[1] We have been writing specifications about the initial and final values of variables, and refining the specifications to obtain programs. But we haven't been concerned with how long a computation takes. Now we consider time. And all we have to do is introduce a time variable t. Up to now, the state of a computation has been the memory contents. But now we consider time to be part of the state, but not part of memory. To observe the state of a computation, you observe the memory contents to see the values of variables like x and y, and you look at the clock on the wall to see the value of variable t. Like the other state variables, t is really [2] 2 mathematical variables t and t prime. t represents the time at which execution starts, and t prime represents the time at which execution ends. If execution never ends, the value of t prime is infinity. [3] The type of t and t prime could be the extended naturals, which means the natural numbers extended with infinity, or the extended nonnegative reals. For some purposes we may want natural valued time, and for other purposes we may want real valued time.

Now that we have a time variable, the definition of [4] implementability changes slightly. It used to say – for all sigma, there exists sigma prime such that S is satisfied. Now sigma and sigma prime include time as well as memory variables, so it says – for all initial states of memory and starting times, there exists a final state of memory and final time to satisfy S with nondecreasing time. A specification is unimplementable if it requires time to go backwards.

[5] There are lots of different ways that we might choose to account for time. I'm going to talk about 2 of them. The first is called real time, maybe because it uses an extended real valued time variable, or maybe because it accounts for the real, actual execution time. You would have to use this for a program that controls a chemical or nuclear reaction or a heart pacemaker, because there the timing is critical. Here's what you do. Whenever you have an [6] assignment to a memory variable, like x, put with it another assignment increasing t by however much time it takes to execute the assignment to x. It doesn't matter whether you put this increase before or after the assignment to x; I'll put it [7] before. This new assignment to t isn't something the computer executes. It just accounts for the time taken to execute the assignment to x. Or, I could say, that the computer does, in a sense, execute the assignment to t, but only in the sense that it takes time to execute the assignment to x. To know how much the increase is, you have to know what machine instructions the compiler generates, and the execution time of these instructions on the computer that will execute the program. A compiler could easily have a table of times, and tell you exactly the time increase for each assignment. Whenever you have an [8] if-then-else-fi, you put a [9] time increase to account for evaluating the condition and branching instructions. Inside P and Q there will be time increases that depend on what's inside P and Q. But for the if, we just need the time increase that accounts for the condition and branching. Whenever there's a [10] specification, which is essentially a procedure call or method call or function call, you need an [11] increase to account for that. For calls that are inlined, or macro expanded, that's no time at all, so we can leave it out. For many other calls, it's just the time for a branch, or jump instruction. For a very few calls, it's the time to push a return address on the stack and then branch, plus the time to pop the return address and branch back later.

Also, any specification you write can talk about time in the same way that it talks about other variables. For example, [12] t prime equals t plus f of sigma, where f of sigma is some function of the state, says that the final time is the initial time plus f of sigma, so it says that f of sigma is the execution time. Or maybe you want to specify that the execution time is bounded above by some function of the initial state, so that's [13] t prime is less than or equal to t plus f of sigma. Or you could specify [14] a lower bound, if you want the computation to take at least f of sigma. But these are just examples. You can say whatever you like about the execution time.

Let's look at [15] an example. In this example, x is an integer variable. For the execution, you don't need to know what P is. What happens is - if x equals 0, the computation is done. Otherwise x is decreased, and then repeat. If x is nonnegative, it's counted down to 0, and that's the end. If x is negative to start, then it keeps being decreased forever. I can't do the real time calculation until the compiler tells me about the code that's generated, and until I know the instruction times. But just to show the calculation, let me suppose that the test and branching requires time [16] 1. Maybe that's 1 nanosecond, or 1 something. The program ok doesn't take any time, so I won't put any increase there. Suppose the assignment also [17] takes time 1. And let's say the call, which is just a branch back, also [18] takes time 1, just for easy calculation. We can prove this refinement for a [19] variety of different choices for P. We can prove [20] x prime equals 0, just talking about the result, ignoring time. Or, [21] we can just talk about the time, ignoring the result. If x starts nonnegative, then it takes time 3 x plus 1, and otherwise the computation ends at time infinity, which means it never ends. You can see the 3 x plus 1 easily enough, because there are 3 increases to t each time round the loop, plus 1 more before you know you're done. The [22] next specification for which we can prove the refinement - talks about both result and time. It says if x starts nonnegative, then the result is 0 and the time is 3 x plus 1. Otherwise the time is infinite. Now that one seems to me to describe the computation perfectly. The [23] last one says that x ends with value 0, and if x started nonnegative then it takes time 3 x plus 1, and if x started negative then it takes infinite time. That one's a bit funny, because it says x ends with value 0 even if it takes forever. That's a little less reasonable than the previous specification. But the last one is better in one way. It's a conjunction, and you can prove each of the conjuncts separately, and then the law of refinement by parts says that the conjunction is also proven. So it's better for proving. My suggestion is to use the one that's easier to prove, which is the last one, and just ignore anything it says about the values of variables at time infinity.

So that was real time. [24] Now here's another way to measure time, called recursive time. This way is less accurate than real time, so you wouldn't use it if time is critical, but for this measure you don't have to know anything about what machine instructions are compiled, or what computer will execute it. Each recursive call costs time 1, and all else is free. [25] Here's the same example we just had, but in this measure, the only time increment is the one for the call. And this refinement can be proven for [26] these specifications. They're the same as before, except that 3 x plus 1 is now just x. The multiplicative and additive constants disappeared. For nonnegative x, it takes time x. For negative x, it still takes time infinity. The recursion in this example is direct, [27] but recursions can also be indirect. The general rule for recursive time is that every loop of calls must contain a time increment.

Now [28] we're going to do an example in detail. Again, x is an integer variable, and it's the only variable except for time. If x equals 1 then ok, else x gets div x 2, which divides x by 2 rounding down if x was odd, then the clock goes tick, and then repeat. The computation repeatedly divides x by 2 until x is finally 1. The specification we're going to prove is [29] x prime equals 1, and if x starts positive then the time is bounded above by log x, and that's the base 2 logarithm, otherwise the computation takes forever. And we're going to prove it by parts. And before we do that, I'm going to rewrite the if [30] as a conjunction, so now we have 3 parts to prove. One is that x is finally 1, another is that if x starts positive then it takes at worst log x time, and the last part is that if x starts nonpositive it takes forever. So [31] here are the 3 refinements that we have to prove, each one dealing with part of the full specification. Notice that the part occurs both on the left side, and again at the end of the right side. Now, each of these refinements is an if-then-else-fi, so it can be proven by cases, if you remember the law of refinement by cases. The first case is: x equals 1 and ok, and the other case is: x is not equal to 1 and the else-part. The first refinement becomes [32]

these two cases. The second refinement becomes [33] these two cases. And the last refinement becomes [34] these two cases. So from 3 parts and 2 cases each, we now have 6 refinements to prove. We have to replace [35] each of the ok's with its binary expression, which is [36] x prime equals x and t prime equals t. ok means that all variables are unchanged, including time. Now [37] look at the expression inside the parentheses here. It can be simplified by 2 uses of the substitution law. In x prime equals 1, replace t with t plus 1, which is trivial since there isn't any t to replace, and then replace x with something, also trivial because there's no x to replace. So it simplifies to [38] x prime equals 1. Next, [39] we simplify the piece inside these parentheses. This time there is a t to be replaced, and there are 2 occurrence of x to be replaced. It should say div x 2 is greater than or equal to 1 implies t prime is less than or equal to t plus 1 plus log of div x 2. So [40] here we go. And [41] the bottom one has an x to replace, but no t to replace. And we get [42] div x 2 less than 1 implies t prime equals infinity. Now there are no programming notations left. It's all just binary calculation and number calculation from here on. Let's go from the [43] top again. In the antecedent we have [44] x prime equals x, and [45] x equals 1. So by transitivity we have [46] x prime equals 1. So [47] that one's done. Of course, if you're doing this for homework, you can't be pointing to things and talking about them, so you have to write it properly as a proof in the usual format, with the hints over on the right side to justify each step. But I'm just talking you through it. In the [48] next one, we have [49] x prime equals 1 in both the antecedent and the consequent, so by specialization [50] that's done. On to the [51] next one. Let's see, it's a little more complicated, so maybe I better do this one properly. Let's write it at the top [52] of a clean page. In the main antecedent, it says x equals 1. One of the context rules says that if we're changing a consequent, we can assume the antecedent. So that means we can change the consequent by changing its xs into 1s and change t prime into t. [53] And this can be simplified because 1 greater than or equal to 1 is true, and true implies something is just that something, so that disappears. And log of 1 is 0, so that disappears. And t less than or equal to t is true. [54] And anything implies true [55]. So that one is done. [56] That's the first three. [57] On to the next one, and again I'll start a fresh page [58]. This one is the hardest. It has lots of implications in it, and I'm going to try to reduce that by using the law of [59] portation. On the right side of this law there are 2 implications, and on the left side only 1. So I'll match our expression with the right side like [60] this. Maybe I can make it clearer by rewriting portation like [61] this. Our expression matches the top line of portation. What I say to myself to recognize the pattern is – something implies an implication. a implies the implication b implies c. And what we do with it is – we keep b implies c. And *a* moves into its antecedent, where it becomes a conjunct. a and b implies c. So our expression becomes [62] this. [63] Now we can simplify a bit. [64] x greater than or equal to 1 and x not equal to 1 becomes x greater than 1. Where it says [65] div x 2 greater than or equal to 1, if half of x is greater than or equal to 1, then x is greater than or equal to 2. Or I'll just say [66] x is greater than 1, again. Now we can use [67] discharge, which says a and a implies b is the same as a and b. In our expression [68] there's a and a implies b. So that gets simplified to [69] this. We're getting there. [70] Now we have a conjunction implying something, and we have to use portation again but in the opposite direction from before. We have to pull x greater than 1 out of the antecedent, and make it an antecedent all on its own. [71] And the point of that portation was to be able to use the [72] connection law. We have t prime less than or equal to something implies t prime less than or equal to something else. Connection says that's true if the something is less than or equal to the something else, so we simplify but [73] this time we have reverse implication in the left margin. But it's all right because we're aiming for true. Now [74] subtract t plus 1 from each side of less than or equal to. And a law of logarithms says subtracting 1 from a log is the same as [75] taking the log of half the value. And log is monotonic, so we can

instead [76] compare its arguments. And finally, div is the same as dividing and rounding down [77]. So we're done the hard one.

That [78] leaves 2 more refinements to prove. Putting [79] this one on a clean page [80] I see it has something implies an implication, so I'll use [81] portation and get this. x less than 1 and x equal 1 is [82] false, and [83] false implies anything. [84] [85] One more to go. On a [86] clean page. Looks right for portation [87]. Now [88] if x is less than 1, we don't need to say that x is unequal to 1. Also, [89] if x is less than 1, then half x is also less than 1, so we can discharge this antecedent [90]. Now we have t prime equal infinity implying itself, so by [91] specialization we're done. And that's [92] the end of that story.

[93] There's one story left this lecture, about termination. If a customer gave you this specification - x prime equals 2, it would be easy to refine. [94] x gets 2, and that's done. But [95] there's an even easier refinement. [96] x prime equals 2. Refine it by itself. Implication is reflexive, so that's a theorem. The execution is an infinite loop. It calls itself. The customer didn't say anything about when the result is wanted, and this refinement delivers the result at time infinity. The customer can [97] complain only when they observe behavior that's contrary to the specification. That means the customer could complain if the computation delivered a final value for x other than 2. But it won't. So there's no reason to complain. I really should have put a [98] time increment before the recursive call. But it's still a theorem, because there's no t to substitute for. So maybe the customer strengthens the specification [99] to say they want x to be 2 and they want the computation to finish at a finite time. The problem with this specification is that it's [100] unimplementable. To be implementable, there must be final values to satisfy the specification with nondecreasing time. And if the start time is infinity, then time would have to decrease in order to end before time infinity. Now I know the customer wants to start at a finite time. But if you could implement this specification, the customer could [101] write an infinite loop, that's easy, and then a dependent composition, and then this specification. And this part [102] starts after the infinite loop, so it does start at time infinity, and somehow ends before time infinity, which is obviously impossible. [103] Anyway, it's easy for the customer to make the specification implementable. Just say [104] x prime equals 2 and if t is less than infinity, then t prime is less than infinity. Now [105] here's the surprise. We can still refine this one as an infinite loop. Even with t gets t plus 1 in there. Because t plus 1 less than infinity is the same as t less than infinity. The customer can complain whenever they see behavior that contradicts the specification. The negation of this specification is – [106] x prime not equal to 2 – or – t less than infinity and t prime equal infinity. So the customer can complain if they see a final value of x that's not equal to 2, which they won't. The customer can also complain if they start the computation at a finite time, which they will, and it runs forever. But the customer can never complain that the computation has taken forever. If the specification does not state an upper bound on the time, then you cannot say it has taken too long. If the customer gives you [107] this specification saying they want the result within 1 second, if that's the agreed unit of time, then you [108] cannot refine it with a loop, because that isn't a theorem. You have to refine it like [109] this. The moral of the story is: it's worthless to say you want termination, or to say that a computation terminates, without giving a time bound. It's just like loaning someone money, and they promise to pay you back sometime, but they don't say when. They can always say I will, and they never have to pay. If you're going to bet with someone whether something will happen or won't happen, and there's no time bound, then bet it will happen. If it does, you win, and if it doesn't, you just say wait longer. You can't lose.