[1] Hello. I'm Eric Hehner, from the computer science department, of the University of Toronto. This talk is called Practical Predicative Programming Primer. So I suppose I should start by saying what predicative programming is, and why you should care. Predicative programming belongs to the branch of computer science called formal methods, which refers to proofs about programs - mostly proof that a program is correct according to a specification. The proof can be made after a program is written, or while it is being written. If you prove each programming step is correct at the time you make it, you save building the rest of the programming on a wrong step you made earlier, and it's harder later to find the error. And formal methods can help you program, as I hope to show you in this talk. But there are a lot of programs written without formal methods, so we have to be able to verify programs after they are written also.

There are several well developed formal methods. This one, predicative programming, is the one that the American Presidential Science Advisory Committee called the best one.

[2] In predicative programming, you first have to decide what quantities are of interest, and introduce a variable for each one. You could be interested in any of these things; there's no right or wrong choice here. To start with, I'm going to look at [3] a traditional choice: just the initial and final values of program variables, and I'll use unprimed and primed variables for them. Now [4] a specification is a binary expression because when you observe something, either it satisfies the specification or it doesn't. That's two possibilities. What we're specifying is computation, or computer behavior. [5] And a program also specifies, or describes computer behavior. The difference is that you can give a program to a computer and get it executed -- because a program is a specification in a limited language for which you have an automatic translator to machine language. But specifications in general can use whatever notations you find helpful. [6] If we're just talking about initial and final values of program variables, then this is the formula for implementability of a specification. Whatever initial values we provide as input, there must be final values for the computer to produce as output satisfying the specification. [7] P refines S means that all behavior satisfying P also satisfies S, so it's just implication. A particular specification worth mentioning is [8] ok, which says that all values are unchanged. Some people call it skip. And [9] assignment looks like this. ok and assignment are just binary expressions. [10] So is if-then-else-fi; it's a disjunction of conjunctions, or equivalently [11] a conjunction of implications. And [12] sequential composition identifies the final values from S with the initial values from R. I don't want to spend time on this; I just want to say that you can prove lots of laws, and one of the most useful laws is [13] the substitution law, that says if you have an assignment followed by any specification, that's the same as the specification with a substitution. [14] In the old Hoare style of program proving, we write a precondition and a postcondition for a block of code like this. [15] In the predicative style, we might write this, because a block of code is also a binary expression. This isn't just a different syntax, because it obeys all the laws of binary algebra. For example, [16] it can be written equivalently this way. Rules like strengthen precondition and weaken postcondition are just ordinary binary laws about strengthening the antecedent and weakening the consequent of an implication. The reason I wrote it [17] this way is [18] that the first implication is a specification, and the block is an implementation, and the reverse implication is refinement. In general, the implementation could be [19] a block of program statements, or it could be [20] a specification that still needs refinement. That's stepwise refinement of a specification through a sequence of specifications, and in the middle of the sequence [21] we have a mixture of specification and program, so this is a true generalization of the old Hoare style. And it's a generalization also because the specification

could be [22] a precondition postcondition pair, but it could also be [23] a disjunction, which is a nondeterministic choice. It could be a [24] choice between pre and postcondition pairs, or it could be a [25] conjunction, and that's really important because you can't write specifications for real problems in one monolithic piece. And the best news is that if you prove each of the parts separately, you get their conjunction for free. It could be [26] a conjunction of implications, and I want to point out that you can't rewrite that as an [27] implication of conjunctions. You can't write [28] this specification as a [29] precondition and postcondition pair because you would lose the fact that precondition 0 is needed just for postcondition 0, not for postcondition 1. In general, a specification can be [30] any binary expression. So this is a generalization of the old Hoare style, and also a simplification because [31] we don't need new rules for each different kind of block and each different kind of specification. Refinement is always just implication.

[32] Here's an example, cubing a natural number without using exponentiation or multiplication. The example doesn't matter; this won't even be the best way to cube, but it's a good way to show how predicative programming works. We want the final value of variable x to be n cubed. [33] One way to refine this specification is [34] to assign n to x, then multiply x by n, then multiply it by n again. The assignment is program already, but the rest isn't. Before we refine the rest, we should [35] prove this refinement. I don't have time to go through all proofs in detail. But you can quickly see the proof length, and the hints on the right say what's involved. A one-point law, the substitution law, and some arithmetic simplification because the problem is arithmetic, and the proof must always involve some domain knowledge. [36] Now [37] we have to refine this specification, which is used twice. [38] Since we don't have multiplication, we have to [39] accumulate a sum, which means starting at zero, and then adding. [40] For proof, start with the right side, [41] and use the substitution law, replacing x with 0 in the final specification, [42] and use it again to replace y with x. Now we [43] simplify, and that completes the proof. Substitutions are trivial; simplification requires domain knowledge. [44] Now we have one new specification that needs refinement. [45] And to refine it, I look at [46] two cases. If y equals 0, there's nothing more to be done. If not, we add one more n, decrease the count by 1, and repeat. [47] Proof, starting with the right side, we use [48] substitution twice, and [49] simplify. Then [50] we use the definitions of if and ok, and finally [51] show that each disjunct implies the specification we're refining. [52] There aren't any new specifications needing refinement, so we're done. Now there are two ways to look at what we have here. One way is that we have three implications that are proven. That's the way the prover looks at it. The prover sees some mathematical expressions, and the program parts are just binary expressions in disguise. To a compiler, the program parts are familiar, and the nonprogram parts are just long identifiers. To see it the compiler's way, let me [53] replace those long identifiers by short ones. And now it's [54] a tiny step to a standard execution language. If you don't like all these calls, maybe because you fear the unnecessary stack activity, you can replace the call to R with a goto [55]. Remember, this isn't my programming language; that's at the top of the page. This is my object code, or assembly language. But if you hate gotos, you can rewrite it as [56] a while loop. And you can replace each of the calls to Q with [57] the body of Q if you want to. But back [58] to the program, and putting back the specifications [59], I want to talk about execution [60] time now. So I add a [61] time variable t. We could talk about [62] real-time if we know how long each operation takes. We don't have to know exactly; upper or lower bounds might be enough. Or we could just [63] count operations. In different circumstances, different measures of time are appropriate. I'm going to be more abstract and [64] count iterations. To make some room for the time calculation, [65] let me just spread it out [66] a little. I need to put t gets t plus

1 somewhere in the loop. I'll put it right [67] here. [68] Now, in the specification [69], we can say something about time. We can say an upper bound, or a lower bound, or both, and since this is an easy example we see that t goes up exactly as y comes down. So [70] the final time is the start time plus y, which says that the execution time is exactly y. And the [71] proof, starting with the right side, is exactly the same as before: substitute, simplify, replace if and ok, and show that each disjunct implies the specification. Proving time is exactly the same as proving results, and that proves more than termination. [72] We have to put the timing [73] here also. [74]. Now we prove this refinement [75]. Just two substitutions and a simplification, and we calculate what timing goes [76] here. [77] Now we know what timing goes [78] in these specifications [79], and we [80] [81] prove the top refinement, starting with the right side. And that tells us [82] what the overall timing is. I wouldn't have guessed that execution time, but it was easy to calculate it.

The execution time is n squared plus n. Maybe we can do better. [83] Let's try for linear. Here's the problem again, and I want a solution of [84] this form. If n is 0, it's easy; just set x to 0. Otherwise decrease n, and go round the loop. Well, that's not right as it stands, because the else part simplifies to x prime equals n minus 1 cubed, which is not what we want; it's too small. So [85] we have to add something to x. This is where we take a step up, and we have to add [86] the difference between n cubed and n plus 1 cubed, which is [87] 3 n squared plus 3 n plus 1. But there's a problem. For one thing, we don't have multiplication or squaring. So I'll [88] introduce a new variable, and add [89] y, and I'll make it be the right amount by [90] augmenting the specification just before it. [91] There. But this specification was supposed to be a loop, or recursive call, so I have to do the same thing [92] here. [93] But now that I've changed the specification, I have to check the refinement. If n equals 0, we have to set x to 0 as before, [94] and now we have to set y to 1 also. And in the else part, we've made sure x has the right value, but now [95] y has to step up. By how much? That's [96] this difference, -- which by high school algebra is [97] 6 n plus 6. And I'll add that amount by [98] introducing variable z [99], and I'll make z have the right value by [100] augmenting the specification again. [101] And now I have to check the refinement. If n equals 0 we have to set z to [102] 6. And in the else part, we have to [103] increase z. By how much? [104] from 6 n plus 6 to 6 (n plus 1) plus 6, -- which is [105] 6. And that's [106] easy. [107] So we're done. Well, I can add a time variable and prove the time is n, but I won't do that right now. What I wanted you to notice is the role played by the formal method. I started with a really simple idea: to decrease n towards 0. After that, everything was dictated by the formal method. I didn't say to myself, I think I'll initialize a variable to 6, and increase it by 6 each iteration. Why 6? Because that's what the formal method said. I could show you other solutions to the cubing problem, but I guess you're sick of cubing right now, so I'll move on.

[108] In the cube example, I didn't use any loop constructs, although I did form loops. In existing programs there are many different loop constructs to deal with. One of them is the while loop. [109] Here is the standard semantic definition. A while loop satisfies this equation. while b do P is equal to -- if b then P and the loop again, else ok. This equation has many solutions, so to finish the definition [110] we say that a while loop is the weakest of the solutions. This says: if W is a solution of the preceding equation, then W implies the while loop, so the while loop is the weakest solution. In the jargon, a while loop is a least fixed-point. This definition is completely unusable for programming. I don't even think it's adequate theoretically, because [111] it doesn't allow us to prove that if the loop body doesn't decrease x, then the loop doesn't decrease x. And if you want to include time, which I do, [112] it becomes more complicated. [113] What textbooks use, at least those that use formal methods, is the old Hoare style, with preconditions, postconditions,

invariants, and variants, to prove loops correct. And [114] here's the while-loop rule, which programmers don't use directly, but instead they [115] divide it into these six parts: finding an invariant, checking that it's true initially, check that the body maintains it, finding a variant, checking that it's decreased, and checking that the postcondition is established. [116] In predicative programming, a loop has to [117] refine a specification, W let's say. And that's the [118] same as saying that W is refined by: if the condition is true, then do the body and repeat; otherwise stop. And we already know about if and semicolon and ok, and that's all we need. All the parts of the old while-loop rule are included here. We might have [119] an until loop - do P until b. I don't remember the old Hoare proof rule for an until loop - I guess there was one. But it's easy to see the predicative programming rule. It has to [120] refine a specification, let's say R, and that's the [121] same as saying that R is refined by: P followed by if b then we're done else repeat. [122] And how about a loop with an exit in the middle. It [123] refines specification L if we can prove [124] this. --- [125] First execute A. [126] Then if b is true we're done. [127] Otherwise execute C and [128] repeat. [129] Here's an example with both 1-level exits and 2-level exits. We need a specification for each loop, so I've used Q for the inner loop. I'll say more about how to formulate these specifications, but it's quite straightforward to say what refinements must be proven for any looping structures. I really have no idea what the old Hoare style of proof rule would be for this.

And we [130] aren't limited to well-structured loops. Here's an example with real spaghetti code. You don't need to read this. I'll show [131] you the branching with a flow chart. You can see that there are even branches into the middle of a loop. [132] Here and here. To verify this program, [133] we just need to prove [134] these refinements. They come straight from the program. An important point is that when we're proving the A refinement, we don't need to know how B is refined. And when we're proving the B refinement, we don't need to know how C is refined. And so on. [135] To prove them, we have to [136] have the specifications. In particular, we have to know that [137] this program is intended to do exponentiation to a natural power. Z prime equals x to the power y. For the others [138] we see that B is invoked when y is odd; C is invoked when y is even, and D is invoked when it's both even and greater than 0. And in each case we are accumulating a product. And finally, [139], E says we're done.

[140] Now I want to talk about how to invent the necessary specifications, and I'll use the familiar binary search example to make my points. Find x in list L. In binary search there's an [141] interval, from h to j, where we're still searching, and it shrinks. [142] Here's the program. If we were doing an old Hoare style proof, we would have to spread it out [143] to make room for assertions. And [144] here they are. I'm not going through this because it's not a good way to do the proof. I'll just point out [145] the invariant, which has to be true at the start of the loop and after each iteration. It says that x does not occur in the outer parts of the list, from [146] 0 to h, and from j to the end. It says what we know to be true at this point in the execution; it's a summary of what we have found out so far. And [147] at the end we have the postcondition, which says [148] if x is anywhere in the list, then [149] it's at position h, and if it isn't anywhere then position h isn't x, so we can tell whether it occurred or not. And [150] there's a variant, but I haven't included that proof obligation here. Now, [151] clearing away the assertions, in predicative programming, [152] I need a label for the whole thing, and a label for the loop. And the [153] proof obligations are these two refinements. [154] A is refined by: h gets 0, j gets the length of the list, and then B. In this refinement, we don't care how B is refined. The proof is just two substitutions. [155] And B is refined by the loop. And that means B is refined by an [156] if. This proof is substitution, simplification, and replacing if and ok by their definitions. Of

course [157] we have to know what A and B are. [158] A specifies the whole search. It says we're searching the whole list to find x. And now here's the main point. [159] B is just like A, except it says we're searching the [160] remaining part of the list to find x. And if we forget [161] this bit, the proof of the refinement fails, but it fails in a way that tells us that we need this bit. [162] An assertion summarizes what's done, and says what's true at some point. A specification says what's still to be done. It says what is the purpose of a block of code. They're quite different. There's no invariant here. For 25 years I've listened to speakers say - of course we need an invariant; everybody knows that. And I sit there choking because I know that we don't need an invariant. [163] Now let's add time. If we're just counting iterations, we put [164] t gets t plus 1 in the loop. We can say [165] that the time is bounded above by the log of the length of the list, and for the loop it's the log of the length of the remaining interval. The proof is just substitutions and simplifications. We don't need any proof of termination because we have an upper bound on execution time, and that's better than proving termination. And for free we have [166] the conjunction of the things we've proven.

[167] Comparing assertions with specifications, and under the name assertion I'm including preconditions, postconditions, and invariants. A variant isn't an assertion but I've tossed it in there too. The first point is [168] that an assertion says what's true at some point in the code, and a specification says what a block of code is intended to do. I conjecture, but I haven't got any good experimental evidence, that programmers find it easier, and prefer, to write comments that say what the following block is for, than to say what's true at strategic points. [169] And there are more strategic points needing assertions than blocks needing specifications. [170] And if you've written a prover for the assertion style, you know about the problem of specification variables: they should be local, or bound to each triple where they are used, not free variables. [171] Specifications are good for all kinds of control constructs, and [172] all kinds of observations, including nonterminating computations and communications. Just to illustrate [173] nontermination, here's a nonterminating loop that repeatedly adds 1 to x. Termination means finite execution time. Nontermination means infinite execution time. So I need a time variable, [174] and I'll count iterations by putting t gets t plus 1 in the loop. And I want to know what [175] specifications are refined by this loop. So I [176] replace the loop with an if, and I can [177] simplify that, and I could have written that last line in the first place. So now let's try to prove [178] that the execution time is 23, just to pick an arbitrary amount. We [179] use the substitution law, replacing t by t plus 1, and there's no x to replace, and that's the same as saying [180] 23 equals 24, which is [181] false. And we would get the same result if we used any finite function of the state as our execution time. Now let's try [182] infinity for the execution time. Again, using [183] the substitution law, we end up with [184] infinity equals 1 plus infinity, which is [185] true in my algebra. So in predicative programming you can prove nontermination. In a more interesting program, you can prove that there's termination in some circumstances, and nontermination in other circumstances.

[186] My last example is a nonterminating program that reads infinite sequences of coefficients of power series, and produces an infinite sequence of coefficients of the product power series. I don't have time for any detail here. If you want detail, it's all in my book. I'll just [187] flash the program at you, and point out that it has [188] parallelism, and [189] a channel parameter, and [190] a local channel declaration, and [191] it's nonterminating, and [192] it has dynamic process generation by having parallelism with a recursive call, and [193] it has synchronization, and [194] dynamic storage allocation because it has local declarations and a recursive call. [195] So it has lots of interesting structure. We need only [196] two specifications. One for the whole thing, which just says that channel C outputs

the product of the power series input on channels A and B, and one for the loop. There are only two refinements to prove. The difficult part, as always, is that we have to give the prover the necessary domain knowledge. In this case, that means saying [197] what "power series multiplication" means.

[198] I want to mention quickly that we can do object orientation in all its glory. And [199] we've done probabilistic programming, and quantum programming. [200] If you want to know more, I have a free book that you can download. And an online course that you can take whenever you want. I hope I have sparked your interest enough that you take a look at the book or the course.