[talking head] This lecture is about data transformation. That means replacing some variables by other variables. Suppose we have specified a data structure and its operations as a program theory. That means we have users variables and implementers variables. Our choice of implementers variables might be to make the explanation clear to the user. For example, we might explain that a stack is just a list where you add items to one end and remove items from that same end. That doesn't mean we really have to implement it as a list. We could replace the list we used in the explanation with some other structure. We just have to implement it so it seems to the user to behave the way we explained it.

Or maybe we have already implemented the data structure, and we want to reimplement it differently. Maybe a different structure would have better performance. Or maybe the structure we have isn't good for some new operations we want to add, so we have to replace the data structure with something else. An implementer can always change the implementers variables in any way that still provides the same operations to the user.

[1] Let the user's variables be u and the implementer's variables be v. U might stand for several variables, not just one, and v might be several variables. And we want to replace the implementer's variables by [2] new implementer's variables, which we'll call w. The way we do it is to write a [3] data transformer that relates the old and new variables. And we use the transformer to [4] transform all the specifications and programs that talk about the old variables into specifications and programs that talk about the new variables. I'll explain this formula in a minute. I think maybe [5] this picture will help. Here's the story. The user cannot access the implementer's variables; can't see them and can't change them. But suppose the user knows about them. Maybe someone explained the operations by talking about the implementation, or maybe the user read the implementation. So the user knows about variables v, and knows about some specification S that talks about v and v prime. If S is a program, then executing S changes initial state v into final state v prime. So that's the top line of the picture. But after the data transformation, the state will actually be w, which is related to v by the transformer D. That's the left side of the picture. The user is not supposed to notice any difference, and that's because [6] whatever state w we start with, [7] there is a related state v for the user to imagine that we're in. [8] This specification is supposed to take us across the bottom line of the picture, from w to w prime. In this formula, D talks about both v and w. D prime talks about v prime and w prime. And S talks about v and v prime. But the whole formula, well it doesn't talk about v and v prime because they're local. So it just talks about w and w prime. It says: [9] whatever state v the user is imagining that we start in, [10] related to w, [11] there is a final state v prime, [12] related to w prime, for the user to imagine as the result of [13] S. So the user won't be surprised. The user can keep on imagining we have state space v, but really we have state space w.

We need an example. [14] This is the simplest example I can think of. The user has a binary variable u, and the implementer has a natural variable v. And there are [15] three operations. Operation zero sets v to 0. Operation increase increases it by 1. And operation inquire asks if it's even or odd, and assigns the answer to the user's variable. So that's what the user can do. Maybe start with zero, then increase a few times, and then see if it's even or odd. Now we get our brilliant idea. We can save space if we reimplement using a binary variable instead of a natural variable. So [16] the new implementer's variable is binary variable w, and the data transformer is w equals even v. Now we have to transform the operations. And we'll start with zero. [17] This is the formula for the transformed operation. The data transformer D is [18] w equals even v, and D prime is w prime equals even v prime. And zero is the assignment v gets 0. This assignment is in a world where the variables are u and v. There isn't any w in the old world before the transformation. So [19] the assignment is v prime equals 0 and u prime equals u. And [20] now we can use one point to get rid of the exists v prime, because we have the conjunct v prime equals something. So [21] here's what we get. Even 0 is true, so that can be simplified. But the tricky part is to get

rid of for all v. We can't use one point because there isn't any v equals something in the antecedent. So we're going to use the change of variable law. Not just renaming. That wouldn't help. [22] Here's the law; it's in the back of the book. And there are others just like it for other quantifiers. I haven't been writing the domains, but for v and v prime the domain is nat. The function f is even. So we're changing from a variable with domain nat to a variable with domain even nat, which is bin. [23] Here's the result of this variable change. It changes even v to r, and now [24] we can apply one-point, and that's why I did it. Variable r doesn't appear in the consequent anyway, so [25] one point gives us just this. In the new world after the transformation, the variables are u and w. There's no v. This is [26] the assignment w gets true. v gets 0 has become w gets true.

The next operation is [27] increase. To transform it, just write this formula, where v is the old variables and D is the data transformer. You don't have to think about it. Just write this, and then simplify. Replacing D, D prime, and increase [28] you get this. In the old world, the variables are u and v, so the assignment is [29] v prime equals v plus 1 and u prime equals u. We get rid of v prime by [30] one point because we have v prime equals something. But we don't have v equals something, so we can't use one point to get rid of v. But we can [31] change the variable to a variable r with domain even nat, which is bin, and get rid of r by [32] one point. In the new world the variables are u and w, so this is [33] an assignment. Increase has been transformed to w gets not w.

The last operation [34] goes the same way, so I'll just fast forward [35] to see that it becomes u gets w. The user is imagining that zero sets a natural variable to 0, but really it just sets a binary variable to true. And the user imagines that every use of increase increases the natural variable by 1. And the user imagines that inquire determines if the natural variable is even, but really there's no calculation because the binary variable already is the answer. If this were a larger example with time consuming operations, the user might be amazed at how quickly the answer was ready.

[36] Here's another example. Actually, it's the same example in the reverse direction. The user's variable and the old implementer's variable are binary. The user can set it to true, flip it as many times as desired, and then ask what its value is. The new implementer's variable is natural, and the data transformer says that the new variable is even just when the old variable was true. Now why would we want to replace a binary variable by a natural variable? My real reason is just to show that we can perform a data transformation in either direction, and we can go from a larger space to a smaller space, or vice versa. But here's a plausible reason. A lot of computers cannot address a single bit, so it may take longer to find the value of a bit than to find the value of a larger variable. Sometimes using more space makes a faster algorithm, and that's true for large data structures also. Anyway, [37] here's how we transform set. I'm skipping some detail here, but look at [38] this line. It says, without changing the user's variable, make w be even. That's what set is transformed to. It's nondeterministic because there are a lot of choices for w prime. So the [39] next step is not an equality, but a refinement, as we choose one of the possibilities. Data transformation always produces the specification that shows you all the possibilities.

Now we transform [40] flip, and the second last line says without changing u, pick a new value for w that has the opposite parity. If it was even, make it odd, if it was odd, make it even. Again, there are a lot of possibilities, and we have to pick one.

And [41] finally, we transform ask. On [42] this line it says that u prime should be equal to even w, so the user's variable will tell if the implementer's variable is even or odd. But it also says even w prime equals even w. That means we can change the value of w as long as we don't change whether it's even or odd. Of course the sensible thing to do is to leave it alone, and that's what [43] the last line says. But the data transformation shows you all the possibilities.

These two examples were very small. They were the simplest ones I could think of. So the result of the transformations was obvious, and I wanted you to see that it produces the right answers. But you should imagine large data structures, and transformations that aren't obvious. It works there too. It shows you possibilities you may not have been aware of, and it transforms all the operations on the old data structure into operations on the new data structure.

[44] Here's a more interesting example. A security switch has three binary user's variables a , b , and c . The users assign values to a and b as input to the switch. The switch's output is assigned to c . The output changes when both inputs have changed. More precisely, the output changes when both inputs differ from what they were the previous time the output changed. The idea is that one user might flip their input indicating a desire for the output to change, but the output does not change until the other user flips their input indicating agreement that the output should change. If the first user changes back before the second user changes, the output does not change. Just to make it easy to remember, I'll say person A works input a, and person B works input b. It seems to me we will need [45] two binary implementer's variables, and I'll call them big A and big B. Big A records the state of input a the last time the output changed, and big B records the state of input b the last time the output changed. We need them in order to tell if an input has changed. [46] [47] Here's the operation that person A performs. If person A wants the output to change, person A [48] changes input a. Or maybe person A already flipped input a, but that didn't flip the switch, and now person A is taking it back. Either way, the only thing person A can do is flip input a. And then, [49] if both inputs differ from what they were last time the output changed, then [50] output c flips, and [51] big A and big B record the current values of the inputs. [52] If it's not true that both inputs have changed, then nothing happens. [53] And there's a similar operation for person B. I think that's a reasonable formalization of the problem, and it's already an implementation of the operations.

But we can implement it better. [54] We can replace implementer's variables big A and big B with nothing. And [55] here's the data transformer that does the job. I'm not claiming it's an obvious transformer to try, but I think you'll admit that it's a pretty simple transformer. But first we have to check that it really is a transformer. [56] That means proving for all values of new variables there exist values of old variables such that the transformer is satisfied. But we don't have any new variables this time. So it's just [57] exists big A and big B such that big A and big B and c are all equal. And [58] that's a theorem by the law of generalization.

[59] Now we have to transform the operations. They each start with a flip of an input variable. That doesn't involve the variables in the transformation, so we can just leave that part alone. The other part is the same for both of them, so there's only one thing that needs to be transformed. [60] This is the formula. For all, and then there's the old variables, then the data transformer, then there exists, the old variables primed, the data transformer primed, and the operation we want to transform. — [61] All right. We [62] expand the assignments, the sequential compositions, and ok. The then-part says inputs a and b are unchanged, c is flipped, and big A and big B record inputs a and b. Now we can get rid of all the quantifications by one point. [63] We can get rid of for all big A and big B because they're both equal to c in an antecedent. And we can [64] get rid of exists big A prime and big B prime because they're both equal to c prime in a conjunct. And we get [65] this. — Now, I'm going to use the if-part as context to change the then-part. From the if-part we have [66] a is unequal to c, so a is equal to not c. And I'm going to use that to replace [67] this a by not c. And [68] b is also equal to not c. And I use that to replace [69] this b by not c. So make that change [70]. Now, in the new world there are just variables a, b, and c, so the then-part is just c gets not c, and the else-part is just ok. [71]-- And believe it or not, I can make this even simpler. [72] You might have to check this out for yourself later. If the

part in parentheses is true, it says c gets true is unequal to c, which is the same as not c. If the part in parentheses is false, it says c gets false is unequal to c, which is the same as c. If you're building a circuit for this security switch, it's just 3 exclusive OR gates and one AND gate. That's a really small circuit. Another way to express it is c gets the majority value of *a*, b, and c. And it amazes me that we don't even need to keep track of what inputs *a* and b were the last time the output changed.

That was a big success, and it leads me to ask: why don't we always transform to get rid of all variables and make everything really simple? We'll find out the answer next lecture.