15-150 Lecture 1: Overview

Lecture by Dan Licata

January 17, 2012

Welcome to 15-150!

1 Parallel Functional Programming

The first thing I want to do is take a poll, to get a sense of what background you folks have: in particular, to figure out who has taken 15-122 last semester.

Let's do it this way: you there, in the back left corner, what's your name? Okay $StudentA, tell the person to your left whether you took 122 or not. Great. Now you, what's your name? Okay, $StudentB, you now know the count of how many people took 122 among all the people in the row to your left, which between $A and $B is either 0,1, or 2. Introduce yourself to your neighbor and pass the count down the row; everyone should add one to the count if they took the course, and just pass it on otherwise. [pause] Alright, now the next row, you do the same thing. [long pause] Man, this is going to take forever. I don't want to spend the whole lecture just collecting this one bit of background data. Let's try this: each remaining row, do what the top two rows just did, but do it at the same time. [pause] Okay, now what we have at the ends here is another row. So let's do the same thing down the side. Great! There's our number!

So. Anyone wondering why I didn't just ask you to raise your hands? What just happened is that you folks acted out a *parallel functional program*. This is our first course objective:

You will learn to write parallel functional programs.

By *parallelism*, I simply mean "the ability to do many things at once". Why is parallelism important? First, most computers these days are *multi-core*: the have many processors, which means they can do many things at once. Second, lots of processing gets done by *clusters*: a bunch of computers working together. This is how Google processes the Internet. So the challenge is to take advantage of these computational resources: how do you write a program in such a way that there are many things that can be done at once? In this course, we will show you an approach based on functional programming.

Unlike imperative programming, which n of you studied last semester, functional programming is all about studying transformations on data. In this case, we transformed you, the students in the class, into the number of you that took 122. That is, we ran a function

```
(* Purpose: count the number of students who took 122 *)
count : students -> int
```

on you. The word `count` is the name of a function that we're going to define. The phrase `students -> int` is a *type*, which is a contract on the behavior of this function. The type `students -> int` should be read "`count` is a function that transforms a collection of students into integer." The line above it is a comment, which explains the purpose of the function.

This semester, we're going to be programming in a language called Standard ML, or SML, and in SML we can implement `count` as follows.

First, we need to explain how `students` are represented. Well, you guys are organized into rows, so let's assume that the collection of all students is a *sequence* of `rows`:

```
type row = int sequence
type students = row sequence
```

and that a row is a sequence of integers, which are either 0 (if you didn't take 122) or 1 (if you did). (In general, I don't like reducing people to numbers, but it's convenient for this example.) The notion of a sequence is something we will spend a lot of time on this semester; for now, you can think of it as an abstract, ordered collection of things, just like you students in the back row there. We'll figure out what we need out of sequences as we go along.

First, let's assume we have a function

```
(* Purpose: add up the numbers in the row
Example: sum applied to the row
         4 8 15 16 23 42
         returns 108 *)
sum : row -> int
```

How do we use this to count the whole class? We *define a function*

```
fun count (s : students) : int = ...
```

This declaration introduces a function with the above type, `students -> int`. How did we implement this function when we acted it out?

1. First count each row (in parallel)
2. Then count the column at the end, which after all is just another sequence of integers.

Here's how this looks in ML:

```
fun count (s : students) : int = sum (map sum s)
```

The phrase `map sum students` corresponds to (1) and the additional phrase `sum ...` corresponds to (2). Let's look at this code in more detail: For (1), `map` is an operation that applies a transformation to a sequence, producing another sequence. In this case the transformation is `sum`, which is the process you folks used to count up a row, and the sequence is the whole classroom. The result is the sequence of integers we computed at the ends of the rows. For (2), we apply the function `sum` to the result of `map`, which is the count of each row.

This example illustrates how you can express parallelism in a functional language: First, you think in terms of *compound data structures*, like sequences, and *bulk operations* that do something to each element of the data structure. Second, it is important that the thing you do is a *mathematical*

transformation, because otherwise doing something to one element could influence the result on another. In the case, the notion of transforming a sequence of integers into its sum is a nice, well-defined mathematical notion. (For those of you who took 122, this emphasis mathematical transformations is a contrast with imperative programming.) Once you have specified a program in such a form, the mapping to concrete hardware (e.g. the two processors in your macbook) is taken care of by the compiler. You think about the things you want done; the compiler thinks about the order in which to do them; and the result is still well-defined.

Computing by calculation An important idea that we will talk about this semester is *computing by calculation*.

Suppose the class looks like

```
row 1: yes no  yes
row 2: no  yes yes
```

We can represent this as follows:

```
val row1 : row = seq [1 , 0 , 1]
val row2 : row = seq [0 , 1 , 1]
val classroom : students = seq [row1 , row2]
```

To count the whole classroom, we apply the function `count` to the —classroom—, as in

```
count classroom
```

This is an *expression*, or a program phrase.

We run a program by *computing by calculation*, just like in high-school algebra. To evaluate the expression `count classroom`, we can calculate as follows:

```
   count classroom
== count (seq [row1 , row2])
== sum (map sum (seq [row1 , row2]))
== sum (seq [sum row1 , sum row2])
== sum (seq [2 , 2])
== 4
```

`4` is a numeral, which is a *value*—the final result of a computation. Calculating out the value of the expression `map sum (seq [row1 , row2])` can be done in parallel, just like we acted out.

This example has illustrated the first goal of the course: you will learn to *write (parallel) functional programs*. The parallelism arises naturally from computing by calculation, because we can calculate in many parts of an expression at once (e.g. adding up each row).

2 Complexity Analysis

The next goal of the course is:

> You will learn to analyze the sequential and parallel running time of your programs.

For example:

- How long does it take to count the class, if only one person can add at a time? This corresponds to execution on a single-processor machine, and is called the *work* of the algorithm.

  Answer: Well, there is one addition per student, plus the addition "down the side" of the classroom, but that row certainly has fewer students than the class as a whole. So the number of additions is roughly proportional to *the number of students in the class*.

- How long does it take to count the class, if infinitely many people can add at a time?

  Answer: We still have to wait for (a) the longest row to do its addition and (b) the "row" at the side to add up at the end. Thus the answer is roughly proportional to the *the length of the longest row* plus *the number of rows*.

  This is called the *critical path*, because it is the limiting factor, even with as many processors as you need. It is determined by the *data dependencies* of the computation—e.g. you can't add yourself to the running total until you know the running total. The length of the critical path is called the *span* of the algorithm.

- How long does it take to count if we have p processors? We'll get to this soon. Intuitively: try to divide the work W evenly over p processors, but you can never do better than the span S.

Asymptotic complexity analysis allows you to predict how long it will take to run your code on really big inputs, without actually running it on them. It is one of the main tools used to choose between different algorithms for the same problem (sometimes the constant factors matter more, but usually it's the behavior when the inputs get very large that matters).

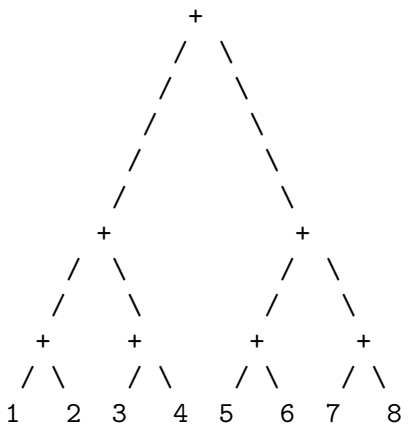
3 Sum with a better span

Now that we know about work and span, let's analyze the process of adding across a row. The work is the number of students in the row, because you need to add that many numbers. The span is the same: you wait for the sum of the people to the left of you before you do anything.

What if a row was really big? Can anyone think of a way to do better?

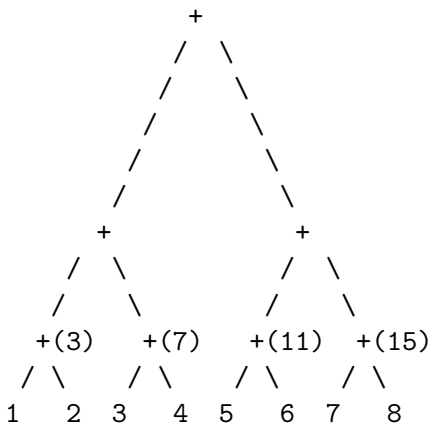
Let's say we had four people: each pair, add your numbers together. OK, now add together the two pairs. OK, now let's try with eight people.

To make it easier to follow the example, we'll add up the sequence `seq [1,2,3,4,5,6,7,8]` (this might be the column that results from counting across 8 rows). To visualize this process, we can draw the computation as a tree of additions:

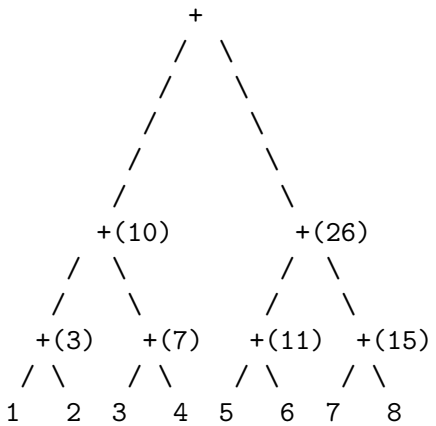


What is the work? There are still 7 additions that you need to do. What about the span? If we have enough processors, we can do it like this:

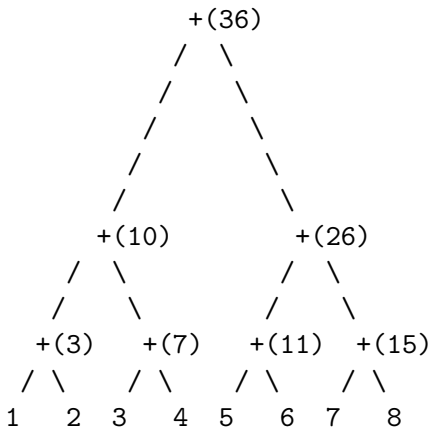
Step 1:



Step 2:



Step 3:



3 steps for 8 items: the span is *logarithmic* in the number of students in the row. This is much better than linear when things get large.

This way of summing up the tree in another common pattern of computation, and is represented by a function `reduce`, which is a general way of putting a binary operator (in this case `+`) at each node in a tree.

We can implement `sum` using `reduce`:

```
fun sum (s : int sequence) = reduce (op+) 0 s
```

4 Reasoning

The third goal of the course is

> You will learn to reason mathematically about the correctness of functional programs.

For example, if we ask the question "is adding up the numbers one by one (we'll call this `sum`) the same as adding them up using a tree (which we will call `sumtree`)?", we can answer it with a *theorem*:

Theorem 1. *For all rows* `r`*,* `sum r` *and* `sumtree r` *compute to the same integer.*

First, let's get some intuition: why should this be true? Returning to the example of `seq[1,2,3,4,5,6,7,8]`, we see that *sum* computes

```
1 + (2 + (3 + (4 + (5 + (6 + (7 + (8 + 0)))))))
```

whereas *sumtree* computes

```
((1 + 2) + (3 + 4)) + ((5 + 6) + (7 + 8))
```

Thus, the basic reason is that *addition is associative* ($(x + y) + z = x + (y + z)$*, with unit 0* ($x + 0 = x$).

Proof. More formally, we can do a proof by *structural induction on the sequence* r. Don't worry if you don't understand this term yet; you will in a couple of weeks.

□

Why do you need to be able to reason about your code like this? First, it's important for *safety-critical systems*, where you really want to know that your program doesn't crash—e.g., it's running on an airplane. Second, we're going to argue that this kind of reasoning is a useful part of program development. Thinking through invariants this way will make it easier to write code. For example, like we saw with `sum` and `sumtree`, it's often the case that there is a simple, obviously correct, but inefficient algorithm for a problem; and also a more complex algorithm that's more efficient. Proving them equivalent *mathematically justifies your program optimization.*

5 Course Overview

To summarize, you will learn to:

- Write parallel functional programs
- Analyze their sequential and parallel time complexity
- Reason mathematically about their correctness
- Structure them using abstract types

The thesis of the course is that functional programming is a good way to do all of this: it makes it easy to express programs (cleanly, elegantly), analyze their time complexity, and verify their correctness. And moreover that this all applies to parallel programs: you can easily specify work that can be done in parallel, and analyze a program's parallel complexity, and prove correctness about parallel executions (indeed, in this course, it will always have the same result as a sequential execution!).

Because of that, we think functional programming is a good tool for you to have in your toolbox, as a beginning computer scientist in 2011. There's going to be a lot of big data out there during your career: Maybe you'll work for Google, and have to process the whole Internet. Maybe you'll design a chip with billions of transistors. Maybe you'll be a scientist working on protein folding or huge amounts of astronomical data. Maybe you'll be an urban planner and collect data from millions of people about how long it takes them to drive to work. We think that the ideas we teach you in this course will help you with these tasks. As you go through the course, we want you to think about this claim, and evaluate it for yourself.

I can't say much about the fourth course goal at this point; it will make sense later.

See the course web page for more on the course logistics.