15-150 Lecture 25:
Effects and Parallelism

Lecture by Stefan Muller

June 26, 2018

1 I/O

We alluded to input and output a bit during yesterday’s lecture, when we defined the game Referee and the Human player, but let’s talk about it a bit more. Standard ML comes with a bunch of structures called the Basis Library; one of these is TextIO, which implements the signature:

signature TEXT_IO =
sig
  val stdIn : instream
  val openIn : string -> instream
  val inputLine : instream -> string option
  val print : string -> unit

  (* ... *)
end

instream represents something you can read from. stdin (standard in, or the terminal) is one such thing; another is openIn "filename". inputLine reads a newline-terminated string from such a stream. print displays characters to standard out. TextIO also lets you read in from files.

2 Effects and Parallelism

Recall from yesterday that side effects or just effects are things an expression might do instead of, or in addition to, returning a value. Effects we’ve considered are non-termination, I/O (printing, reading input, sending something over a network, . . . ), raising exceptions and, as of yesterday, mutation.

Parallelism is about efficiency—running code on multiple processors at once. There are different ways to achieve parallelism. In this course, we have considered deterministic parallelism, where you make sure that the behavior of a program is independent of the schedule. Indeed, we have reasoned about the extensional behavior of expressions without ever discussing how we evaluate them!

*based partially on notes by Brandon Bohrer and Michael Erdmann
An alternative is *concurrency*. In concurrency, it is common to embrace nondeterminism and effects. Indeed, these are often the whole point of concurrency: you *want* it to be clear to the outside world that multiple things are happening at the same time (e.g. that your web browser is loading a page while it is also listening to see if you click the stop button).

But for deterministic parallelism, we need to consider, for each effect, whether it may be used in parallel code while preserving determinacy. We consider the four aforementioned effects here.

### 2.1 Non-termination

Is non-termination dangerous in parallel code? What happens if you run

\[
\text{Seq.map } f \langle 1, 2 \rangle
\]

where \(f\) 1 terminates but \(f\) 2 loops. Then no matter the schedule, the expression loops: whether it works on \(f\) 1 first and then \(f\) 2, or vice versa, it eventually has to do all of the work in \(f\) 2, and so it will loop. In the kind of parallelism we’ve discussed, programs always do all of the same work no matter their schedule—some schedules might just finish more quickly than others.

\[
\begin{array}{c}
/ \ \\
/ \ \\
f 1 \ f 2
\end{array}
\]

### 2.2 Exceptions

Similar problems come up with exceptions. For example, in

\[
\text{Seq.map } (\text{fn } x = \text{raise Fail (Int.toString } x)) \ (\text{seqFromList } [1, 2, 3])
\]

what exception do you see?

\[
\begin{array}{c}
/ \ | \ \\
/ \ | \ \\
f 1 \ f 2 \ f 3
\end{array}
\]

If the program just raised whichever exception happened to actually be raised first, then you might see any of *Fail "1"* or *Fail "2"* or *Fail "3"* depending on how the program was scheduled. This is bad, because the program no longer has a well-defined meaning.

We can fix this by defining the meaning to be that the program raises the “leftmost” exception that gets raised—i.e. it agrees with the sequential semantics where *map* is evaluated left-to-right. This means that, at the join-point of a fork-join parallelism construct, like *map*, the program must wait for all tasks to complete, and choose the appropriate exception to propagate.
In fact, even if \texttt{Seq.map} did the wrong (ill-defined) thing, we could implement our own \texttt{emap} that does the right thing. The reason we can do this is that you can handle an exception and turn it into an option—this lets us implement the necessary join-point logic ourselves. It illustrates how the join-point logic in the implementation of \texttt{Seq.map} works.

First, we define a result to be either a success (carrying a value), or a failure (carrying an exception packet, which has type \texttt{exn}):

\begin{verbatim}
datatype 'a result = Success of 'a | Failure of exn
\end{verbatim}

Next, we can write generic operations for converting a function that may raise an exception into a function that never does, but optionally returns an exception packet instead of a result, and back:

\begin{verbatim}
fun total (f : 'a -> 'b) : 'a -> 'b result =
  fn x => Success (f x) handle e => Failure e

fun partial (f : 'a -> 'b result) : ('a -> 'b) =
  fn x => case f x of Success v => v | Failure e => raise e
\end{verbatim}

Next, we implement the join-point logic:

\begin{verbatim}
fun leftToRight (s : 'b result Seq.seq) : 'b Seq.seq result =
  Seq.mapreduce (fn x => case x of
    Success v => Success (Seq.singleton v)
  | Failure e => Failure e)
    (Success (Seq.empty()))
    (fn (x,y) => case (x,y) of
      (Success v1 , Success v2) =>
        Success (Seq.append (v1,v2))
      | (Failure e , _) => Failure e
      | (Success _ , Failure e) => Failure e)
    s
\end{verbatim}

The second anonymous function implements a binary join: If both sides come back as a success, the result is a success. If the left side raises an exception, that exception gets raised. If the left side succeeds and the right raises, only then does the right exception get raised.

Finally, we chain these functions together to wrap \texttt{map}: compute an explicit success/failure; then do the join-point logic; and finally raise the appropriate exception if one was raised during the map:

\begin{verbatim}
fun emap (f : 'a -> 'b) : 'a Seq.seq -> 'b Seq.seq =
  partial (leftToRight o (Seq.map (total f)))
\end{verbatim}

What is the span? Unfortunately, this adds a logarithmic factor to \texttt{Seq.map}, because the exception propagates in a tree-like manner. With this function, you are free to use exceptions in parallel code, and you will see a deterministic, left-to-right semantics for which exception gets raised.
2.3 I/O

If you evaluate:

List.map (fn x => (print (Int.toString x); x + 1)) [10,20,30]

The value is [11,21,31], as you would expect, and you see 102030.

What about:

Seq.map (fn x => (print (Int.toString x); x + 1))
(seqfromlist [10,20,30])

The value is <11,21,31>, but what do you see? Recall that the cost graph looks like:

```
  .
 / | \
/  |  \
f 10 f 20 f 30
\  |  /  \
\  |  /  
  .
```

102030
201030
302010
320010
...

Each of these corresponds to one of the possible schedules. That is, the order in which things get printed depends on the order in which things get executed! The meaning of a program that uses input/output in parallel code is not well-defined, but rather depends the schedule. Moreover, as the last possibility shows, it might not even correspond to any particular execution of what the program-level tasks are: there is no guarantee that, when you print "12", the two characters get printed out together, without something else happening in between!

This situation, in which the behavior of the program depends on the scheduling, is called a race condition. Race conditions are bad because they mean you need to think about parallelism when reasoning about the behavior of your program, not just the efficiency. This is much harder.

One way to avoid these pitfalls is to keep I/O out of parallel code by pushing effects to the periphery. At the end of the day, you want your program to read and print stuff—that’s why you run it. But in our implementation of Nim, the effects were at the “outside”—in the referee’s loop, and in the human player. In the interesting, computationally intensive code, like the game tree search and the board update, we programmed functionally, in order to exploit parallelism.

2.4 Mutation

Let’s return to our bank account example from yesterday.
val acct = ref 100
val _ = Seq.tabulate (fn 0 => acct := (!acct) + 100
| 1 => acct := (!acct) - 50
| _ => raise Seq.Range)

The deposit and withdrawal are run in parallel. But the deposit code actually reads the contents of \( r \) first, then writes the updated value. Similarly the withdraw code reads the contents of \( r \) before writing the updated value. In each case the updated value gets computed as an increment or decrement of whatever value was read. So it is possible (depending on the process scheduling) for the two reads to happen before either of the two writes. And then the order in which the writes get scheduled will determine who gets to set the final contents of \( r \). The possible final values for the contents of \( r \) include 50, 150, and 200. This is probably NOT what a bank (or even an account holder) would desire. It is also another example of a race condition.

We could avoid this race condition, as with I/O above, by simply pushing mutation effects to the periphery of the program. But that’s not always possible or beneficial. So there are two other options we’ll consider as well:

- Use locks to enforce atomic execution of updates to shared state, so no other processor can be trying to read or write; however, programming with locks is hard to get right. We will not take this path in this class.

- Keep mutation effects in parallel code benign. It’s OK if multiple processors are each mutating state as long as they can’t see each others’ state. There’s no race condition there.

An example: in our lectures on parallelism, we’ve pretended that it’s always beneficial to do things in parallel when possible. In reality, this isn’t true. Like everything else in computing, parallelism comes with overhead and sometimes if we’re only doing a small amount of work, it’s not worth the overhead of parallelism. So practical implementations of divide-and-conquer parallel algorithms often “bottom out” at a certain problem size and solve the problem using the best-known sequential algorithm, which is often imperative. For example, in-place (imperative) insertion sort can be faster than Quicksort or mergesort for small lists. So if we were implementing parallel mergesort, we might divide the sequence until it reached a certain threshold, then allocate imperative arrays to sort the small lists using the faster algorithm.

3 Mixing Parallelism and Concurrency

Note: the purpose of this section is twofold: to show the problems that come with mixing deterministic parallelism with concurrency, and to show the power that this combination gives us. We show you the problems so you know that while parallelism and effects are both powerful tools, mixing them is playing with fire! We show you the power of these tools to convince you that the techniques used in this class are applicable in practical domains. The solutions to the problems we will see, and the examples that we give for using this power, are outside the scope of the class.

Most concurrency is done using languages or libraries that expose lower-level primitives like threads. We don’t need to go all the way down to that level if we give ourselves slightly lower-level primitives than the ones, like Sequences, we’ve been using so far.
3.1 Futures

Futures look a lot like suspensions, but rather than getting delayed until they’re forced, they execute in parallel.

```ocaml
type 'a future
val future : (unit -> 'a) -> 'a future
val force : 'a future -> 'a
```

The power of futures is that they allow us to express cost graphs that are not just series-parallel.

```ocaml
val f1 = future (fn () => fib 40)
val f2 = future (fn () => fib 41)
val fib40 = force f1
val fib41 = force f2
```

```
future
/ \
/ \
/ \
fib 40 future
| / \
| / \
| / ...
| / fib 41
| / \
| / \
| /...
| / \
| / \
| / force
```

3.2 Dynamic parallelism

With sequences, the “width” or amount of parallelism is fixed when you tabulate the sequence. Futures also let us spin off parallel computations dynamically, in response to other computations or even user input.

3.2.1 Fibonacci server

```ocaml
(* Compute and print Fibonacci of $n$ *)
fun fib n () =
  let fun fib' n () =
    if n <= 1 then 1
    else let val (r1, r2) = (fib' (n-1), fib' (n-2))
    in r1 + r2
    end
  in
  print (((Int.toString (fib' n ())) ^ "\n")
```

6
fun fib_server () =
  let fun loop futures =
    let val _ = print "Enter a number.\n"
    val request = inputLine ()
    in case request of
    NONE => List.app force futures
    | SOME "done" => (print "cleaning up\n"; List.app force futures)
    | SOME s =>
      (case Int.fromString s of
        NONE => loop futures
        | SOME n =>
          let val _ = print ("Computing fib(" ^ (Int.toString n) ^ "\n")
          val f = future (fib n)
          in loop (f::futures) end)
      end
    end
  in loop [] end
val _ = fib_server ()

### 3.2.2 Work and span

The cost graph for the Fibonacci server is shown below. What’s its work and span? We can easily calculate the work and span of `fib`, but what about `input`? Does that even make sense?
3.2.3 Scheduling

There are at least two ways the behavior of this program is dependent on how it’s scheduled:

- There’s a race condition between the two outputs (and, depending on how the terminal works, between the output of the Fibonacci and the last input).
- Clearly, the desired behavior of the program is for the output and input in `fib_server` to happen concurrently with the Fibonacci computation. But at the level of cost graphs, where we explicitly don’t “bake in” a schedule, there’s no way to express this! So SML is free to do the whole Fibonacci computation and then eventually get back to the interaction.

3.3 Communication using state

Sometimes parallel computation needs to pull in data as it’s running. This means we can’t push the I/O to the periphery, and these are also not benign effects. In this case, we have to think carefully to make sure that the effects are safe, and possibly use locks to ensure that the behavior is what we’d expect.

For example, one thread might be performing computations and reporting its results to another through a ref. Under many circumstances, this can be safe if one thread is just writing and another is just reading.

```sml
val log = ref 0
fun readloop last =  
  let val new = !log
    in
    (if new <> last then
      print ("New value : " ^ (Int.toString new) ^ ">
    else
    ();
    readloop new
  end

fun writeloop n = 
  let val fibn = fib n
    in
    log := fibn;
    writeloop (n + 1)
  end

val _ = future (fn _ => readloop 0)
val _ = writeloop 1
```

If we have more than one thread writing and/or the data is more complicated than a single integer, we might need locks to ensure that the state remains consistent (remember the bank account example above). Another example of the above pattern would be a web server that spawns threads to handle new connections, writes requests to a log, and has a thread that periodically checks the log and updates a statistics file.