Lecture 28: Concurrent ML

CSC 131
Spring, 2019

Kim Bruce

Parallelism in Functional Langs

- Extremely natural.
  - When evaluating f(exp1,exp2,exp3), why not evaluate all in parallel?
  - Experts suggest using immutable data for parallelism to avoid race conditions
  - If no side effects then order of evaluation not relevant. No race conditions!!!
- What could go wrong?

Concurrent ML

- Designed by John Reppy, now U. of Chicago
- Shared memory poor fit for functional langs
  - Message passing
- Threads share dynamically created channels carrying values of arbitrary type
- Threads synchronize by send and receive on channels.

Threads in CML

- New thread created using spawn:
  - val spawn: (unit → unit) → thread_id
- New thread applies function argument to ( ) to begin execution.
  - Terminates when function returns.
  - storage is garbage collected
- Returns unique id for child thread to parent

Channels

- Channels carry values of arbitrary type
  - type ‘a chan
- Created by:
  - val channel: unit → ‘a chan
  - type inferred by use, only carry values of type ‘a
- Unused channels are garbage collected.

Synchronous Send & Receive

- Synchronous ops:
  - val send: ‘a chan * ‘a → unit
  - val recv: ‘a chan → ‘a
- Send blocks its thread until message received
- Recv blocks until matching send occurs
- Synchronize w/ *rendezvous*.

Synchronizing

```
fun child_talk() = let
      val ch = channel()
      val pr = CIO.print
   in
      spawn(fn() => (pr “begin 1\n”; send(ch,0);
                                     pr “end 1\n”));
      spawn(fn() => (pr “begin 2\n”; recv ch;
                                     pr “end 2\n”));
   end;
```

results in

```
begin 1 }  either order
begin 2
end 1   }  either order
end 2
```

Emulate Cell as Thread

- Mutable cell as server accepting requests to set and get value
  - I.e. cell is *secretly* a pair of channels - for request and reply to queries

```
signature CELL = sig
  type ‘a cell
  val new: ‘a -> ‘a cell
  val get: ‘a cell -> ‘a
  val set: ‘a cell * ‘a -> unit
end
```

Mutable Cells as Threads

```
structure Cell :> CELL = struct
   datatype ‘a request = GET | PUT of ‘a

   datatype ‘a cell =
      CELL of {reqCh: ‘a request chan, replyCh: ‘a chan}

   fun new x = ...

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

   fun set (CELL{reqCh, replyCh},x) = (send(reqCh, PUT x))
end
```

More

```
fun new x =
   let
      val reqCh = channel()
      val replyCh = channel()
      fun server x =
        (case (recv reqCh) of
           GET => (send(replyCh,x); server x)
         | PUT x’ => server x’)
   in
      (spawn (fn () => server x);
      CELL {reqCh = reqCh, replyCh = replyCh})
   end
```

Observations

- No mutable storage used. State is in recursion
- Request/reply protocol hidden behind CELL abstraction. Can’t accidentally recv from replyCh w/out first sending GET request.
- Synchronous send ensures cell ops are *atomic*.

Streams as Threads

- Streams can be viewed as suspended computations, producing values only on demand.
- Emulate as threads using send and recv
  - dataflow network

Streams

- Stream of natural numbers

```
fun nats_from start =
  let
    val ch = channel()
    fun loop i = (send(ch,i); loop(i+1))
  in
    spawn(fn () => loop start); ch
  end
```

- recv's on returned channel yield successive nats, starting w/ "start"

Summary

- Synchronous fragment of CML provides
  - multiple threads of control
  - Dynamically-allocated communication channels
  - Synchronous send and receive on channels
- Next: Asynchronous CML and first-class events.

More Primitives

- `'a event`: represent synchronous operations that return a value of type 'a when sync takes place
  - `sync     : 'a event → 'a`
  - `recvEvt: 'a chan → 'a event`
  - `sendEvt: ('a chan * 'a) → unit event`
- Define:
  - `fun recv(ch) = sync(recvEvt(ch))`
  - `fun send(ch,v) = sync(sendEvt(ch,v))`
- Allow creation of more complex events and then syncing on them!

Using Events

- More primitives:
  - `choose: 'a event list → 'a event`
  - `wrap: ('a event * ('a → 'b)) → 'b event`
  - `forever: 'a * ('a → 'a) → unit`
    - `forever b f` computes `f(b),f(f(b))`, ..., for side effects
- Define select function:
  - `fun select(evs) = sync(choose(evs))`

Example

- Repeatedly read from channels in either order:

```
fun add(in1, in2, out) = forever()(fn() =>
  let
    val (a,b) = select [
      wrap(recEvt in1, fn a => (a,recv in2)),
      wrap(recEvt in2, fn b => (recv in1,b))
    ]
  in
    send(out,a+b)
  end
)
```

Asynchronous Write

```
fun asyncWrite(inp, out1,out2) = forever()(fn() =>
  let
    val x = recv inp
  in
    select [
      wrap(sendEvt(out1,x), fn () => send(out2,x)),
      wrap(sendEvt(out2,x), fn () => send(out1,x))
    ]
  end
```

CML

- Supports synchronous and asynchronous message sends (using sync and events)
- Many more features built-up in libraries.

Comparing Mechanisms

- Shared memory concurrency
  - Semaphores & locks very low level.
  - Monitors are passive regions encapsulating resources to be shared (mutual exclusion). Cooperation enforced by wait and signal statements.
- For best results
  - Maximize number of variables accessible by only a single thread
  - Use immutable values wherever possible
  - Use locks or higher-level constructs to avoid data races for all other variables.

Comparing Mechanisms

- Distributed Systems
  - Everything active in Ada tasks (resources and processes) and in Scala actors
  - CML primitives support synchronous and asynchronous communications.
- Problems
  - Must worry about mailboxes filling w/asynchronous message passing.
  - Data in messages must be copied (OK if immutable)

Why PLs?

- Deeper understanding of principal features of programming languages
- Explore design space of language features
- Different ways of thinking about programming
- Languages change regularly over time
  - Evaluate suitability for intended purpose
  - Understand choices in design space
- Implementation issues & efficiency

Topics in Recent PL Meetings

- Fixing/Replacing Javascript (types?)
- Gradual types
- Providing security (esp for mobile devices)
- New languages: Go, Dart, Rust, ...
- Concurrency

Class Topics

- Syntax (formal) and semantics (informal and formal) of programming language concepts.
  - Structure of compilers / interpreters.
  - Binding time.
  - Variables: static vs. dynamic scope, lifetime, l-values vs. r-values.
- Run-time structure of programming languages.
  - Allocation of storage at run-time: stack & heap.
  - Parameter passing mechanisms.
  - Storage reclamation - explicit & automatic

Class Topics

- Lambda calculus & functional languages
- OOLs
  - Subtype vs. inheritance (mixins, too)
  - implementation
- Types in programming languages.
  - Available types and their representation.
  - Issues in type-checking & type-inference.
  - Static vs. dynamic type-checking.
  - Problems with pointers.

Class Topics

- Abstract data types
  - Information hiding, encapsulation
  - Modules
- Control structures
  - iterators, exception handling, and continuations.
- Polymorphism - implicit and explicit.
- Concurrency & Parallelism
  - Shared memory, semaphores, locks, monitor
  - Distributed systems, message passing
    - Synchronous vs asynchronous

Final Exam

- Comprehensive, but heavy emphasis on last half.
  - 24 hour take-home.
  - Pick up from CS office, 2nd floor Edmunds between 8:30 a.m. - noon and 1 p.m. - 4:30 p.m.
  - Available by Monday at 9 a.m.
  - Due 24 hours after pickup, but Wednesday at midnight at latest.
  - Submit via submit web page.

How To Study

- Make sure can do all homework on your own
  - May be at disadvantage if relied too much on partners!!
- Review problems at end of chapters
  - Lecture notes and in-class notes key
- Study in groups ahead of time.
  - Don't assume you can learn what you need in 24 hours of exam — you can't!