[1] We have been talking about theory design. There are two styles of theory design, and so far we have just talked about data theory design. That style is to have operations like [2] push, which take a data structure as argument, and return a data structure as result. One thing you might do with the result is [3] assign it to a variable. In practice, data structures can be large, and you don't really want to pass them around. Most people would program in the program theory style, and just say [4] push x to mean push x onto the stack, and we don't need to say which stack if there's only one.

In the program theory style, the variables are divided into two kinds, [5] the user's variables and the implementer's variables. The user and the implementer cannot see or change each other's variables, except through the operations provided by the theory. The actual stack structure is an implementer's variable, so the user cannot make arbitrary changes to it. The user says push and pop to make the changes allowed by the theory.

[6] A simple version of stack theory, as a program theory, might introduce the names push, pop, and top. Push is a procedure, that means when you apply it to an argument it becomes a program. Pop already is a program. And top is an expression. Here [7] are all the axioms we need. The first one says that if we push x onto the stack, then afterward the value of top is x. And the other one says that if you push and then pop, you've done nothing. That's enough to prove that any number of pushes followed by the same number of pops puts everything back the way it was. We [8] start with the second axiom, and then we can [9] put ok anywhere we want because it's the identity for sequential composition. And then we [10] use the second axiom again, and you see that 2 pushes are undone by 2 pops. [11]

Let's start with the first axiom, [12] and then stick ok [13] on the end, and then use the [14] result from a moment ago, and we see that if we push something onto the stack, and then do further pushes and pops on top of that, then later, we find the same value x on top. And that's all we want from a stack.

For [15] stack implementation, we need an implementer's variable, so s of type list of any length of X values. Push [16] is a procedure, which is a function whose body is a program. It's an assignment to s that joins x to the end. Pop [17] is a program that cuts one item off the end of s. And [18] top just expresses the last item of list s. I claim that's an implementation, but we have to prove it. And that means, using the definitions of the implementation and the theories used in the implementation, prove that the axioms are satisfied. Here's [19] the first axiom. We use the implementation, then we get rid of the assignment notation, and then the antecedent says that s prime is list s followed by one more item x. So the last item of s prime is x, and that's the consequent. The other axiom is proved similarly.

When we add new axioms, we have to worry about [20] consistency. And the only proof of consistency is implementation. So after we prove both axioms, we know that if list theory and function theory are consistent, which they are, then so is stack theory. The [21] other question is completeness, and again stack theory is incomplete, because if we start with pop we can't prove or disprove anything.

[22] A slightly fancier version of stack theory has two more names. [23] Is empty is a binary expression that tells whether the stack is empty. After a push, it's false. And [24] make empty is a program that makes the stack empty, so after make empty, [25] is empty is true.

[26] Here's a really weak version of stack theory. In the versions we had so far, a push followed by a pop implied ok, which means everything is back the way it was. In this version, any balanced number of pushes and pops puts top back the way it was, which is all we care about when using a stack, but it allows other things to change. We don't need to implement balance. It's not something a user of the theory can do. It just helps us to write the axioms. This stack theory could be extended with [27] a counter, that counts pushes and pops. I've added operation start, that zeroes the counter, and push and pop each add 1 to the

counter. The earlier versions of stack theory didn't allow this because a push followed by a pop had to put everything back the way it was. So that's another reason for making theories weak. You can extend a weak theory in more ways than you can extend a stronger theory.

Let's move on to [28] queue theory now. In the program version, we have a program [29] make-empty-queue, and a binary expression is-empty-queue. After executing make-empty-queue, is-empty-queue is true. We have procedure [30] join. If we join x onto the queue, and it was an empty queue, then front is x and is empty queue is false. If [31] we join x onto a queue, and it was not empty, then front doesn't change and the queue is still nonempty. [32] If we have an empty queue, then a join followed by leave is the same as make empty queue. And [33] if we have a nonempty queue, the a join followed by leave is the same as the other way round. And that's program queue theory. Theory design is not easy. You have to think of all the relevant situations and what should happen. One thing that helps is to see if you can find the value of each expression after each operation. Sometimes you don't want to know, like after make empty queue and then leave. But it makes a good check list.

[34] Now I want to present an unusual version of tree theory. Imagine a tree that's infinite in all directions. If you keep going down, you never come to a leaf. If you keep going up, you never come to the root. It can be implemented on a computer because you can visit only a finite part of the tree in a finite time, so only a finite part of the tree needs to be stored in memory. Actually the tree can be built as it is being used, so only the used part of the tree is built. But conceptually, it's infinite. Imagine that you are located somewhere in the tree. [35] Variable node tells you the value of the item where you are, and you can [36] change that value with an assignment to node. [37] Variable aim tells what direction you are facing, and you can [38] change that direction with an assignment to aim. Up means towards the parent node, left means towards the left child node, and right means towards the right child node. [39] Program go moves you to the next node in the direction you are facing, and turns you to face back the way you came. That's the theory, informally. You can wander around the tree assigning node values, and whenever you revisit a node, you can see what value is stored there. To present the axioms of the theory, I need [40] an auxiliary specification. It's not a program for the user and it doesn't have to be implemented. It just helps me write the axioms. It says go off and do whatever you want and then come back. Its purpose is so we can say that when you get back, you'll find the node value just the way you left it. [41] Here are the axioms. I won't read them in detail, but look at the [42] second one. That's the one that says if you go off and do whatever you want and then come back, you'll find the node value the way you left it. [43]

When you implement this theory, and you want to prove that the implementation is correct, you have to propose a definition for work, using the implementer's variables, just like the operations of the theory. But work can be any specification; it doesn't have to be a program. It just has to allow you to prove the axioms of the theory.

Years ago, people thought that programs weren't mathematical expressions. So if you want to prove anything, you have to use functions for the operations on data types. But they programmed using imperative state-changing operations, like assignment statements. So there was a gap between the programs and the theories. Some people thought the best way to close the gap is to program with functions, and there are some very nice functional programming languages. The other way to close the gap is to use program theories, and that's what we did today.