

[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 first disjunct is the left picture, and the last disjunct is the right picture. [13] The left disjunct says that the items of  $Q$  from 0 to  $p$  are the items of  $R$  from  $f$  to  $b$ . [14] The right disjunct says: or the items of  $Q$  from 0 to  $p$  are the items of  $R$  from  $f$  to  $n$  followed by [15] the items from 0 to  $b$ . Now we can transform.

[16] Let's start with `make-empty-queue`. This is the formula, using the old variables  $Q$  and  $p$ . `Make-empty-queue` was just [17]  $p$  gets 0, and in the old variables that's [18]  $p'$  equals 0 and the other variables are unchanged. Now we simplify, but let me save a lot of time and just show you the [19] result. It says make  $f$  and  $b$  be equal, which is empty in the inside mode, or make  $f$  be 0 and  $b$  be  $n$ , which is empty in the outside mode. So I [20] chose to make them both be 0. That seems good for `make-empty-queue`.

[21] Now let's try `is-empty-queue`. That was defined as [22]  $c$  gets the result of comparing  $p$  and 0. In the variables before transformation that's [23]  $c'$  equals the result of  $p$  equals 0, and all other variables unchanged. Now again I'll save time and show you the [24] result. There are 4 cases. The [25] 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 [26] 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 [27] next case starts in the outside mode and ends in the inside mode, and the [28] 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 [29], 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 [30] cases again, the case [31]  $f$  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 [32] 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? [33] 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 [34] new variable  $m$  to say which mode we're in. Here's what the data transformer should be.  $m$  indicates the inside mode, and not  $m$  indicates the outside mode.

Now [35] we have to transform make-empty-queue again. Skipping all the details, we again have to choose between the inside mode and the outside mode, and again I've chosen the inside mode, but this time we also have to say  $m$  gets true.

Now [36] is-empty-queue, which went wrong before. This time, [37] when we start in the inside mode,  $f$  equals  $b$  tells us the queue is empty. And [38] when we start in the outside mode,  $b$  equals 0 and  $f$  equals  $n$  tells us the queue is empty. And we *can* [39] refine it to a program. —

Let me [40] quickly show you the results of transforming the other operations. [41] In the inside mode, the queue is full when  $f$  to  $b$  is the whole list. And in the outside mode, when  $f$  equals  $b$ . [42] To join a queue that isn't full, no matter whether it's the inside or outside mode, put the new item at the back of the queue. If  $b$  equals  $n$ , we're just wrapping around to the start of the list and switching from inside to outside mode. [43] To leave a queue that isn't empty, no matter whether it's the inside or outside mode, move the front pointer along one place, unless it's at the end, in which case move it to the start and switch to the inside mode. [44] The front of the queue is the item at position  $f$ , unless  $f$  equals  $n$ , in which case it's the item at position 0. You can write these programs without data transformation, but data transformation is automatic, and tells you all the possibilities, and guarantees the correct result. 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.

[45] Let me sum up a few things about data transformation. [46] 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 [47] 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. [48] And if you do replace part, you can replace another part later. You can do it in parts. [49] 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. [50] 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.