[1] Here we have a concurrent composition. On one side, we have an output on channel c. On the other side we have an input on channel c, and then the value read is assigned to variable x. Now you might think that the value output on one side, namely 2, is the value input on the other side. Well, I forgive you for thinking that. It's a reasonable mistake. Let's see what really happens. [2] The left side says the message at the write cursor is 2, and the write cursor is increased, and I'm ignoring time. The right side says the read cursor is increased, and then x is assigned the message just read. So w is a variable on the left side, and r and x are variables on the right side, and let's say there aren't any other variables. Then [3] the message at w is 2, w and r are increased, and the final value of x is M of r. There's no way to conclude that x is 2 because we don't know that w and r are equal. And they might not be. Maybe this concurrent composition is [4] part of something larger, and there was an earlier output on c, and a later input on c. The first input might get the 1 that was output first, and the second input might get the 2 that was output second. I say might because this might be part of something even larger. What we need for communication between processes is a fresh channel that hasn't been used before. And we get it by [5] channel declaration. It looks like [6] this. C is the name of this fresh, local channel, and T is the type of value that will be communicated on this channel. And P is the program, or specification that c is local to. It's defined like [7] this. It's a message script, a time script, a read cursor, and a write cursor, and the two cursors are initialized to 0. Using a local channel [8] here's what we get. The channel declaration is some [9] existential quantifications, and the concurrent composition we already worked out. Now here's the important bit. r and w are initialized to 0, so that's a [10] substitution for r and w, making x prime equal to M 0, and M 0 equal to 2. And that's why x prime is equal to 2. After we [11] get rid of the local variables, all we're left with is x prime equal to 2, and other variables unchanged if there are any, and that's just [12] the assignment x gets 2. That's the net effect of the top line. x gets 2. We were ignoring time, but [13] if you include time, then there's a wait for input because, according to transit time, it takes time 1 for the output to become available as input, so it's an assignment that takes time 1.

[14] For deadlock we definitely have to include time, because deadlock means processes waiting on each other. Well, [15] this example is just one process waiting on itself. On a fresh channel, first it tries to do input, with the usual expression for input wait, and then it does output. [16] Replacing chan by its definition, and input by its definition, and output by its definition, we get this. Now from chan we have r and w are initialized to 0. Let me just make that into explicit assignments [17], and also I've written out the final assignment as an increase to w and leaving r and t alone. Ok, now I can simplify using the substitution law 4 times, and [18] here's what we get. The place to look is [19] here. We have script T 0 equal to the maximum of time t and script T 0 plus 1. So script T 0 is bigger than or equal to itself plus 1. How can anything be bigger than or equal to itself plus 1? Only if it's infinity, because infinity is equal to itself plus 1. So [20] t prime has to be infinity also. When we get rid of all the local variables, [21] the only thing left is t prime equals infinity. And that makes sense because the execution will wait forever for that first input.

[22] Here's the more usual example of deadlock, with two processes waiting on each other. Each of them tries to read first, and then write after. To see the deadlock, I have to [23] put in the wait for inputs. Now we [24] expand by using the definition of chan and input and output, partitioning the variables, and using the substitution law to simplify. I've done all that and here's what I got. The important line is [25] here. It says that T c 0 is bigger than or equal to T d 0 plus 1, and also that T d 0 is bigger than or equal to T c 0 plus 1. The only way two things can each be bigger than or equal to the other plus 1 is if they're both infinity. So that makes [26] t prime equal to infinity. And when I get rid of all the local variables, [27] that's all that's left. If you want to prove that a program is free of deadlock, you just have to prove that execution doesn't take forever waiting for input.

[28] To end the lecture, I have a fine example of how communication, concurrency, and recursive refinement can all work together. It's power series multiplication. The [29] input on channel *a* is a sequence of numbers, which we'll call *a* 0, *a* 1, *a* 2, and so on, which are the [30] coefficients of a power series that we'll call capital *A*. And the [31] input on channel b is a sequence of numbers, which we'll call b 0, b 1, b 2, and so on, which are the [32] coefficients of a power series that we'll call capital B. And the [33] output on channel c is also a sequence of numbers, which we'll call c 0, c 1, c 2, and so on, which are the [34] coefficients of a power series that we'll call capital C, which is the [35] product of the two input series. So that's the specification, except for the timing, which is that the outputs have to be produced one per time unit. That's a difficult time constraint, as we'll see in a moment. If we multiply out big A times big B, we see [36] what the coefficients of big C have to be. As we go along in this computation we have to save all the coefficients that we read, because they are all still needed for each new output, so we need dynamic storage allocation. And, as we go farther and farther out in the series, there's more and more computation needed for each new output coefficient. For c n we need n plus 1 multiplications and n additions. So we need more and more computation in each time unit. We're going to need dynamic process generation, too. Does that sound like a hard problem? How many lines of code do you think we'll need? I've seen solutions that ranged from 5 to 10 pages of code, with no proof of correctness, of course. For the solution I'm going to show you, it helps to make [37] these abbreviations. Big A 1 is the power series on channel *a* that starts with coefficient 1. Big A 2 is the power series on channel *a* that starts with coefficient 2. And similarly for channel b. The reason they help is that I can get rid of [38] these informal dots and express the entire output by writing power series in [39] these four positions. Like [40] this. [41] These two products just multiply the coefficients of a series by one number, so that's an easy loop. The [42] product A 1 times B 1 is just a recursive call. It's exactly the same problem we started with, one coefficient farther along. But there's a problem; we don't want the recursive call to produce its outputs on channel c because they aren't the answer; they're just an ingredient in the answer. So we have to have a channel parameter.

[43] Here's the specification. But I give it a [44] channel parameter c. And then we [45] apply this function to channel argument c, so it's just the same specification. And [46] here's the program. The whole program. It's just 5 lines long. It starts by [47] reading the first coefficient from channels *a* and b, and then [48] outputting c 0. Good start. Now before we read any more input, we have to remember the input we have, because we will need it again. So [49] we declare local variable *a* 0 and assign it the input that we read on channel *a*. Likewise we declare local variable b 0 and assign it the input that we read on channel b. And we need a local channel declaration for the output from the recursive call. And [50] here is the recursive call, with argument d. That will produce the coefficients of [51] this product. [52] In parallel with that, we [53] read the next input on *a* and b, and we can [54] output c 1. That's the first two coefficients, c 0 and c 1, done, and we still have all the rest to go. That's what [55] this specification says, where big D is the power series from channel d. And we have to refine this specification. That [56] starts by reading the next coefficient from *a* and b, which is *a* 2 and b 2. And [57] the first coefficient from d. And then outputting the sum of [58] *a* 0 times b 2, plus [59] the first coefficient from the recursive call, plus [60] *a* 2 times b 0. And [61] then repeat. End of program. [62]

I don't expect that explanation to be totally convincing. What's convincing is the proof of these two refinements. The details of the proofs are not important. They're just like many of the proofs we've done before, and the proofs are in the textbook. What's important is that you see how the theory of programming can help you write elegant, efficient programs that you can prove correct. The theory helps you be a much better programmer.

What happened to the dynamic storage allocation we were going to need? Can you see it? It's [63] here. This is a recursive refinement, and each recursion contains local variable declarations. And where's the dynamic process generation? It's [64] here. Every recursive call introduces more concurrency. We don't need any new programming language constructs to create processes. We just need to use the structures we have intelligently.