Object call() {
          for(int j = 0; j < 7; j++) energySource.useEnergy(1);
          return null;
        }
      });
```

…

useEnergy Transaction

```
public class EnergySource {
  private final long MAXLEVEL = 100;
  final Ref<Long> level = new Ref<Long>(MAXLEVEL);
  final Ref<Long> usageCount = new Ref<Long>(0L);
…
```

```
public boolean useEnergy(final long units) {
    return  new Atomic<Boolean>() {
      public Boolean atomically() {
        long currentLevel = level.get();
        if(units > 0 && currentLevel >= units) {
          level.swap(currentLevel - units);
          usageCount.swap(usageCount.get() + 1);
          return true;
        } else {
          return false;
        }
      }
    }.execute();
  }
```

Summary

- Transactions may end up retried several times.
  - Must be *idempotent*.
  - "Unexpected" retries.
- Efficient immutable datastructures (via "smart sharing")
  - TransactionalMap
  - TransactionalVector