Prolog detects pathological self reference in the Gödel sentence

This sentence G ↔ ¬(F ⊢ G) and its negation G ↔ ~(F ⊢ ¬G) are shown to meet the conventional definition of incompleteness: Incomplete(T) ↔ ∃φ ((T ⊬ φ) ∧ (T ⊬ ¬φ)). They meet conventional definition of incompleteness because neither the sentence nor its negation is provable in F (or any other formal system).

The first incompleteness theorem states that in any consistent formal system *F* within which a certain amount of arithmetic can be carried out, there are statements of the language of *F* which can neither be proved nor disproved in *F*. Raatikainen, Panu, "Gödel's Incompleteness Theorems",*The Stanford Encyclopedia of Philosophy* (Fall 2018 Edition)

If Gödel's sentence is unprovable in Peano arithmetic (PA) only because PA is insufficiently expressive to prove this sentence and this sentence is a sentence of PA, then it would make perfect sense to say that PA is incomplete.

If Gödel's sentence is unprovable in PA and the fact that it is unprovable in PA is (for example) provable type theory then we could say type theory proves that his sentence is not provable in PA.

When we say that Gödel's sentence is true and unprovable we are actually saying that it is true that it is unprovable in PA. If it was totally unprovable in every formal system then it would simply be untrue.

There is no formal system that can prove a sentence of this formal system only expressing its own unprovability because such a sentence is erroneous:
"This sentence cannot be proven." is semantically vacuous. "This sentence cannot be proven to be a box of chocolates." can be proven to be true.

The conventional definition of incompleteness:
Incomplete(T) ↔ ∃φ ((T ⊬ φ) ∧ (T ⊬ ¬φ))

When we see that the following Prolog expressions satisfy the above definition of incompleteness then we can see that they are equivalent to the Gödel sentence in the 1931 incompleteness proof.

```
?- G = not(provable(F, G)).          % G = ¬(F ⊢ G)
?- G = not(provable(F, not(G))).     % G = ¬(F ⊢ ¬G)
```

When we test the above pair of expressions we find that neither of them are are provable in the Prolog formal system: **(SWI-Prolog (threaded, 64 bits, version 7.6.4)**

```
?- unify_with_occurs_check(G, not(provable(F, G))).
false.

?- unify_with_occurs_check(G, not(provable(F, not(G)))).
false.
```

Thus fulfilling the conventional definition of incompleteness, and proving equivalence to the 1931 Gödel "Incompleteness" sentence. The 1931 Gödel Incompleteness theorem correctly concludes that neither G nor ¬G are provable in F. The key detail that it leaves out is that **neither G nor ¬G are provable in F because both are erroneous cyclic terms that cannot be resolved in any formal system what-so-ever.**

BEGIN:(Gödel 1931:39-41)
...there is also a close relationship with the "liar" antinomy,14 ...

We are therefore confronted with a proposition which asserts its own unprovability.15

14 Every epistemological antinomy can likewise be used for a similar undecidability proof.

15 In spite of appearances, there is nothing circular about such a proposition, since it begins by asserting the unprovability of a wholly determinate formula (namely the q-th in the alphabetical arrangement with a definite substitution), and only subsequently (and in some way by accident) does it emerge that this formula is precisely that by which the proposition was itself expressed.
END:(Gödel 1931:39-41)

Gödel's footnote 15 is dodgy in that although it denies the circularity of his proposition he affirms its circularity in the same paragraph that he denies it:

Removing the dodgy words from the above.

> a proposition...begins by asserting the unprovability of a wholly determinate formula...this formula is precisely that by which the proposition was itself expressed.

Paraphrasing the above using less clumsy words:

> a proposition asserts the unprovability of a formula that expresses this same proposition:

AKA the same pathological self-reference as this Prolog expression:
G = not(provable(F, G)).

Prolog detects and reports pathological self-reference:
?- unify_with_occurs_check(G, not(provable(F, G))).
false.

Since Gödel says that: "Every epistemological antinomy can likewise be used for a similar undecidability proof." (including the liar antimony AKA liar paradox) then we can analyze how Prolog handles the Liar Paradox:

```
?- LP = not(true(LP)).
LP = not(true(LP)).

?- unify_with_occurs_check(LP, not(true(LP))).
false.
```

Because the Prolog Liar Paradox has an **"uninstantiated subterm of itself"** we can know that unification will fail because it specifies **"some kind of infinite structure."** The quotes come from: (Clocksin and Mellish 2003:255) (see below). If we simply take Gödel at his word: **"Every epistemological antinomy can likewise be used for a similar undecidability proof."** then the Liar Paradox is equivalent to his own undecidable sentence.

Godel, Kurt 1931. On Formally Undecidable Propositions of Principia Mathematica And Related Systems I, page 39-41.

BEGIN:(Clocksin & Mellish 2003:254)

Finally, a note about how Prolog matching sometimes differs from the unification used in Resolution. Most Prolog systems will allow you to satisfy goals like:

```
equal(X, X).
?- equal(foo(Y), Y).
```

that is, they will allow you to match a term against an uninstantiated subterm of itself. In this example, foo(Y) is matched against Y, which appears within it. As a result, Y will stand for foo(Y), which is foo(foo(Y)) (because of what Y stands for), which is foo(foo(foo(Y))), and so on. So Y ends up standing for some kind of infinite structure.

Note that, whereas they may allow you to construct something like this, most Prolog systems will not be able to write it out at the end. According to the formal definition of Unification, this kind of "infinite term" should never come to exist. Thus Prolog systems that allow a term to match an uninstantiated subterm of itself do not act correctly as Resolution theorem provers. In order to make them do so, we would have to add a check that a variable cannot be instantiated to something containing itself. Such a check, an *occurs check*, would be straightforward to implement, but would slow down the execution of Prolog programs considerably. Since it would only affect very few programs, most implementors have simply left it out [1].

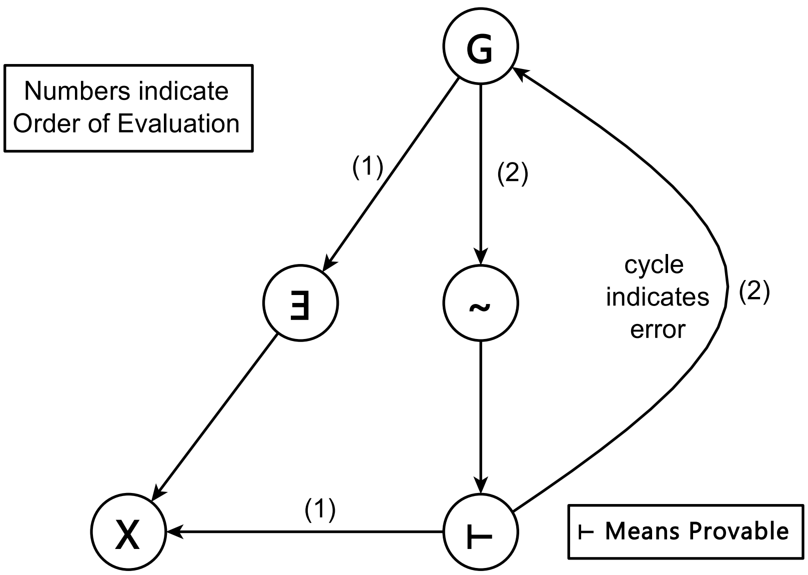
[1] The Prolog standard states that the result is *undefined* if a Prolog system attempts to match a term against an uninstantiated subterm of itself, which means that programs which cause tills to happen will not be portable. A portable program should ensure that wherever an occurs check might be applicable the built-in predicate unify_with_occurs_check/2 is used explicitly instead of the normal unification operation of the Prolog implementation. As its name suggests, this predicate acts like =/2 except that it fails if an occurs check detects an illegal attempt to instantiate a variable. **END:(Clocksin & Mellish 2003:254)**

Clocksin, W.F. and Mellish, C.S. 2003. Programming in Prolog Using the ISO Standard Fifth Edition, 254. Berlin Heidelberg: Springer-Verlag.

G := ∃X ~Provable(X, G)   // Written in Minimal Type Theory **
Automatically translated into a Directed Acyclic Graph by the MTT compiler

```
[01] G              (02)(04)
[02] THERE_EXISTS (03)
[03] X
[04] NOT            (05)
[05] Provable       (03)(01) // cycle indicates
                             // infinite evaluation loop
```

** x := y means x is defined to be another name for y



The first incompleteness theorem states that in any consistent formal system *F* within which a certain amount of arithmetic can be carried out, there are statements of the language of *F* which can neither be proved nor disproved in *F*. Raatikainen, Panu, "Gödel's Incompleteness Theorems",*The Stanford Encyclopedia of Philosophy* (Fall 2018 Edition)

If this expression is true then the Gödel sentence is false

~∃F ∈ Formal_Systems ~∃G ∈ WFF(F) (G ↔ (~(F ⊢ G) ∨ ~(F ⊢ ~G)))
There are no WFF G of any Formal_System F such that G is materially equivalent to its own unprovability or irrefutability in F.