[talking head] One of the problems with shared variables, and also with interactive variables, is that they provide too much information. A process doesn't need the complete state of another process at all times. It just needs something about the state, and only at some times. Another problem is that a process has to somehow time its reads so that it isn't reading the value of a shared variable or an interactive variable while its owner process is in the middle of writing to it. There's another way to get interaction between processes that doesn't have these problems. It's called [1] communication channels. And it's good for communication between computers and people, also. [2] A communication channel is 4 things. [3] The message script is the string of all communications on the channel. That's all communications, past, present, and future. It's not a variable; it's a constant. [4] The time script is the string of times that the messages are sent. It is also a constant. [5] The read cursor is a variable indexing the two scripts, telling how much has been read so far. [6] Likewise the write cursor is a variable telling how much has been written. [7] Here's a picture of it. By the way, if there's more than one channel, we use the channel name as part of the message script name, and part of the time script name, and cursor names, to say which channel we're talking about. But if there's only one channel, or it's obvious which channel we're talking about, we might leave out the channel name, as I've done in this picture. The messages here are all natural numbers, but they could be any type. The times have to be nondecreasing numbers, as you see here. The write cursor, w, shows how much has been written, and the read cursor, r, shows how much has been read. As the computation progresses, w and r might increase, but they never decrease.

To an observer, watching the computation, future message are unknown. The observer does not see these scripts, or the cursors. The observer just sees the messages as they are read. But a specifier has to be able to say what the messages will be when the program is executed. So the specifier needs to refer to all messages and times, past, present and future, and that's what the scripts are for. They are not programming notations. [8] Here are some programming notations. [9] This one is output, or writing. On channel c, write the value of expression e. It says that the message, at the write cursor, is e, and the time at the write cursor is t, and increase the write cursor. [10] Next comes reading, or input. In most programming languages, input has a variable to serve as the target of the read. But not here. The value being read already exists in some buffer somewhere, and there's really no need to make another copy of it. So reading is just increasing the read cursor. [11] The way we refer to the value that was just read is by the channel name. That saves having to misspell something because the name that is appropriate for the channel is also the name that would be appropriate for a variable to receive a message from the channel. And last [12] is a binary expression, check c, that tells whether there are any written but unread messages on this channel. Script T is the list of times that messages are written, so it says: the time when the message-to-be-read-next was written is before now. [13][14] Here's a sample program just to see how they look. [15] If there is any input on channel key currently available, then [16] read it. And [17] if the key we just read is the letter y, [18] then output this text to channel screen, [19] else output some other text, [20] and if there's no input available on channel key, then print a prompt on channel screen. [21]

As a simple example of specification, [22] we can repeatedly input numbers from channel c, and output their doubles on channel d. Formally, that's [23] for all n, the message on channel d at w plus n is twice the message on channel c at r plus n. We cannot assume that the messages we are reading and writing are the first ones ever read or written on these channels. We have to start at the current read and write cursor positions, and specify that after that, whatever we find on channel c will be doubled and written on channel d. Let's call the specification [24] S for short. And [25] now we refine it. The program says read on channel c, and then output on channel d twice what was read, and then repeat. We have to [26] prove the refinement. So I'll [27] start with the right side. Reading on c [28] just means increasing the read cursor. Writing on d means [29] the message on d at the write cursor is the expression being written, which is twice the message on c that was just read. And there's a time part, but I left that out because we're not worrying about time in this one. And increase the write cursor. And then [30] S again. [31] This assignment will be used in the substitution law to substitute for [32] r c right here. And [33] these two assignments will be used for substitution in [34] S. So [35] here's what we get. The first output is twice the first input, and then all the outputs after that are twice the inputs after that. We can simplify that [36], and what we get is just [37] S again, so that's the proof.

[38] We need to account for the time of communications. As always, [39] there's real time, where we need to know the implementation to know how much time to charge for the various operations. [40] Without the implementation, there's transit time, in which input and output take no time, but communication transit takes time 1. That means, after the message has been written, there's 1 time unit before it's ready to be read. It's in transit between the writer and the reader. Also, the reader might have to wait a long time before the writer writes the message that the reader is supposed to read. So whenever there's an [41] input, we put t gets the maximum of t and time script at read cursor plus 1 in front of it. We have to wait at least until it's written, that's script T sub r, plus 1 more time unit for transit. But we might not have to wait at all if the message was written more than 1 time unit ago. So that's the maximum of the current time, and the time the message is ready to be read. [42] Checking for input becomes script T sub r, plus 1, is less than or equal to time t.

Let's try an example with time. [43] W says wait for input, and then read it. This isn't really much of an example, because it's just a single input, which is a programming notation, with the normal input wait in front of it. So it's program already, and we don't need to implement it. But I'm going to implement it like [44] this, because this is how input really is implemented on every computer. If the input is there then read it; if not then check again, repeatedly. It's a busy-wait loop, with recursive time. To [45] prove the refinement, I start with the right side. We [46] replace check c and W with their definitions. In the then-part, I'm putting [47] t gets t, which is harmless, and in the else-part I'm using the substitution law [48] to replace t in the middle part with t plus 1. Now the main step. In the then-part, we can assume the if-part is true, so the time t is the [49] maximum of time t and script T sub r, plus 1. In the else-part, we can assume the if-part is false, so script T sub r, plus 1 is strictly bigger than time t, and I'm also assuming time is integer-valued, so the maximum is script T sub r, plus 1, and it's also the maximum of [50] time t and script T sub r, plus 1. Now the then-part and else-part are the same, and they're both [51] W. And that completes the proof.

[52] Here's an example of a recursively defined program. It's the example we saw a few minutes ago, where the outputs were double the inputs, but there we had a specification, and we refined it to a program and proved the refinement. Here we are defining dbl by this equation. What makes this different from the recursive definitions we saw before is that this is an infinite loop. It has more than one solution. The [53] weakest solution is the specification we saw before, but with timing added. It says that the outputs on channel d are double the inputs on channel c, and they occur one per time unit. A [54] stronger solution, in fact the strongest implementable solution, says also that the read cursor for c and the write cursor for d and the time all end at infinity, and the write cursor for c and the read cursor for d are unchanged. The [55] strongest solution is false, which is unimplementable. If we want to define dbl as the weakest solution, we need to add an induction axiom. Even without an induction axiom, we can [56] use dbl to refine the weakest solution, and [57] we can execute it.

[58] If you are given a recursive definition, which is what any procedure in your favorite programming language is, or even any loop in your favorite programming language, and you want to know what is being defined, you can try recursive construction, as usual. [59] Suppose we start with true. Then [60] dbl 1 is obtained by using dbl 0 in the definition,

and after simplification, it says that the first output is twice the first input, and it happens at time t, right away. [61] We get dbl 2 by using dbl 1, and after simplification it tells us that the first two outputs are double the first two inputs, and they happen at t and t plus 1. [62] When we get to dbl infinity, it says what all the outputs are, and when they are, and it is the weakest solution of the dbl equation. And of course we could try other starting points and get other solutions.

[63] The last topic for this lecture is the monitor. It's important because this is the way shared variables are implemented. Monitors have been invented by more than one person, and their origin is in some dispute. But they are used whenever shared memory for parallel processes is wanted. [64] A monitor for a shared variable x is a process in parallel with all processes that share variable x. If some process, let's say process 0, writes to x, that means assigns to x, there must be a channel, let's say channel x0in, to send the value to the monitor for x. And the monitor has a channel x0ack to send back an acknowledgement that it has assigned the value to x. If another process writes to x, there must be a channel, x1in, to send the values it assigns, and a channel x1ack for the monitor to send back acknowledgements. And so on. The left side of the picture shows the channels for the processes that assign values to x. On the right side we have the channels for the processes that need the value of variable x. They may be the same processes that write to x, or different processes. If process 0 needs the value of variable x, it sends a request, on channel x0req, and then the monitor sends back the current value on channel x0out. And so on for any other processes. What does the monitor do? Roughly speaking, it serves the requests from the other processes to assign to or read from the value of variable x, in the order it receives the requests. Why do we need a process to do that? Because there may be conflicting requests, for example, two processes may request to assign different values to x at the same time. The monitor has to resolve all such conflicts. [65] Here's its specification. There's [66] one line like this for each input channel to the monitor. Each of these lines serves one request to assign to x or to read x. And they're all disjoined together, so at least one of them has to be satisfied, and then the monitor repeats endlessly. [67] The first line is for process 0 to assign to x. The action performed is [68] to read the value to be assigned, then [69] to assign it to x, then [70] to send back the acknowledgement. On [71] this line, the action is to [72] read the request for a value from process 1, then [73] send back the current value of x to process 1. [74] That's what happens. Now the harder part is when it happens. [75] Here it says if there is input already on a channel from another process, then the corresponding action can happen. If not, if there are no inputs already there, then [76] these equations determine which action happens. The script Ts are the times the next input will be sent, and [77] m is the minimum of all these times. So if the first request that will arrive is on this channel, then this action can be performed. [78] There is some nondeterminism. If there's a tie for which request comes first, either one can be served first and the other next. If requests arrive faster than they can be served, again the order is nondeterministic. When we implement a monitor, we make a choice about how to resolve the nondeterminism. For example, [79] here's one way to implement it. The channels are checked cyclically, serving requests whenever input is available.

I've shown you how to specify monitors for shared variables, and how to implement them. What I haven't shown you is how to use shared variables and reason about them in programs. That's because I find them too hard to reason about. That's why I stick to interactive variables, which are easier but still hard enough. Or better yet, to boundary variables, using channels for interaction.