[1] Hello. I'm Eric Hehner, from the computer science department, of the University of Toronto. I want to talk to you about the halting problem, which is generally considered to be a foundational result in computing. But first, [2] I'm going back two and a half thousand years ago, to an ancient Greek named Epimenides, who was interested in the nature of truth. [3] Then I'll look at what Kurt Gödel had to say about provability. He was maybe the greatest logician of the nineteen hundreds. Then [4] I'll have enough context to present the halting problem, which was invented by Alan Turing, who was a pioneer of computer science. But I warn you that what I have to say is not the generally accepted viewpoint on the subject.

[5] Now, back to Epmenides. [6] He said all Cretans are liars. And since Epimenides was a Cretan, he was saying that he is a liar. So if his statement is true, then it's a lie. It seems to contradict itself. Epimenides wasn't really claiming that all Cretans are liars. He was really trying to illustrate a point about self-contradictory statements. But actually his statement is not self-contradictory. If there's even one other Cretan, and that other Cretan is a truth teller, then Epimenides' statement is just false. Epimenides is a liar, but not all Cretans are liars. [7] In the Christian bible, Saint Paul completely failed to understand what Epimenides was trying to say. He just accepted his statement, and added a bit of color to it.

[8] Here is the simplest version. It's called the Liar's Paradox, and it really is [9] self-contradictory. I'm calling this statement [10] L, for Liar's Paradox. L is the statement that L is false. Mathematically [11] that's L equals - L equals false. If we try replacing L [12] with true, then inside the brackets, true equals false is [13] false, and true equals false is [14] false, so that wasn't a solution. If we try replacing L [15] with false, then inside the brackets, false equals false is [16] true, and false equals true is [17] false, so that wasn't a solution either. So it's [18] an equation with no solution. As a definition or specification of L [19], we call it inconsistent.

[20] We can make this inconsistency a little more complicated, and a little less obvious, by using two sentences. I name the first sentence B and I name the second sentence G, [21] and mathematically they are two equations with no solution. There's no way to assign values to both B and G to make both sentences true. The first sentence by itself is not inconsistent. It's perfectly possible that the next sentence after it could be true. And the next sentence by itself is not inconsistent. It's perfectly possible that the sentence before it could be false. But together they are inconsistent.

[22] Now I add the last complication: a parameter, so B can say whether any sentence is true, not just sentence G. If you want to make expressions into values so you can pass them as parameters, you have to encode them, and the simplest encoding is character strings. So B of s is true if s is a character string that encodes a true expression, and false otherwise. And G is defined as the string that encodes the expression B of G equals false. Now we can show an example like [23] B applied to the string zero equals zero. Zero equals zero is true, so B of that string is true. And [24] B applied to the string zero equals one. Zero equals one is false, so B of that string is false. We might be fooled by these examples into thinking that B is well defined. The problem arises when [25] B is applied to G. Is B of G true? If it is, then G represents a false sentence, so B of G should be false. If B of G is false, then G represents a true sentence, so B of G should be true. [26] It's the same inconsistency we've had all along. It just got dressed up in fancier clothes.

[27] Now let's look at Gödel's most famous result. [28] It's really the same thing again, but instead of talking about truth, it talks about provability. B of s is true if string s represents a provable expression, and false otherwise. Gödel called this function Bew, B E W, which is short for Beweisbar, which is German for provable, but I just call it B. And G is the famous Gödel sentence, so I called it G. Gödel used a numeric encoding instead of a string encoding, but that doesn't matter. The question is: what is B of G? [29] Suppose it's true. Then [30] G represents a false sentence. [31] If you are using a consistent logic to do proofs, you can't prove a false sentence. [32] So B of G should be false. [33] If B of G is

false, then [34] G represents a true sentence. [35] If you are using a complete logic to do proofs, then all true sentences are provable. [36] So B of G should be true. [37] Gödel's conclusion was that if you can define B, then your logic is either inconsistent or incomplete. Most of Gödel's paper was showing how to define B. It's defined to apply to all sentence codes. But there's only one sentence code that he wants to apply it to, and that's G. So he could have made a simpler definition that applies only to G. And for only one sentence, you don't need a sentence encoding. The definitions of B and G [38] could have been like this. B is true if G is a provable sentence. All the arguments stay the same. The conclusion stays the same. As a matter of fact, we don't need two definitions. [39] It's just like the liar's paradox before I dressed it up.

[40] Now we're ready for Turing. And we start with [41] a program that looks a lot like the liar's paradox. This procedure is written in the Pascal programming language, which was popular in the 1970s and 80s. But the choice of language doesn't matter. Any programming language will do. This is a procedure named twist. I haven't quite finished writing it. Between the keywords **if** and **then**, I need to replace (execution of twist terminates) with either true or false. If the execution of twist terminates, I want to replace it with [42] true. But this creates an infinite loop. The execution of twist is nonterminating, so that was the wrong choice. If the execution of twist is nonterminating, I want to replace it with [43] false. But that creates a procedure whose execution terminates. So that was the wrong choice also. [44] The problem isn't any programming difficulty. The problem is that there's an inconsistency in the specification (execution of twist terminates). Well it doesn't sound inconsistent. [45] Someone said to me: Either execution of twist terminates, or it doesn't. If it terminates, use true. If it doesn't, use false. How can there possibly be an inconsistency? But I hope you can see the inconsistency.

Now, I'm going to do the same thing I did with the liar's paradox. I'm dividing this one definition into two. [46] I have made the specification (execution of twist terminates) into a separate function called halts. Well, the function header is there, but the function body is missing. Instead there's a comment specifying what the body should be. Procedure twist is all there. If execution of twist terminates, then [47] halts returns true. That's Pascal's way of saying return true. If halts returns true, that makes twist a nonterminating loop, so halts should return false. [48] If halts returns false, that makes twist a procedure whose execution terminates. So halts should return true. There's no way to write the body of halts to satisfy the specification. The problem is not that halts is a perfectly well defined but incomputable function. The problem is that the specification of halts is inconsistent. [49] I can just hear that same person saying the same thing as before. If you just look at the halts specification, it looks ok. To see the inconsistency, you have to look at halts and twist together.

[50] Now I'm going to make it a little more complicated by adding a parameter, so halts can say whether the execution of any procedure terminates, not just twist. The parameter p is a string that represents any parameterless Pascal procedure. And there's a dictionary of function and procedure definitions so halts can look up things and get their texts for analysis. But it isn't going to help. [51] Now, when someone thinks there's no inconsistency, they can point to procedures where halts works. Like [52] procedure stop, whose execution terminates immediately. We can say what result halts should have when applied to stop. And [53] procedure go, whose execution goes on forever. halts applied to go should be false. So the specification of halts looks all right. But it doesn't matter how many good examples there are. It just takes one bad example to show the inconsistency, and that's [54] twist. If you say halts of twist should be true, then execution of twist is nonterminating, so halts of twist should be false. And if you say halts of twist should be false, then execution of twist is terminating, so halts of twist should be true. This inconsistency is not about what the execution of twist does. It's about what the specification of halts says it should do. We can't program halts because its specification is inconsistent.

There is one more complication. [55] This time, halts doesn't apply to parameterless procedures. It applies to procedures with one string input parameter. This time, halts has two parameters. Parameter p represents the procedure we're applying halts to, and i is the input to that procedure. So halts is supposed to tell us whether execution of p terminates when its input is i. Procedure twist is as usual, but with a parameter added for input. This is now the full blown halting problem, as Turing conceived it. And [56] here is Turing's argument. Assume that halts is computable, and that it has been programmed according to its specification. Does execution of twist applied to twist terminate? If it terminates, then halts applied to twist twist returns true, and so we see from the body of twist that execution of twist applied to twist does not terminate. If it does not terminate, then halts of twist twist returns false, and so we see from the body of twist that execution of twist applied to twist terminates. This is inconsistent. Therefore function halts cannot have been programmed according to its specification; halts is incomputable. Turing's argument [57] starts with the assumption that halts is computable. Then he finds an inconsistency. So he concludes that the computability assumption was wrong; halts is incomputable. But the computability assumption was completely unnecessary to the argument. [58] Let's leave it out, and then we can't conclude that it was wrong. Whether halts is computable or not, [59] what does its specification say halts of twist twist should be? If it's true it should be false. If it's false it should be true. Same inconsistency without the computability assumption. The only possible conclusion is that the specification of halts is inconsistent. The conclusion is NOT that halts is perfectly well defined but incomputable. It's the same inconsistency we've had all along. Turing could have made his argument [60] using the simplest version, without the parameters and with just a single procedure. Assume that the expression (execution of twist terminates) is computable, and that it has been programmed according to its specification. Does execution of twist terminate? If it terminates, then (execution of twist terminates) is true, and so we see from the body of twist that its execution does not terminate. If it does not terminate, then (execution of twist terminates) is false, and so we see from the body of twist that its execution terminates. This is inconsistent. Therefore the expression (execution of twist terminates) cannot have been programmed according to its specification; it is incomputable. But there are only two possibilities for programming (execution of twist terminates); they are true and false. Calling this choice incomputable says that one of them is correct but we cannot determine which one. [61] In fact, neither of them is correct, and that is called an inconsistent specification.

The inconsistency we have been looking at has nothing to do with computability, and it has nothing to do with [62] halting, either. This procedure says that if its execution prints an A then it prints a B, and otherwise it prints an A. Either way, its execution halts, so halting is not an issue. Turing's argument concludes that the expression (execution of twist one prints A) is incomputable. The conclusion [63] ought to be that the specification (execution of twist one prints A) is inconsistent. I could have dressed up this example by making two definitions and adding a couple of parameters. Anyway, the same argument can be used to claim that anything is incomputable. [64] This procedure says: if its execution calumates, then execute a procedure that doesn't calumate. And if its execution doesn't calumate, then execute a procedure that does calumate. Turing's argument is equally good or bad here, and it concludes that calumation is incomputable. But calumation is meaningless. I made up the word. That seems suspicious to me.

[65] Ok, now I want to go back to the Liar's Paradox and [66] change it to the liar's dilemma by changing the word false to true. [67] We give the sentence the name U, and express it mathematically as this equation. If U [68] has value true, this is the equation, and true equals true [69] is true, and again true equals true [70] is true. So U equals true is a solution to the equation. If U [71] has value false, this is the equation, and false equals true [72] is false, and false equals false [73] is true. So U equals false is also a solution to the equation. Having no solution is called inconsistent, or overdetermined, and having more

than one solution is called [74] underdetermined.

[75] This was Gödel's paradox, which we looked at earlier, and we concluded that if the logic is complete, then it's inconsistent, or overdetermined. Now leave B the same, but in G, change [76] false to true and call it H. So [77] what happens when B is applied to H? Well, if it gives true, then H is a true sentence, so B of H should give true. And if it gives false, then H is a false sentence, so B of H should give false. So it could be either answer. [78] The specification of B is overdetermined for some sentences, like G, and underdetermined for some sentences, like H.

[79] This is the halting problem, or programmer's paradox. That's exactly the halts specification and twist procedure that we saw before. Now I'm leaving the halts specification the same, but in twist I'm putting a [80] not in front of halts, and renaming it straight. So what happens when [81] halts is applied to straight straight? If we suppose it's true, then execution of straight terminates, so it should be true. If we suppose it's false, then execution of straight does not terminate, so it should be false. So both ways are right. This shows that the specification of halts is [82] underdetermined. It sounds like a good specification, but in some cases it's overdetermined, and in other cases it's underdetermined.

[83] Now I want to talk about specifications a little more generally. We've been specifying computer behavior. The specification says what behavior we want, and the computer is supposed to behave as specified. [84] But now, I want to include people, or other things, that could be the agent performing the specified behavior. [85] And I mean to be quite liberal about the language of specifications. It can be first order logic, or English, or anything else. [86] It can be a question, which specifies a response, or a command, or a description of behavior. And I want to divide specifications [87] into two kinds. An objective specification is one where the specified behavior does not vary depending on the agent that performs it, and a subjective specification is one where the specified behavior does vary depending on the agent that performs it. As an [88] example of an objective specification, given a natural number, what is its square. [89] What is the number of words in this question? [90] What is the name of the first Turing Award winner? [91] I'm not talking about people who don't know or forget or make mistakes or tell lies. In each case, there is a correct answer, and it doesn't depend on who you ask. The simplest [92] subjective specification is What is your name? [93] If we ask Alice, the correct answer is Alice, and if we ask Bob, the correct answer is Bob. The correct answer changes, depending on who you ask. [94] The computer equivalent would be What is your IP address? [95] Here's a more interesting example. Can Alice correctly answer no to this question? Well, let's [96] ask Alice. If she says no, she's saying that she cannot correctly answer no. She's saying that no is incorrect [97], contradicting her own answer. If she says [98] yes, she's saying that no is the correct answer, so her answer is [99] wrong. But if we ask [100] Bob, he can say no, and that's correct, because Alice cannot correctly answer no. This question is subjective because one agent correctly answers no, and another agent has no correct answer. For Alice, it's a self-contradictory, or inconsistent, or unsatisfiable specification. For Bob, it's a consistent answerable question. [101] How about this one? Can any woman correctly answer no to this question? Alice is a woman, so for her, it's another unanswerable question, another inconsistent specification. But for Bob, who isn't a woman, it's a consistent answerable question. How about [102] this one? Can anyone correctly answer no to this question? That's objective, because no matter who you ask, the result is the same. It's unanswerable. [103] Some of these questions were self-referential. The question refers to itself. And [104] some of the questions have a twist: the word no. If we replace the word no with the word yes, they're all objectively consistent.

[105] Now let's get back to computing. In the 1930s, people proposed a variety of computing formalisms, including these three. The Church-Turing thesis says: whatever can be computed, can be computed by any one of these three things. They all have the same computing power. They each, differently but equivalently, define what it means to compute.

[106] A modern statement of the thesis is that all programming languages have the same computing power, and they each, differently but equivalently, define what it means to compute. So [107] here's a question: can any Pascal program correctly answer no to this question? Pascal was a popular programming language in the 1970s and 80s. [108] We can certainly write a Pascal program to print no. As an answer to the question, it says that it's [109] incorrect. We can also write a [110] Pascal program to print yes, but that says that no is the correct answer, so [111] that's wrong too. But if we write a [112] Python program to print no in answer to the question, it's correct. It doesn't matter whether it's people or programs. A program in one language can answer the question, but a program in another language can't. [113] The Church-Turing thesis may be true for all objective specifications, but it isn't true for subjective specifications.

[114] Here's another version of the Church-Turing thesis. It says you can translate any program from one programming language to another. [115] And yet another version of the Church-Turing thesis says you can write an interpreter in any language for programs in any language.

I want to look at program translation now. [116] Is text p a Pascal program, where p is an input parameter? [117] We can write a Python program to answer that question. Maybe it's a Pascal compiler written in Python. Any compiler checks its input to see if it is a program in the language it's compiling, and prints error messages if it isn't, and generates object code if it is. [118] So we can do that. And we can translate it to [119] Pascal, or just write a Pascal compiler in Pascal. And for the same input, it will give the same answer. [120]
It's an [121] objective specification. To make it subjective, we make it refer to the agent who answers. [122] Is the program answering this question a Pascal program? To write a [123] Python program to answer the question, there are two ways. [124] The easy way is to just print no, because that's the correct answer. The [125] hard way is to give the program access to its own text, so it does the analysis just like a compiler does, and discovers the answer is no. Now we translate the program to Pascal. If we [126] programmed the easy way, the translated Pascal program also prints no, but now that answer is wrong. If we programmed [127] the hard way, the translated program accesses its own text, and goes through the same sort of analysis, and comes up with yes, which is the right answer. This time we have a [128] subjective specification. Either the translation behaves the same way, prints the same thing, but now it's wrong, or the translation behaves differently, prints a different answer, which is correct.

[129] Here's what's going on. Program p is applied to input x, but I'm also writing the programming language p is written in explicitly. For objective specifications, changing the language doesn't change the result. That's what the Church-Turing thesis is talking about. But for [130] subjective specifications, changing the language can change the result. Or the result can stay the same but it no longer satisfies the specification.

[131] Now let's get back to the halting problem. This is what we had. The specification of halts as a Pascal function to compute halting for all Pascal procedures is inconsistent. There is no such Pascal function. But could there be a [132] Python function to compute halting for all Pascal procedures? A Pascal procedure cannot call a Python function, so there's no twist procedure, no twisted self-reference. -- The usual argument against this is that if you could write a Python program to compute halting for all Pascal programs, then you could just translate it into Pascal. But we know there's no Pascal program to compute halting for all Pascal programs. So there can't be a Python program to do that either. -- This argument is wrong because the specification is subjective. Maybe there *can* be a Python program to compute halting for all Pascal programs. It gives correct answers. If you [133] translate it into Pascal, the translation *cannot* satisfy the specification. It sometimes gives wrong answers. Maybe the Python halts says twist terminates. Its translation into Pascal says twist doesn't terminate. And that's why twist terminates. So the

Python program is right but its Pascal translation is wrong.

[134] There's a town called Pascalville, with some people in it. Those squares are the people. And nearby there's another town called Pythonville with some other people in it. A person might admire some people, and not admire other people. A person might admire themself [135], or not. A person who admires themself is conceited, and they get Xs. A person who doesn't admire themself is modest, and they get check marks. A [136] character judge for a town is someone who admires the modest people in that town, and doesn't admire the conceited people. In Pythonville, there's a person named [137] Monty, and Monty is a character judge for Pascalville. He admires all those people in Pascalville who have check marks, and he doesn't admire the people in Pascalville who have Xs. There's no problem here. The specification of judge for Pascalville is perfectly consistent for Monty, and he can satisfy the specification. The problem arises when Blaise [138], who lives in Pascalville, decides to be a character judge for Pascalville. He says that if Monty can do it, so can he. He just has to do exactly what Monty does. If Monty admires someone, so will he. If Monty doesn't admire someone, neither will he. In other words, Blaise will translate Monty into Pascalville. But Blaise is wrong. Blaise cannot be a character judge for Pascalville because Blaise would have to admire himself if and only if he does not admire himself. That's how there could be a Python program to compute halting for Pascal programs, but no Pascal program to do the same thing.

[139] I've been talking about different programming languages, but it's not really a language problem. Here are two identical Turing Machines. One is in location A, and the other is in location B. They both execute Turing Machine language programs. Same language on both machines. On machine A, there cannot be a program to compute halting for all programs on machine A. But on machine B, maybe there *can* be a program to compute halting for all programs on machine A. That's because Turing Machines do not communicate with each other, so a program on A cannot call a program on B, even though both machines use the same language. Well, you might say, just carry the program that works from machine B to machine A. There's no language translation, but there is location translation, and that has the same effect. On machine B, the program gives correct answers. But the identical program running on identical machine A gives wrong answers. Does that sound absurd?

[140] Here's a simple example of location dependence. Is this sentence written on page 1? When it's on page 1, the answer is yes. But when the exact same question, written in the same language, is in a different location, the same answer is wrong. The halting problem is a little more interesting because the question here is not a twisted self reference, and the halting problem is. [141] So look at this question. Can a person in location A correctly answer no to this question. When Alice is in location B, she can correctly answer no. But when she [142] walks over to location A, she cannot answer correctly at all. Both answers are wrong. That's why a Turing Machine program on machine B might be able to compute halting for all Turing Machine programs on machine A, even though a program on machine A can't.

[143] In conclusion, the liar's paradox, Gödel's incompleteness theorem, and the halting problem are all twisted self-references, and they are all [144] inconsistent specifications. [145] Specifications can be objective or subjective. [146] The Church-Turing thesis applies to objective specifications, but not to subjective ones. [147] For objective specifications, translation between languages can preserve both the specification and the behavior, but for subjective specifications, translations may not preserve the specification, and they may not preserve the behavior. [148] For the halting problem, [149] it is inconsistent to ask for an L-program to compute halting for all L-programs, where L could be the programming language, or the computer where the program resides. [150] The proof of the inconsistency is a twisted self-reference, which is indicative of a subjective specification. [151] It *is* consistent to ask for an L program to compute halting for all M

programs because [152] that gets rid of the self-reference.

I'm done, and I hope you found my video interesting. If you want to see Turing's proof of the halting problem, [153] here it is. It's not easy to read, so I've put some more modern words inside square brackets to help. And when Turing says machine, he means program. I'll just point out [154] the computability assumption. That's what I wrote as saying assume that halts is computable, and that it has been programmed according to its specification. Then there's [155] the twist procedure. Then there's the [156] question what is the result of applying halts to twist. And [157] finally, the discovery of a self-contradiction, and the conclusion that halting is incomputable. The proof doesn't seem to mention any programming language, but implicitly it does. [158] It talks about the standard description of a computing machine, which is a number that encodes a program. And Turing Machine programs can be numbered because they are in a language, the Turing Machine language, that has syntactic rules that enable us to enumerate programs. And then a diagonal program D is assumed to be in that same enumeration, so it's in the same language, and the halting program H is constructed from D, so it's also in the same language. [159] The proof fails to recognize the language dependence. It also fails to recognize location dependence by assuming there's only one computer. Turing's proof proves that there cannot be a program in the Turing Machine programming language, running on a Turing Machine, that determines halting for all programs in that same language running on that same machine. But it doesn't rule out a halting program in a different language, or on a different machine, to compute halting for all Turing Machine programs.

[160] There's a proof that halting is incomputable by Bob Boyer and J Moore that's noteworthy because they claim it is completely formalized and verified using their automated prover ACL. That's not quite true because ACL is a constructive theorem prover, and the theorem they are proving is a nonconstructive theorem. [161] But they argue that their proof is valid anyway, and I find their arguments to be completely convincing. [162] They use LISP as their programming language, and they encode programs as text, rather than as numbers. [163] They define function CIRC like this. If execution of the program encoded as argument A, given input A, terminates, then CIRC's execution is an infinite loop, and otherwise it terminates. It's exactly the same as the twist procedure, but in LISP rather than Pascal. Their proof proves that there is no LISP function to compute halting for all LISP functions. But it doesn't rule out a halting program in a different language, or on a different machine, to compute halting for all LISP programs.

Sometimes when I talk about halting to a live audience, at the end someone comes up to me and says: that's not the true version, or the correct version, of halting, and that's why you get your funny results. Then they give me what they say is the true and correct version. I look at it later, and find out that it's equivalent to the version I used. If it's a nicer presentation, then maybe I use it next time I talk, and someone else says <u>that's</u> not the true and correct version. Then they give me another version. So now I have a whole collection of them, and I put them on my website. The differences among them are superficial; at their core, they are all twisted self-references.

If you have the time and interest, I'll show you just one more. [164] This one is used in the computability course at my university. [165] We start by defining the mathematical halting function like this. C is a finite character set, and C star is the set of all finite sequences of characters in C. So H takes two texts and returns either yes or no. Some of these texts are programs, and if p is a program whose execution on input text i terminates, then H of p i is yes. Otherwise it's no. Now we wonder: could there be a program, which we'll call twist, with one text input, that behaves like [166] this? For all texts p, if H of p p equals no, then execution of twist on input p terminates, and if H of p p equals yes, then execution of twist on input p does not terminate. Well, if there is such a program, then [167] what is H of twist twist? If it's no, then execution of twist on input twist terminates, so according to the definition of H, H of twist twist equals yes. And if H of twist twist equals yes, then execution of twist on twist does not terminate, so according to the definition of H, H of twist twist equals no. That's inconsistent,

[168] so there cannot be such a program as twist.

[169] Now suppose that there's another program, let's call it halts, that computes the mathematical halting function H. Then we could program twist like this: [170] execute halts of p p, but don't output. If the output from executing halts of p p would be “no”, terminate execution. If the output from executing halts of p p would be “yes”, loop forever. But we already know that there is no such program as twist, [171] so there can't be a halts program. That's the proof, but I have three criticisms.

My first criticism is that this proof doesn't distinguish between a program and a text encoding of the program. Gödel and Turing both understood the importance of that distinction, although they used numeric encodings because the text, or character string, data type had not yet been invented when they did their work. To see the difference, [172] look at the arithmetic expression one plus two and the text one plus two. The arithmetic expression one plus two [173] is equal to three, but the text one plus two is not equal to the text three.

Fortunately, in this proof, it's easily fixed. We just need to add a few words. Sometimes the word [174] program has to be changed to [175] a text encoding or representing a program. And a few more changes like that. So I'm not bothered with this problem.

My second criticism of the proof is its failure to recognize the dependence on a programming language. H is defined using the phrase [176] if p is a text representing a program, so we need to know if a text represents a program, so we need to know the rules of program formation; in other words, we need to know the programming language. And since we [177] apply H to twist, we are assuming that twist is in that same language. When we program twist, it [178] calls halts, so we are assuming halts is callable from twist. [179] The conclusion should have been that H cannot be computed by a program in the language over which H is defined. This conclusion leaves open the possibility that H can be computed in a language that differs from the language over which H is defined.

The last criticism of the proof is that it's unnecessarily complicated. H and twist and halts do not need input parameters; they could be defined for only the one input they are applied to in the proof. But maybe the complication is needed for the proof to seem respectable.

No matter what version of the halting problem you think is the true and correct version, it always has the same limitation: it assumes that halts and twist are in the same language, or at least that twist can call halts. But if they are in different languages, or if twist cannot call halts, then maybe we can compute halting.