15-150 Lectures 24 and 25: Scheduling; Introduction to Effects

Lectures by Dan Licata

April 12 and 17, 2011

1 Brent's Principle

In lectures 17 and 18, we discussed cost graphs, which can be used to reason abstractly about the time complexity of your program, independently of the schedule. For example, we showed how to associate a cost graph with each operation on sequences, so that you can reason about the work and span of your code via these graphs, without knowing the particular schedule.

But what do work and span predict about the actual running time of your code? The work predicts the running-time on one processor; the span on "infinitely many" (which you are unlikely to have). What can we say about the 2 processors in your laptop, or the 1000 in your cluster?

Theorem 1 (Brent's Theorem)**.** *An expression with work w and span s can be run on a p-processor machine in time $O(max(w/p, s)$.*

That is, you try to divide the total work w up into chunks of p, but you can never do better than the span s. For example, if you have 10 units of work to do and span 5, you can achieve the span on 2 processors. If you have 15 units of work, it will take at least 8 steps on 2 processors. But if you increase the number of processors to 3, you can achieve the span 5. If you have 5 units of work to do, the fact that you have 2 processors doesn't help: you still need 5 steps.

Brent's theorem should be true for any language you design; otherwise, you have gotten the cost semantics wrong! Thus, we will sometimes refer to it as *Brent's Principle*: a language should be designed so that Brent's Principle is in fact a theorem.

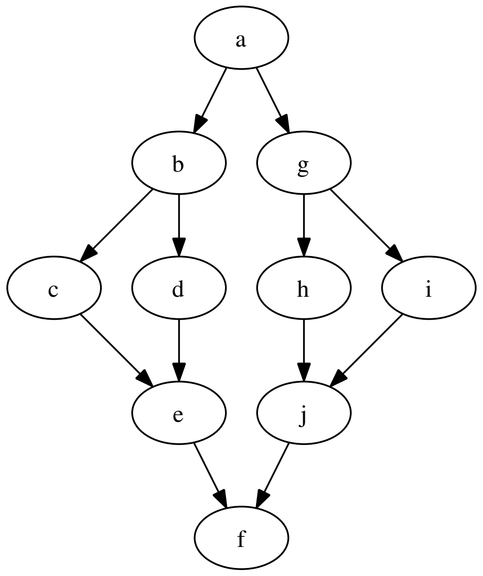
2 Scheduling

A *schedule* says, for each time-step, what unit of work each processor is doing. How dooes the compiler actually schedule work onto processors? It turns out cost graphs are a helpful data structure for scheduling. Let's take a little time to understand how the compiler might do this.

A schedule can be generated by *pebbling* a graph. To schedule a cost graph onto p processors, you play a pebble game with p pebbles. The rules of the pebble game are: you can pick up a pebble from a node or from your hand and place it on a node all of whose predecessors have been visited, marking that node as visited.

To generate a schedule that says what each pebble (processor) is doing at each time step, we will divide the pebbling up into steps, and say that at each step you can play at most p pebbles. The nodes played on are the units of work completed in that timestep. The restriction to playing on nodes whose predecessors have been visited ensures that dependencies are respected. At any time, the nodes whose predecessors have been visited, but have not themselves been visited, are the *frontier*.

Consider the following cost graph:



A 2-pebbling with pebbles X and O might start out like this:

Step	**X**	**O**
1	a	
2	b	g
3	c	

In the first step, we can only play a pebble on the source (all of it's zero predecessors have been pebbled), because no other node in available. This corresponds to a processor being idle because there is not enough available work to be done. In the second, we can play on both of the next two nodes. In the third step, we can play on any of the four nodes at the next level, but we can choose to play on only one of them. This corresponds to a processor being idle, even though there is work that could be done. A *greedy schedule* assigns as many processors as possible work at each time step. (A non-greedy scheduler might be more efficient overall if not all units of work took the same time.)

Two scheduling algorithms are $pDFS$ (p depth-first search) and $pBFS$ (p breadth-first search). DFS prefers the left-most bottom-most available nodes, whereas BFS prefers higher nodes (but then left-to-right). At each step, you play as many pebbles as you can.

Here is a schedule using 2DFS (let's say we prefer processor X if only one unit of work is available):

Step	X	O
1	a	
2	b	g
3	c	d
4	e	h
5	i	
6	j	
7	f	

In step 4, we prefer node e to node i because e is bottom-most.

Consequently, in the remaining steps there is only one node available to work on.

Here's a schedule using 2BFS:

Step	X	O
1	a	
2	b	g
3	c	d
4	h	i
5	e	j
6	f	

This time, we prefer i to e because it is higher.

Consequently, in step 6, two units of work are available to do, so we can finish in 6 steps instead of 7.

Thus: different scheduling algorithms can give different overall run-time. Additionally, they differ in how many times a processor stops working on the computation it is working on, and starts working on an unrelated computation. E.g. in BFS, in step 4, both processors jump over to the other side of the graph. This can be bad for cache (the small amount of memory very close to your processor) performance, but the details are subtle.

3 Parallelism and Effects

Effects are things your program might do instead of, or in addition to, returning a value. Examples we have seen so far: non-termination, I/O (printing, reading input, sending something over a network, ...), and exceptions.

Parallelism is about efficiency: running code on multiple processors at once. There are different ways to achieve parallelism. One is *determinstic parallelism*: you make sure that the behavior of a program is independent is independent of the schedule. This is what we consider in this course. An alternative is *concurrency*: thinking about multiple processes/threads at the language level. In concurrency, it is common to embrace non-terminism and effects. 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 three aforementioned effects here.

3.1 Non-termination

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

```
Seq.map f <1,2>
```

where `f 1` terminates but `f2` loops. Then no matter the schedule, the expression loops: whether it works on `f 1` first and then `f 2`, or vice verse, it eventually has to do all of the work in `f 2`, and so will it the loop. This is because, in the kind of parallelism we have discussed, programs always do all of the same work—in parallel execution, they just might do it faster.

3.2 I/O

What happens if you do

```
List.map (fn x => let val () = print (Int.toString x) in 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 => let val () = print (Int.toString x) in x + 1)
        (seqfromlist [10,20,30])
```

?

The value is `<11,21,31>`, but what do you see?

Possibilities include:

```
102030
201030
302010
321000
```

(Draw the cost graph, and look at 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. It is dependent on 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 happening in some other task in between!

This is bad, because it means you need to think about parallelism when reasoning about the behavior of your program, not just the efficiency. This is much harder.

To avoid this, we advocate **keeping 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 this implementation of game, the effects are 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 program functionally, so that we may exploit parallelism.

3.3 Exceptions

Similar problems come up with exceptions. For example, in

```
Seq.map (fn x => raise Fail (Int.toString x)) (seqFromList [1,2,3])
```

what exception do you see?

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 now 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 `Seq.map` did the wrong (ill-defined) thing, we could implement our own `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 `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 `exn`):

```
datatype 'a result = Success of 'a | Failure of exn
```

Next, we recall the 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:

```
fun reify (f : 'a -> 'b) : 'a -> 'b result =
    fn x => Success (f x) handle e => Failure e

fun reflect (f : 'a -> 'b result) : ('a -> 'b) =
    fn x => case f x of Success v => v | Failure e => raise e
```

Next, we implement the join-point logic:

```
fun leftToRight (s : 'b result Seq.seq) : 'b Seq.seq result =
    Seq.mapreduce (fn Success v => Success (Seq.singleton v)
                    | Failure e => Failure e)
                  (Success (Seq.empty()))
                  (fn (Success v1 , Success v2) => Success (Seq.append (v1,v2))
                    | (Failure e , _) => Failure e
                    | (Success _ , Failure e) => Failure e)
                  s
```

The final argument 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 `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:

```
fun emap (f : 'a -> 'b) : 'a Seq.seq -> 'b Seq.seq =
    reflect (leftToRight o (Seq.map (reify f)))
```

What is the span? Unfortunately, this adds a logarithmic factor to `map`, because the exception propagates in a tree-like manner. A primitive implementation of `map` can track which exception to raise in constant time. Thus, you are free to use exceptions in parallel code, and you will see a deterministic, left-to-right semantics for which exception gets raised.

4 Mutable Cells

Functional programming is all about transforming data into new data. Imperative programming is all about updating and changing data.

To get started with imperative programming, ML provides a type `'a ref` representing *mutable memory cells*. This type is equipped with:

- A constructor `ref : 'a -> 'a ref`. Evaluating `ref v` creates and returns a new memory cell containing the value `v`. Pattern-matching a cell with the pattern `ref p` gets the contents.
- An operation `:= : 'a ref * 'a -> unit` that updates the contents of a cell.

For example:

```
val account = ref 100
val (ref cur) = account
val () = account := 400
val (ref cur) = account
```

1. In the first line, evaluating `ref 100` creates a new memory cell containing the value 100, which we will write as a box:

   $\boxed{100}$

   and binds the variable `account` to this cell.

2. The next line pattern-matches `account` as `ref cur`, which binds `cur` to the value currently in the box, in this case `100`.

3. The next line updates the box to contain the value `400`.

4. The final line reads the current value in the box, in this case `400`.

It's important to note that, with mutation, the same program can have different results if it is evaluated multiple times. For example, the two copies of the line `val (ref cur) = account` bind `cur` to two different numbers, depending on the value currently in the box. This is characteristic of effects such as mutation and input/output, but not something that a pure functional program will do.[1]

It's also important to note that the value of the *variable* `account` never changed. It is still a mathematical variable, just like always: it always stands for the same box. However, that box can have different values at different times, which is a new ingredient.

Here are some useful functions on references:

```
fun !(ref v) = v
fun update (f : 'a -> 'a) (r : 'a ref) : unit =
  let val (ref cur) = r in r := f cur end
```

The first gets the value of a cell, so we can write `!r` instead of pattern-matching. The second combines a read and a write; e.g. `update (fn x => x + 1) r` is like `r++` (except it doesn't return either the before the after value).

For example:

[1] Insanity is doing the same thing over and over again and expecting different results.

```
fun deposit  n a = update (fn x => x + n) a
fun withdraw n a = update (fn x => x - n) a
```

4.1 Mutation and Parallelism: Race Conditions

What happens if I do

```
val account = ref 100
val () = deposit 100 account
val () = withdraw 50 account
```

In the first line, `account` is bound to a box, which has the following values

after line 1	$\boxed{100}$
after line 2	$\boxed{200}$
after line 3	$\boxed{150}$

What happens if I do

```
val account = ref 100
val _ = Seq.tabulate (fn 0 => deposit 100 account | 1 => withdraw 50 account) 2
```

The `tabulate` runs the deposit and the withdrawal in parallel. So the value in the box at the end is 150, because addition is commutative, so it doesn't matter which order I do the transactions in, right?

Wrong. The reason is that the two transactions might not happen in either sequence: they might be interleaved. In particular, the value in the box at the end might be 200, or 50.

To see this we can make a cost graph and look at the various possible schedules for two processors:

```
                          .
                         / \
   ref cur = account .     . ref cur = account
                     |     |
account := cur + 100 .     . account := cur - 50
                         \ /
                          .
```

If the schedule does first the whole left side, then the right, then we get 150 in the box at the end. Similarly, the schedule does first the whole right side, then the left, then we also get 150. However, what happens with a breadth-first schedule? We read the *initial* value into `cur` on both sides (on the two different processors), and then write the updated values. So one write will clobber the other, and we will get either 200 or 50.

This is called a *race condition*: two processors are accessing the same memory cell at the same time, with at least one doing a write. Consequently, the meaning is dependent on the schedule.

What can we do about race conditions? One possibility is to use *locks* to force the read and write on each side to happen *atomically*, without any other processors accessing the memory. Before doing the left-side, you acquire a lock (no one else touch this for now!); after, you release the lock (okay, I'm done; go!). Programming with locks is pretty tricky, though.

Another possibility is to *avoid mutation in parallel code when possible.* That is, inasmuch as possible, you want to rely on functional programming in parallel code, because that way you inherently avoid race conditions. It is also possible to make careful use of mutation in parallel code, in such a way that you still get a schedule-independent result—see the discussion about benign effects below.

4.2 Mutation and Verification

Exercise: prove that `1 = 2`.

```
fun ++ r = (r := !r + 1; !r)
val r = ref 0

++r ()
== 1

(++ r ; ++ r)
== (r := r + 1; !r) ; ++ r
== let val _ = (r := r + 1; !r) in ++ r
== let val _ = !r in ++ r
== ++ r
== 2

By reflexivity,
++ r == ++ r

Therefore, by transitivity,
1 == 2
```

Where did we go wrong?

One way to diagnose the problem is that *reflexivity doesn't hold*: `++ r == ++ r` implies that an expression means the same thing if you run it twice; in the presence of mutation, that's not true!

Another way to diagnose the problem is that an expression by itself doesn't tell you everything you need to know to evaluate a program. The second `++ r` is really in a different memory than the first one, so it's wrong to assert that they are equal.

There is a large body of work on reasoning about imperative programs with mutation. We won't have time to get to this in 150. But this example shows that reasoning about imperative programs is a lot harder than reasoning about purely functional programs, which is why we've been focusing on pure programming all semester.