Epimenides, Gödel, Turing: an Eternal Gölden Twist [0]

Eric C.R. Hehner

Department of Computer Science, University of Toronto
hehner@cs.utoronto.ca

Abstract: The Halting Problem is a version of the Liar's Paradox.

Epimenides

An ancient Cretan named Epimenides is reported to have said "All Cretans are liars." [1]. This is supposed to be self-contradictory, but it misses the mark. If there is any other Cretan, and that Cretan is a truth-teller, then Epimenides' sentence is simply false: Epimenides is a liar, but not all Cretans are liars. Saint Paul missed the point completely, taking Epimenides' statement at face value, and elaborating: "It was one of themselves, one of their own prophets, who said, "Cretans were never anything but liars, dangerous animals, and lazy": and that is a true statement." [5]. I will refer to the simpler sentence

This sentence is false.

as the Liar's Paradox. If that sentence is true, then, according to the sentence, it is false. If it is false, then it is true. That simple sentence is self-contradictory.

I give the sentence a name, say L for Liar.

L: L is false.

As a mathematical formula, it becomes

$L = (L=false)$

As an equation in unknown L, it has no solution, because the equation is $false$ regardless of whether L is $true$ or $false$. As a definition or specification of L it is called "inconsistent". (I am using italic $true$ and $false$ for the binary values representing truth and falsity.)

A slightly more complicated version presents the inconsistency as two sentences.

The next sentence is true.

The previous sentence is false.

Naming the first sentence B and the second G, as mathematical formulas, they become

$B = (G=true)$

$G = (B=false)$

These two equations in the two unknowns B and G have no solution: there is no assignment of binary values to B and G that satisfies the two equations. They are inconsistent. If you look at either one of the sentences alone, there is no inconsistency. It may make sense to say that the next sentence is true, and it may make sense to say that the previous sentence is false. But together they are inconsistent.

Let me complicate this inconsistency by adding a parameter, so B can say whether any sentence is true, not just sentence G. To reduce contention over truth and falsity, I will stick with mathematical sentences, otherwise known as binary expressions (allowing subexpressions of any type, including functions). To pass sentences as data, we need to encode them in some way. The easiest encoding is as a character string. Now B becomes a function from strings to binary values, and the pair of sentences become

$B(s)$ = $true$ if string s represents a binary expression with value $true$;
$false$ otherwise

G = " $B(G) = false$ "

I have made two definitions: B and G. Since G is just a character string, there cannot be anything wrong with its definition; it represents the binary expression $B(G) = false$. But the definition of B, no matter how carefully worded, no matter how clear it sounds, conceals an inconsistency. I am not concerned with computing B; I just want to define a mathematical function. The parameter allows us to show a large number of examples, like $B(\text{“0=0”}) = true$ and $B(\text{“0=1”}) = false$, which are not problematic. They may fool us into believing that the definition of B makes sense. But they are irrelevant. The inconsistency is revealed by applying B to G. If $B(G) = true$, then G represents a $false$ expression, so $B(G)$ should be $false$. If $B(G) = false$, then G represents a $true$ expression, so $B(G)$ should be $true$. The inconsistency is the same as in the unparameterized, unencoded version of the Liar's Paradox.

Gödel

The Liar's Paradox is about truth. Gödel used the same self-contradictory construction to talk about provability [2]. He used a numeric, rather than string, encoding of sentences, and he used the name Bew (short for Beweisbar, which is German for provable) for a function similar to B. The sentence encoded by G is popularly called “the Gödel sentence”. With our notations and encoding, B and G become

$B(s)$ = $true$ if string s represents a provable binary expression;
$false$ otherwise
G = “$B(G) = false$”

What is the value of $B(G)$? If we suppose $B(G)$ is $true$, then G represents a $false$ sentence, and in a consistent logic, no $false$ sentence is provable, so $B(G)$ should be $false$. If we suppose $B(G)$ is $false$, then G represents a $true$ sentence, and in a complete logic, all $true$ sentences are provable, so $B(G)$ should be $true$. Gödel concluded that if a logic is expressive enough to define B, then the logic is either inconsistent or incomplete.

Most of Gödel's paper [2] is devoted to showing how to define Bew. His definition was equivalent to programming a prover, using a functional language, namely the logic of *Principia Mathematica* [6] (hence the name of his paper). That was an amazing piece of work. But Gödel needn't have gone to so much trouble. Bew is defined to apply to all sentence encodings. But there is only one sentence encoding he wants to apply it to. For a single sentence, we don't need a sentence encoding. Define

B = $true$ if G is a provable binary expression;
$false$ otherwise
G = (B=$false$)

What is the value of B? If we suppose B is $true$, then G is a $false$ sentence, and in a consistent logic, no $false$ sentence is provable, so B should be $false$. If we suppose B is $false$, then G is a $true$ sentence, and in a complete logic, all $true$ sentences are provable, so B should be $true$. We can conclude from this simpler construction that if a logic is expressive enough to define B, then the logic is either inconsistent or incomplete.

Even this simpler construction includes one more definition than necessary. We could define

B = $true$ if B=$false$ is a provable binary expression;
$false$ otherwise

What is the value of B? If we suppose B is $true$, then B=$false$ is a $false$ sentence, and in a consistent logic, no $false$ sentence is provable, so B should be $false$. If we suppose B is $false$, then B=$false$ is a $true$ sentence, and in a complete logic, all $true$ sentences are

provable, so *B* should be *true*. If a logic is expressive enough to define *B*, then the logic is either inconsistent or incomplete.

Turing

Epimenides talked about truth; Gödel talked about provability; Turing talked about computability using the same sort of arguments [4]. For my examples, I will use the Pascal programming language, but the choice of language is irrelevant; any other programming language would do just as well. I'll start with a procedure named *twist* that is closely analogous to the Liar's Paradox.

```
procedure twist;
begin
    if (execution of twist terminates) then twist
end
```

I have not finished writing procedure *twist*; what remains is to replace the informal binary expression (execution of *twist* terminates) with either *true* or *false*, whichever one is appropriate. The problem in doing so is that the informal binary expression refers to itself in a self-contradictory manner: if the execution of procedure *twist* terminates, it should be replaced with *true*, creating a procedure whose execution does not terminate; if the execution of *twist* does not terminate, it should be replaced with *false*, creating a procedure whose execution does terminate. This is not a programming problem, not a computability problem, not a lack of expressiveness of Pascal. The problem is that the informal binary expression is an inconsistent specification. One might protest:

> 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 the inconsistency is clear enough that no-one will protest.

As we did with the Liar's Paradox, let's present the same inconsistency as two declarations.

```
const halts = { true if execution of twist terminates, false otherwise };

procedure twist;
begin
    if halts then twist
end
```

The value of constant *halts* is missing. In place of the value there is a comment to specify what the value should be. If execution of procedure *twist* terminates, then the value should be *true*. If execution of procedure *twist* does not terminate, then the value should be *false*. So there is no problem in programming the value of *halts* after we determine whether the execution of *twist* terminates. If we suppose it does, then *halts=true*, and so we see that execution of *twist* does not terminate. If we suppose it does not, then *halts=false*, and so we see that execution of *twist* does terminate.

Procedure *twist* has been written in its entirety. Syntactically, it is a procedure; to determine that *halts* is being used correctly within *twist*, we need only the type of *halts*, not the value, and we have the type. Semantically, it is a procedure; to determine the meaning of *twist* we need only the specification of *halts*, not its implementation, and we have the specification. (That important programming principle enables a programmer to use pieces of programs written by other people, knowing only their specifications, not their implementations. It also enables a programmer to change the implementation of part of a program, but still satisfying the specification, without knowing where and why the part is being used.) So there is nothing

wrong with the definition of *twist*. The problem is that we cannot write the value of *halts* to satisfy its specification. This is not a programming problem, not a computability problem, not a lack of expressiveness of Pascal. The problem is that the specification of *halts* is inconsistent. One might protest:

> 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?

The inconsistency cannot be seen by looking only at *halts* or only at *twist*. Each refers to the other, and together they are inconsistent.

Let me complicate this inconsistency by adding a parameter, so *halts* can say whether execution of any parameterless Pascal procedure terminates, not just *twist*. To pass procedures as data, we need to encode them in some way, and the easiest encoding is as a character string. (Whenever programs are presented as input data to a compiler or interpreter, they are presented as character strings.) We could pass the whole procedure as text, but it is simpler to pass just the procedure name as text, and to assume there is a dictionary of function and procedure definitions that is accessible to *halts*, so that the call *halts* ('*twist*') allows *halts* to look up '*twist*' and '*halts*' in the dictionary, and retrieve their texts for analysis.

function *halts* (*p*: **string**): **boolean**;
{ return *true* if *p* represents a parameterless Pascal procedure whose execution terminates; }
{ return *false* otherwise }

procedure *twist*;
begin
    if *halts* ('*twist*') **then** *twist*
end

To determine that *twist* is syntactically a Pascal procedure, we need only the header for *halts*, not the body, and we have the header. To determine the semantics of *twist*, we need only the specification of *halts*, not its implementation, and we have the specification.

As before, we cannot write the body of *halts* to satisfy the specification. No matter how carefully worded it is, no matter how clear it sounds, the specification conceals an inconsistency. The inconsistency is revealed by applying *halts* to '*twist*'. If *halts* ('*twist*') = *true*, then execution of *twist* is nonterminating, so *halts* ('*twist*') should be *false*. If *halts* ('*twist*') = *false*, then execution of *twist* is terminating, so *halts* ('*twist*') should be *true*. This is still not a programming problem, not a computability problem, not a lack of expressiveness of Pascal. It is still the same inconsistency that was present in the unparameterized, unencoded version, and the same inconsistency that was present in the *twist* procedure. One might protest:

> Either execution of a procedure represented by *p* terminates, or it doesn't. If it terminates, *halts* (*p*) should return *true*; if it doesn't, *halts* (*p*) should return *false*. How can there possibly be an inconsistency?

Now the protest starts to sound more plausible because the parameter allows us to show a large number of examples which are not problematic. For example, if we define *stop* and *go* as

> **procedure** *stop*; **begin end**
> **procedure** *go*; **begin** *go* **end**

then

> *halts* ('*stop*') = *true*
> *halts* ('*go*') = *false*

These nonproblematic examples may fool us into believing that the specification of *halts* makes sense. But they are irrelevant. Procedure *twist* shows us the inconsistency.

There is one last complication: a second parameter so *halts* can say whether execution of any

Pascal procedure with an input parameter terminates.

```
function halts (p, i: string): boolean;
{ return true if p represents a Pascal procedure with one string input parameter }
{ whose execution terminates when given input i ; return false otherwise }

procedure twist (s: string);
begin
    if halts (s, s) then twist (s)
end
```

This is now a modern version of Turing's Halting Problem. Turing's argument is as follows.

> Assume that *halts* is computable, and that it has been programmed according to its specification. Does execution of *twist* ('*twist*') terminate? If it terminates, then *halts* ('*twist*', '*twist*') returns *true* , and so we see from the body of *twist* that execution of *twist* ('*twist*') does not terminate. If it does not terminate, then *halts* ('*twist*', '*twist*') returns *false* , and so we see from the body of *twist* that execution of *twist* ('*twist*') terminates. This is inconsistent. Therefore function *halts* cannot have been programmed according to its specification; *halts* is incomputable.

The two parameters (*p*, *i*) make a two-dimensional space, and point ('*twist*', '*twist*') is on its diagonal, which is why the argument is sometimes called a "diagonal argument". But any string would do equally well as a value for the second parameter, and the second parameter adds nothing to Turing's argument.

The surprise, and the main point of this paper, is that the computability assumption is unnecessary to the argument. Without assuming that *halts* is computable, I ask what the specification of *halts* says the result of *halts* ('*twist*', '*twist*') should be. If the specification says the result should be *true* , then the semantics of *twist* ('*twist*') is nontermination, so *halts* ('*twist*', '*twist*') should be *false* . If the specification says the result should be *false* , then the semantics of *twist* ('*twist*') is termination, so *halts* ('*twist*', '*twist*') should be *true* . This is inconsistent. Therefore *halts* cannot be programmed according to its specification. But the problem is not incomputability; it is inconsistency of specification. It is the same inconsistency that was present in all previous versions, before I added the complications of parameters and encodings. It is just the Liar's Paradox in fancy clothing. In fact, Turing's argument could have been applied to the simplest version of *twist* with equal (in)validity.

```
procedure twist;
begin
    if (execution of twist terminates) then twist
end
```

> 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. In fact, neither of them is correct, and that is called inconsistent.

Turing's argument can be applied to any property of program execution. For example,

procedure *twist1*;
begin
    if (execution of *twist1* prints 'A') **then** *print* ('B') **else** *print* ('A')
end

Termination of execution of *twist1* is not in question: when (execution of *twist1* prints 'A') is replaced with either *true* or *false* , whichever is appropriate, execution of *twist1* terminates. The question is whether 'A' or 'B' is printed. Turing's argument says that the property "prints 'A' " is incomputable, and so is every property of program execution (except for the trivial "always *true* " and "always *false* " properties) [3]. But the problem is not incomputability; the problem is inconsistency of specification.

Underdetermination

The Liar's Paradox, the Gödel sentence, and Halting Problem are all examples of inconsistency, which is also known as overdetermination. Here, "determination" means ruling out impossible candidates for solution: if we rule out all candidates, we have overdetermination; if we are left with more than one solution, we have underdetermination. An example is the sentence

    This sentence is true.

Whereas the Liar's Paradox can be neither true nor false, the sentence just written can be either true or false. Giving the sentence the name *U* for underdetermined, it becomes the formula

    U = (*U*=*true*)

As an equation in unknown *U* , it has two solutions: both *true* and *false* . As a specification of *U* , it is consistent, but falls short of determining *U* .

Here is an example of the underdetermination of Gödel's provability specification.

    B (*s*) = *true* if string *s* represents a provable binary expression;
                false otherwise
    H = " *B* (*H*) = *true* "

That is the same specification of *B* as before. Now we ask: What is the value of *B* (*H*) ? If we suppose *B* (*H*) = *true* , then *H* represents the sentence *true*=*true* , which is provable, so *B* (*H*) should be *true* , as supposed. If we suppose *B* (*H*) = *false* , then *H* represents the sentence *false*=*true* , which is not provable, so *B* (*H*) should be *false* , as supposed. The specification of *B* is both overdetermined (for *G*) and underdetermined (for *H*).

Here is an example of the underdetermination of Turing's halting specification.

function *halts* (*p*, *i*: **string**): **boolean**;
{ return *true* if *p* represents a Pascal procedure with one string input parameter }
{ whose execution terminates when given input *i* ; return *false* otherwise }

procedure *straight* (*s*: **string**);
begin
    if not *halts* (*s*, *s*) **then** *straight* (*s*)
end

That is the same *halts* specification as before; it says that the *halts* function will tell us whether the execution of a procedure terminates. What does it say about *straight* ? If we suppose that *halts* ('*straight*', '*straight*') = *true* , we see from the body of *straight* that its execution terminates, so that was the right supposition. If we suppose that *halts* ('*straight*', '*straight*') = *false* , we see from the body of *straight* that its execution does not terminate, so again that was the right supposition. That is another inadequacy of the *halts* specification. The specification sounds just right: neither overdetermined nor underdetermined. But we are forced by the examples to admit that the specification is not as it sounds. In at least one instance (*twist*), the *halts* specification is overdetermined, and in at least one instance (*straight*), the *halts* specification is underdetermined.

Conclusion

Epimenides' construction shows us:

> Asking for a function whose result is *true* for all and only those strings representing true sentences in a sufficiently expressive language is both overdetermined (inconsistent) and underdetermined.

Gödel's construction shows us:

> Asking for a function whose result is *true* for all and only those strings representing provable sentences in a sufficiently expressive language is both overdetermined (inconsistent) and underdetermined.

Turing's construction shows us:

> Asking for a function, written in a programming language, whose result is *true* for all and only those strings representing procedures, written in that same language, whose execution terminates, is both overdetermined (inconsistent) and underdetermined.

If “incomputable” meant having an inconsistent specification, then *halts* would be incomputable. But “incomputable” does not mean “inconsistent”. It means that a well-defined mathematical function, one with a consistent specification, cannot be computed. That has not been proven.

Conjecture

I conjecture that every consistent first-order specification is satisfied by a computable function. This conjecture is a bit like the Löwenheim-Skolem theorem that every consistent first-order theory has a countable model.

References

[0] The title of this paper pays homage to the wonderful book by Douglas R. Hofstadter: *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books, 1979

[1] Epimenides Paradox, Wikipedia, http://en.wikipedia.org/wiki/Epimenides_paradox

[2] K.Gödel: über Formal Unentscheidbare Sätze de Principia Mathematica und Verwandter Systeme I, *Monatshefte für Mathematik und Physik* v.38 p.173-198, Leipzig, 1931

[3] H.G.Rice: Classes of Recursively Enumerable Sets and their Decision Problems, *Transactions of the American Mathematical Society* v.74 p.358-366, 1953

[4] A.M.Turing: on Computable Numbers with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society* s.2 v.42 p.230-265, 1936; correction s.2 v.43 p.544-546, 1937

[5] The Jerusalem Bible, Reader's Edition, Titus, chapter 1 verse 12

[6] A.N.Whitehead, B.Russell: *Principia Mathematica*, Cambridge University Press, 1910

other papers on halting