[1] Here's a good example of data transformation. It's a limited queue, which means a queue with a limited capacity to hold items. It can hold at most n items in a list named Q, with an index named p to say how much of the queue is occupied. Those are the implementer's variables. The user's variables are c, which is binary, and x, which is the same type as the items. From the picture, you see that the items occupy the space between 0 and p. If the items were people, they would be facing to the left. The front of the queue is the left end, and the back is the right end. Operation [2] make-empty-queue assigns 0 to p. Operation [3] is-empty-queue tests if p is 0, and reports the result in user's variable c. Similarly [4] is full queue tests if p equals n and reports the result by assignment to c. Operation [5] join assigns x to position p, and then increases p by 1. That works only if the queue isn't full. Operation [6] leave has to get rid of item 0, and it does that by moving all the other items left one place, and then decreasing p. That works only if the queue isn't empty. And finally, operation [7] front just assigns q 0 to x. All these operations are fast, taking no time in the recursive measure, except for leave, which takes time p minus 1. The purpose of the transformation is so that all operations will take no time. [8] The new implementer's variables are R, which is a list variable that holds the queue items, and index variables f and b, which stand for front and back. Here's [9] the old picture again, and [10] here's the new picture. The items are in the segment between f and b. Joining is pretty much the same as before, but leaving is just increasing f by 1, which is a fast operation. As new items join and old items leave, the queue sort of moves to the right, and when it bumps into the end, it [11] just continues from the beginning again, moving cyclically around the list. When f is to the left of b, the items are between them. When b is to the left of f, the items are in the outer portions. So now, with the help of the pictures, maybe we can write the data transformer. [12] There are two cases. The top line is the left picture, and the bottom line is the right picture. In the top line it says that p is equal to [13] b minus f, and that the items of Q between 0 and p are the items of R between f and b. The bottom line says that p is equal to [14] n minus f, that's the front piece of the queue, [15] plus b, which is the back piece of the queue. And [16] the items of Q from 0 to p are the same as the [17] items of R from f to n followed by [18] the items from 0 to b. Now we can transform.

[19] Let's start with make-empty-queue. This is the formula, using the old variables Q and p. Make-empty-queue was just [20] p gets 0, and in the old variables that's [21] p prime equals 0 and the other variables are unchanged. Now we simplify, but let me save a lot of time and just show you the [22] result. It says make f and b be equal. It doesn't matter which index they are, as long as they're equal. So we [23] choose something. That seems good for make-empty-queue.

[24] Now let's try is-empty-queue. That was defined as [25] c gets the result of comparing p and 0. In the variables before transformation that's [26] c prime equals the result of p equals 0, and all other variables unchanged. Now again I'll save time and show you the [27] result. There are 4 cases. The [28] first case says f less than b, which means it's in the inside mode, with the items between f and b, at the start of the operation, and also at the end of the operation. But it doesn't insist that f and b are unchanged. It allows us to shift the items left or right if we want to. I can't imagine why we would want to, but the data transformation shows you all the possibilities, so it says you can shift the items while reporting whether the queue is empty. The [29] next case starts in the inside mode, and ends in the outside mode, in case you want to shift the items past the end. We don't, but there's the possibility. The [30] next case starts in the outside mode and ends in the inside mode, and the [31] last case starts and ends in the outside mode. Aside from all that useless shifting, what does it say about whether the queue is empty? In each case [32], it says — that the queue is not empty. — That can't be right. What about when the queue *is* empty? Something very interesting is happening here. If we look at the [33] cases again, the case [34] f equals b is missing. That would be the case where the queue is empty. An operation

cannot control its input. It can't insist that f is unequal to b. Is-empty-queue has been transformed into an [35] unimplementable specification. That's one of the great things about data transformation. If you make a mistake in designing the new data structure, it transforms to an unimplementable specification, so you can't refine it to a program. It even tries to tell you what mistake you made. So what was our mistake? [36] Let's look at the picture again. There was no problem transforming Make-empty-queue. The problem was with is-empty-queue. We can't tell if a queue is empty. And that's because f equals b could be empty, if we're in the inside mode on the left, or it could be full, if we're in the outside mode on the right. We need a new variable m to say which mode we're in. Here's [37] what the data transformer was. And [38] here's what it should be. I've added m to the top line meaning inside mode, and not m to the bottom line meaning outside mode.

Now [39] we have to transform make-empty-queue again. Skipping all the details, the result is f gets 0 and b gets 0 as before, but this time m gets true. Now [40] is-empty-queue, which went wrong before. This time the [41] missing case is here, we have m, which means inside mode, with f equals b, and c prime is true to report empty. And we *can* [42] refine it to a program: c gets the conjunction m - and - f equals b.

Let me [43] quickly show you the results of transforming the other operations. [44] Is-full-queue is also a test whether f equals b, but this time in the outside mode. [45] join puts x in position b, and then advances b in a circular fashion. [46] leave advances f in a circular fashion. And [47] front just looks at the item in position f. In each case, it's a refinement, not an equality, because we keep being offered the opportunity to shift the items if we want to. But we don't.

[48] Let me sum up a few things about data transformation. [49] You don't have to have the same number of variables after the transformation, and the limited queue illustrated this point because after there were two more than before. Also [50] you don't have to replace the entire space of implementer's variables if you don't want to. We did in the examples, but sometimes you just want to replace some of the variables. [51] And if you do replace part, you can replace another part later. You can do it in parts. [52] What you see people do in practice is to define a new data space, and then reprogram the operations. That way, you don't know if you got it right or wrong. People make mistakes, and they don't know they've done so. So we get bugs. [53] What they should do is write a data transformer. Then you don't have to think up new operations. The transformer tells you all the possibilities. And if you make a mistake writing the transformer, you find out because you get an unimplementable specification.