Fundamental Structures of Computer Science II

15-212-ML Fall 1998

CONCURRENCY

Concurrency

Operations in a program are *concurrent* if they $could$ be executed in parallel.

Concurrent programs are inherently *nondeterministic*.

Concurrent programming languages provide abstraction mechanisms for concurrency, with less overhead than using system-level processes.

Forcing Sequential Operations

Being forced to have sequential operations can have disatrous effects for inherently concurrent applications.

Example: the Unix `xrn` "lost connection to remote news server" feature.

Natural Places for Concurrent Programming

- Interactive systems such as window managers and GUIs. User interaction can be complex.
- A spreadsheet might provide an editor, a window for graphical displays of data, and even a speech interface.
- Many applications have “deadtime” between input events when running.
- If application is output intensive, concurrency can make it responsive to input.
- Distributed systems – each node has its own state and control flow. Anything involving a network.
- Client-server protocols.

Language Support

Abstraction is essential to writing and maintaining software

When it comes to concurrency, most languages do not provide useful abstractions

Raw process creation using **fork**() is provided in C

Reasoning About Languages

In SML there is a clean semantics that aids our reasoning

$$\frac{\eta \vdash e_1 \hookrightarrow \mathit{true} \qquad \eta \vdash e_2 \hookrightarrow v}{\eta \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_2 \hookrightarrow v}$$

With references, reasoning becomes more complicated

```
let
   val memo = ref (fn () => raise Impossible)
   fun s'() =
      let
         val r = s()
      in
         memo := (fn () => r);
         r
      end
in
   memo := s';
   fn () => (!memo)()
end
```

Concurrent Programming is Hard

Sequential programs are deterministic

```
let
  val x = ref 0
in
  x := !x + 1; x := !x + 2;
  print (!x)
end
```

Reasoning required when programming with concurrency is much more complicated

```
let
  val x = ref 0
in
  (x := !x + 1) || (x := !x + 2);
  print (!x)
end
```

Ingredients for Concurrency

- A mechanism for introducing sequential *threads* of control, or *processes*
- A way for processes to *communicate*
- A mechanism for processes to *synchronize* to restrict order of execution and limit nondeterminism

Flavors of Concurrent Language

Shared-memory

- Rely on imperative features for interprocess communication
- Separate synchronization primitives to control access to shared state.
- Example: Java

Message-passing

- Unified mechanism for synchronization and communication
- Example: CML

Interference

```
let
  val x = ref 0
in
  (x := !x + 1)  || (x := !x + 2);
  print (!x)
end
```

Interference can occur when two processes are accessing *critical regions* of code where assignments are made

Synchronization provides the mechanism for avoiding interference, by allowing a process to obtain a *lock* in a critical region.

Deadlock

Consider the following schematic of processes P and Q:

P: acquire A; acquire B; compute; release B; release A;
Q: acquire B; acquire A; compute; release A; release B;

This can *deadlock* with P holding the lock on A and Q holding the lock on B.

Livelock

Now consider P and Q defined as:

```
P:                                 Q:

l: acquire A;                      l: acquire B;
if (B is held)                     if (A is held)
   then (release A; goto l)           then (release B; goto l)
   else acquire B                     else acquire A
compute                            compute
release B; release A               release A; release B
```

What might happen here?

Threads in Java

```
public interface Runnable {
    public abstract void run();
}

public class Thread implements Runnable {
    public Thread();
    public Thread(String name);
    ...
    public void run();
    public void start()
        throws IllegalThreadStateException;
    public final void stop();
    public final void suspend()
        throws SecurityException;
    public final void resume()
        throws SecurityException;
    ...
}
```

Threads in Java (cont)

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }
    public void run() {
        // compute primes ≥ minPrime
        ...
    }
}

PrimeThread p = new PrimeThread(101);
p.start();
```

Threads in Java (cont)

- A *synchronized* method acquires a lock before it executes.
- If a variable is ever to be assigned by one thread and used by another, all accesses to that variable should be *synchronized*.
- Locking is carried out using *monitors* in the JVM.
- Synchronization makes methods and blocks *atomic*.
- Deadlocking is not prevented.

Threads in Java (cont)

```
public class Box {
  private  Object boxContents;
  public synchronized Object get() {
     Object contents = boxContents;
     boxContents = null;
     return contents;
  }
  public synchronized boolean put(Object contents){
     if (boxContents != null) return false;
     boxContents = contents;
     return true;
  }
}
```

Shared Memory Model

- Promotes efficiency, but not correctness
- Requires "defensive programming" (protect your data from interference).
- Poor fit with value-oriented programming (ML)

Message-Passing Languages

- Processes communicate by sending messages across channels.
- The communication may either be *blocking* (or synchronous) or *non-blocking* (asynchronous) [4 possibilities]
- Mailbox metaphor: In asynchronous send, once the letter is in the mailbox, the sender can proceed with other tasks.
- Telephone metaphor: In synchronous send, the sender is tied up until his message is received.
- *Synchronous message passing is easier to reason about.*

Basic CML Primitives: Threads

A *thread* is a CML process. Initially, there is a single thread but more can be created using `spawn`:

```
val spawn : (unit -> unit) -> thread_id
```

- Creates a new thread to evaluate the function.
- The number of threads is unbounded.
- Since threads are represented by ML values, their storage can be recycled by the garbage collector.

Basic CML Primitives: Channels

For threads to be useful, we need ways to communicate between them.

For communicating values of type `'a`, CML provides

```
type 'a chan
```

The *send* and *receive* operations are

```
val recv : 'a chan -> 'a
val send : ('a chan * 'a) -> unit
```

Message passing is synchronous

Basic CML Primitives: Channels (cont)

When a thread executes a `send` or `receive` on a channel, it blocks until some other thread offers the complementary communication.

The message is then passed from sender to receiver, and the threads continue.

Message passing involves both communication and synchronization

Example: Reference Cells

As a simple example of channels, as well as client-server protocols, we'll implement the following signature for mutable cells:

```
signature CELL = sig
  type 'a cell
  val cell : 'a -> 'a cell
  val get  : 'a cell -> 'a
  val put  : 'a cell * 'a -> unit
end
```

Example: Reference Cells (cont)

```
structure Cell :> CELL = struct
  datatype 'a request = GET | PUT of 'a
  datatype 'a cell = CELL of
   {reqCh:'a request CML.chan, replyCh:'a CML.chan}

  fun get (CELL{reqCh, replyCh}) =
        (CML.send (reqCh, GET); CML.recv replyCh)

  fun put (CELL{reqCh, ...}, x) =
          CML.send (reqCh, PUT x)
  ...
end
```

Example: Reference Cells (cont)

```
structure Cell :> CELL = struct
  datatype 'a request = GET | PUT of 'a
  datatype 'a cell = CELL of
   {reqCh:'a request CML.chan, replyCh:'a CML.chan}

  fun cell x =
     let
       val reqCh = CML.channel()
       val replyCh = CML.channel()
       fun loop x =
        (case (CML.recv reqCh)
         of GET => (CML.send (replyCh, x); loop x)
          | (PUT x') => loop x')
     in
       CML.spawn (fn () => loop x);
       CELL {reqCh = reqCh, replyCh = replyCh}
     end

  fun get (CELL{reqCh, replyCh}) =
        (CML.send (reqCh, GET); CML.recv replyCh)
  fun put (CELL{reqCh, ...}, x) =
          CML.send (reqCh, PUT x)
end
```

Example: Reference Cells (cont)

This is an example of *client-server* style of concurrent programming.

Why is the implementation correct? Why can't multiple clients interfere with each other, and receive each other's messages, since they are communicating on the same channel?

Example: Streams

```
fun nats start =
   let
      val ch = CML.channel()
      fun count i = (CML.send(ch, i); count(i+1))
   in
      CML.spawn (fn () => count start);
      ch
   end
```

Sieve of Eratosthenes

```
fun filter (p, inCh) =
  let
    val outCh = CML.channel();
    fun loop () =
      let
        val i = CML.recv inCh
      in
        if ((i mod p) <> 0)
          then CML.send (outCh, i)
          else ();
        loop()
      end
  in
    CML.spawn loop;
    outCh
  end
```

Sieve of Eratosthenes (cont)

```
fun sieve() =
  let
    val primes = CML.channel()
    fun head ch =
      let
        val p = CML.recv ch
      in
        CML.send (primes, p);
        head (filter (p, ch))
      end
  in
    CML.spawn (fn () => head (nats 2));
    primes
  end
```

How do these streams differ from those we built using suspensions?

Dataflow Networks

This is an example of a *dataflow network*, where the data moves from one process to another.

Simplest example is *pipeline* where data moves along a chain of processes.

The Fibonacci Network

[Picture of network here]

Fibonacci Network

```
fun fibchannel () =
  let
    val outCh = CML.channel()
    val  c1 = CML.channel() and c2 = CML.channel()
    and c3 = CML.channel()
    val  c4 = CML.channel() and c5 = CML.channel()
  in
    delay (SOME 0) (c4, c5);
    copy (c2, c3, c4);
    add (c3, c5, c1);
    copy (c1, c2, outCh);
    CML.send (c1, 1);
    outCh
  end
```

Fibonacci Network (cont)

Implemented using infinitely looping threads:

```
val forever : 'a -> ('a -> 'a) -> unit
```

```
fun forever init f =
   let
      fun loop s = loop (f s)
   in
      ignore (CML.spawn (fn () => loop init))
   end
```

Fibonacci Network (cont)

```
fun add (inCh1, inCh2, outCh) =
   forever ()
     (fn () =>
        CML.send (outCh,
          (CML.recv inCh1) + (CML.recv inCh2)))

fun delay init (inCh, outCh) =
   forever init
     (fn  NONE =>  SOME(CML.recv inCh)
       | (SOME x) => (CML.send(outCh, x); NONE))

fun copy (inCh, outCh1, outCh2) =
   forever ()
     (fn () =>
       let val x = CML.recv inCh
       in CML.send(outCh1, x); CML.send(outCh2, x)
       end)
```

Fibonacci Network: A Bug

```
fun fibchannel () =
   let
      val outCh = CML.channel()
      val  c1 = CML.channel() and c2 = CML.channel()
      and c3 = CML.channel()
      val  c4 = CML.channel() and c5 = CML.channel()
   in
      delay (SOME 0) (c4, c5);
      copy (c2, c4, c3);
      add (c3, c5, c1);
      copy (c1, c2, outCh);
      CML.send (c1, 1);
      outCh
   end
```

Fibonacci Network: A Bug

So, deadlock can be very hard to guard against.

The language can help: introduce nondeterminism in the order of blocking communication operations.

Leads us to *events*

Events

CML provides the type constructor for abstract synchronous operations:

type `'a event`

An α `event` returns a value of type α when it is synchronized on.

Events (cont)

Events allow us to manipulate multiple concurrent operations without explicitly synchronizing on any specific one.

Enables us to reason about and manipulate the nondeterminism in a program

```
select [
   recvEvt chan1,
   recvEvt chan2
]
```

The thread blocks at the `select` statement, waiting to receive a message from *either* channel

Basic Event Operations

```
val sendEvt : ('a chan * 'a) -> unit event
val recvEvt : 'a chan -> 'a event
val wrap    : 'a event * ('a->'b) -> 'b event
val select  : 'a event list -> 'a
```

The `wrap` function enables us to build up event values

Events in Java

Events are "broadcast" and various classes of events can be subscribed to by "listening" to the broadcast channel

```
public abstract interface MouseListener extends EventListener {
    public abstract void mouseClicked  (MouseEvent e);
    public abstract void mouseEntered  (MouseEvent e);
    public abstract void mouseExited   (MouseEvent e);
    public abstract void mousePressed  (MouseEvent e);
    public abstract void mouseReleased (MouseEvent e);
}
```

Does not support selection from a dynamically changing set of events.

Example: Reference Cells Using Events

```
datatype 'a cell = CELL of
  {getCh:'a CML.chan, putCh:'a CML.chan}

fun get (CELL{getCh, ...}) = CML.recv getCh
fun put (CELL{putCh, ...}, x) = CML.send (putCh, x)

fun cell x =
   let
     val getCh = CML.channel()
     val putCh = CML.channel()
     fun loop x = select [
          wrap (sendEvt(getCh, x), fn () => loop x),
          wrap (recvEvt putCh, loop)
         ]
   in
     CML.spawn (fn () => loop x);
     CELL {getCh = getCh, putCh = putCh}
   end
```

Concurrency and Computation

Concurrency can enable more powerful computation.

Recall that there is no way to compute the disjunction of two semi-decision procedures in SML.

Using *parallel* computation, however, we can.

How?

Parallel Or & Semi-Decision Procedures

```
fun parallelOr (p, q) x =
  let
    val chp = CML.channel()
    val chq = CML.channel()
    fun decide pred ch =
      if pred(x) then CML.send(ch, true)
      else ()
  in
    CML.spawn (fn () => decide p chp);
    CML.spawn (fn () => decide q chq);
    CML.select [
          CML.recvEvt chp,
          CML.recvEvt chq
        ]
  end
```

Reference/Acknowledgement

Most of the material presented here has been adapted from the manuscript *Concurrent Programming in ML* by John Reppy.