Language Support for Concurrency

December 14, 2011

1. Software Transactional Memory

based on Hoare's Conditional Critical Regions

```
public int get() {
  atomic (items != 0) {
    items --;
    return buffer[items];
  }
}
```

What's missing:

what is the data protected ?

when is a blocked thread released ?

What does STM offer ?

dynamically non-conflicting executions can operate concurrently

CCR conditions re-evaluated only on a shared update

non-blocking implementation (prevents deadlock, priority inversion)

Goals: minimal restrictions for code enclosed in `atomic`

low implementation overhead *outside* CCRs

Sample implementation [Harris,Fraser - OOPSLA03]

```
void STMStart()
void STMAbort()
boolean STMCommit()
boolean STMValidate()
void STMWait()
```

Sample implementation - Clojure refs

Clojure: dynamic language (Lisp dialect) compiled to Java bytecode

Refs allow shared use of mutable storage locations
- mutation of location allowed only in transaction

2. Persistent Data Structures

All values are immutable
including composite ones

change is actually a function that returns a new value
old value still exist and can be used

To change state:
construct new compound value
change the reference
⇒ can be done much easier

3. Actors

Everything is an actor.

Actors may
- send messages to other actors
- create new actors (a finite number)
- designate behavior for next message received

Similar to
- Smalltalk (send messages)
- process algebras

4. Dataflow

Examples in Oz [Wikipedia]

– Programs wait until variables bound to values

```
thread
   Z = X+Y     % waits until both X and Y are bound.
   {Browse Z}  % shows the value of Z.
end
thread X = 40 end
thread Y = 2 end
```

– immutable values (cannot change while bound)

5. Tuple Spaces

[after vanRoy and Haridi]

`{TS write(T)}` adds tuple `T` to the tuple space.

`{TS read(L T)}` waits for tuple with label `L`.

`{TS readnb(L T B)}` no wait, returns with `B=true/false`

can be implemented with a lock, a dictionary and a concurrent queue

Concurrent Queue in Linda

```
fun {NewQueue}
  X TS={New TupleSpace init}
  proc {Insert X}
  N S E1 in
    {TS read(q q(N S X|E1))}
    {TS write(q(N+1 S E1))}
  end
  fun {Delete}
  N S1 E X in
    {TS read(q q(N X|S1 E))}
    {TS write(q(N-1 S1 E))}
    X
  end
in
  {TS write(q(0 X X))}
  queue(insert:Insert delete:Delete)
end
```