A NOTE ON GÖDEL´S THEOREM

J. Ulisses Ferreira

Trv Pirapora 36 Costa Azul, 41770-220, Salvador, Brazil

ABSTRACT

This short and informal article shows that, although Godel's theorem is valid using classical logic, there exists some four-valued logical system that is able to prove that arithmetic is both sound and complete. This article also describes a four-valued Prolog in some informal, brief and intuitive manner.

KEYWORDS

Gödel, incompleteness theorem, four-valued logic, Hilbert's program

1. INTRODUCTION

In 1931, Kurt Gödel, after his revolutionary theorem, placed a full stop on Hilbert's dream of formalizing mathematics. Gödel demonstrated that it would not work even for arithmetic[1].

On the other hand, in the nineties, I started inserting a third value called "unknown" in Plain[2], i.e. a programming language that I was designing at that time. The *unknown* constant was theoretically referred to as *uu* since the end of nineties, when I was doing PhD abroad. In my own PhD thesis, I introduced a five-valued logic from the values in *{tt, ff, uu, ii, kk}*[3]. In 2004, I published not only that logic as a journal article but I also published a 7-valued logic in a Conference in San Diego[4], adding the values *{fi, it}* ("false or inconsistent", "inconsistent or true", respectively) for being able to be used together with the same uncertainty model proposed during my Master's course in 1990. My 7-valued logic that makes use of that uncertainty model permits that, during the computation, as the system obtains novel pieces of information, variables change their values. An example of this is the paternity test: before the discovery of the DNA test, it was possible to conclude whether a child was a daughter or son of a particular man by hereditary physical characteristics. However, there was always uncertainty up to some extent. The uncertainty factor could be represented by *uu* (unknown), at least as an initial state of some variable. Since the DNA test was discovered, all variables which represents the hypothesis of being the child's father should change their states from *uu* to either *kk* or *tt* or *ff*.

As part of my previous contribution, the *kk* value means "knowable", and it is usable when something is not already known, but it is already known that it is consistent. It can either be *true* or *false* but not both. It can be known by God or someone else or some machine, for instance, but it is not already known by the machine or person who is deductively reasoning, and it may be unknown forever, but at least its consistency is guaranteed. This is the meaning of the *kk* value, which fits in the referred uncertainty model when a variable thresholds collapse: *False* = *True*, which means that there is nothing strictly between the *False* and the *True* thresholds. In the above example of the paternity test, *uu* used to represent the initial state before the discovery of the DNA test, whereas *kk* represents the initial state given the existence of the DNA test, but before knowing the result of a particular DNA test, either *ff* or *tt*.

Individually and previously, Kleene, Lukasiewicz and Priest proposed their three-valued logics.

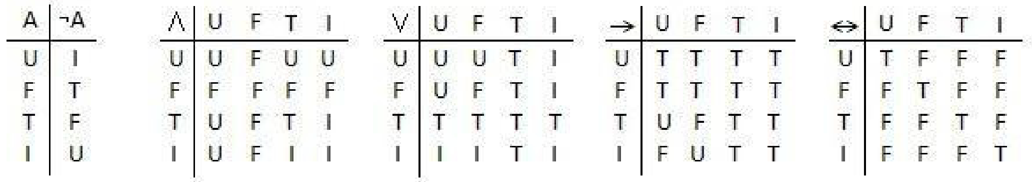
In 1977, Nuel Belnap (1930-) had proposed his logic on 4 values[5]. What I observed several years ago is that Gödel's proof may not work together with some logics that have more than 3 values. The 4 necessary values mean "true", "false", "unknown" and "inconsistent", or similar meanings. That is, at least these 4 values and meanings. The latter two values correspond to *N* (none) and *B* (both) in the four-valued Belnap's logic, respectively, and correspond to *uu* and *ii*, respectively, in both referred logics of mine, as well as in my four-valued logic presented in this article, and the four-valued Prolog also described here.

The problem Gödel introduced was due to existing self-references and paradoxes, which made propositions of arithmetic result in both *true* and *false*. Together with the observation that any proof over mathematics was also a mathematical object itself. However, Boolean logics are clearly unable to permit that formal systems written in them capture the problem pointed out by Gödel in his theorem.

One condition for a four-valued formal system being able to prove all true propositions, and only the true propositions, is certainly that it has the same results of the classical logic, except for when one or two operands have values other than *true* and *false*. In any proof, a result here is the true value only, and do not include values such as "unknown". Belnap's four-valued logic does not fail under this condition, for, although $(B \vee N = T)$, where *T* represents the true value, the values of the operands are "*B*" and "*N*", which are not Boolean.

In my PhD thesis, there was a kind of typo in the truth-table for the specific case *ff*↔*ff*, which results in *tt* in my logic but *ff* was written instead: a kind of mistake in only one of the two truth-tables for the equivalence operation. However, taking this into account, and by using only one of those equivalence operations, my five-valued logic suffices regarding that condition. Moreover, I checked in 2007, by using a program of mine, which seems to be correct, whether such a four-valued logic exists, and its computation resulted in several logics, where one of them was Belnap's logic. The 12th solution written by my program computation was the logic which pleased me the most, and I think that other people might have done the same as I did, and even preferred the same logic as I did, but I have never seen one. In 2011, I could not claim the authorship of that four-valued logic, but the true truth-tables of my preference are the following:

Table 1. The present author's four-valued logic true table



A	¬A
U	I
F	T
T	F
I	U

∧	U	F	T	I
U	U	F	U	U
F	F	F	F	F
T	U	F	T	I
I	U	F	I	I

∨	U	F	T	I
U	U	U	T	I
F	U	F	T	I
T	T	T	T	T
I	I	I	T	I

→	U	F	T	I
U	T	T	T	T
F	T	T	T	T
T	U	F	T	T
I	F	U	T	T

⇔	U	F	T	I
U	T	F	F	F
F	F	T	F	F
T	F	F	T	F
I	F	F	F	T

Section 2 dedicates to Gödel's theorem and his proof, and I informally describe a system for capturing all possible results. In section 3, I briefly describe a four-valued Prolog programming language, whereas section 4 contais my conclusions.

2. On Gödel's Proof

The set of all propositions on arithmetical true is written for the two Boolean values, but that set could be complete but cannot be sound, i.e. it is clearly inconsistent. However, I write an external layer with an external view of that set. My layer is written with 4 or more values. It interprets that set and, thus, both layers together form a formal system where the two-valued system is the server while the four-valued system is the client. The external layer makes use of the internal one.

The system with at least four values is pretty simple and works in the following manner:

Whenever one attempts to prove that a proposition is true and the two-valued system results in *true*, the external layer still tries to prove that the proposition is *false*: If the two-valued system results in *true*, the external layer results in *ii*, the inconsistent value. However, on the other hand, if the two-valued system results in *false* instead, the external layer results in *true*.

Whenever one attempts to prove that a proposition is *true* and the two-valued system results in *false*, the external layer still tries to prove that the proposition is *false*: If the two-valued system results in *false*, the external layer results in *uu*, the unknown value. However, on the other hand, if the two-valued system results in *true* instead, the external layer results in *false*.

In other words, the meaning of a two-valued internal query is only the attempt to prove, which either succeed or not. In this way, the whole formal system is clearly sound and complete. In 1997, I wrote a three-valued Prolog which I called Kleene at that time and Globallog in the following year[6] for becoming more modest, and the same language is the subject of one of the chapters of my PhD thesis.

The system described above in this section can be more clearly written in a Pascal-like language style as follows:

1. An algorithm in Pascal-like language

```
type
  proposition = string;

  QueryAnswers = (LocalFalse, LocalTrue, NotFound);
  FourValues = (uu, ff, tt, ii);

(* … *)

function TryProposition2v(p: proposition, q: QueryAnswers): boolean;
begin
   (*

     any polynomial search algorithm with unification for checking whether the
     proposition p is true. Alternatively, this function also returns

     the information that the search algorithm has been unable to answer
     whether the proposition p is true or false with respect to the current state
     of the knowledge base. In this case, where no unification has been found,
     the result is false.

   *)
end;

function proposition4v(p: proposition): FourValues;
begin

   if TryProposition2v(p,LocalTrue) then if
     TryProposition2v(p,LocalFalse) then
```

```
          proposition4v := ii
        else

          proposition4v := tt
      else

        if TryProposition2v(p,LocalFalse) then
          proposition4v := ff

        else
          proposition4v := uu
    end;
```

Clearly, such an algorithm captures all possibilities, and can be adapted to extend from the propositional logic to a more sophisticated and even second-order logic with propositions.

Certainly, we are unable to state all mathematical true, for mathematics is a science and, as such, new theorems and proofs, new mathematical objects, are being formulated all the time by researchers. So, a proper four-valued formal system is able to state when a proposition is still unknown due to the *uu* value. On the other hand, such a formal system captures the notion of paradoxes due to the *ii* value. Therefore, it is sound and complete.

3. A Four-Valued Prolog

For further work, a four-valued Prolog can be formally defined, implemented and used. In this section, I introduce a brief, informal and intuitive description of the adaptation of the three-valued Prolog defined by the present author[6]. Let us call Prolog4v the sample programming language whose interpreter is intended to be the four-valued formal system.

3.1 Syntactical and Semantic Definitions

Definition 1. A program in Prolog4v is a sequence S of clauses $c_1 \ldots c_n$. Thus, it is said that a computation by S proves a goal g if and only if there exists some c_i in S such that g is an immediate consequence of c_i , assuming that the body of c_i can be proven. The notion of clause and body are in the following definition subsection.

Given S as a sequence of clauses $c_1 \ldots c_n$, a program in Prolog4v corresponds to the disjunction among all clauses. That is: $c_1 \vee \ldots \vee c_n$, where the disjunctive operator $\vee$ is the same operator of the four-valued logic in table 1. Nonetheless, the interpreter, also called formal system here, carries out its computation “downwards”, i.e. from the first to the last clause. The sequence of clauses is often written like a Prolog program is, i.e. one clause occupies one line.

Definition 2. A clause is a language construct which has one of the forms bellow:

$$[\textbf{not}]\ p(t_1, , \ldots , t_n).$$

or

$$[\textbf{not}]\ p(t_1, , \ldots , t_n) \leftarrow [\textbf{not}]\ p_1(t_{1,1}, , \ldots , t_{r,1}), \ldots , [\textbf{not}]\ p_m(t_{1,m} , \ldots , t_{s,m}).$$

The first clause above corresponds to a *fact* whereas the second clause corresponds to a *rule*. All clauses end with a dot symbol. As usual in syntax definitions, the above brackets are not part of the language but, instead, they mean that the negation operator in optional in the clauses. Any rule contains its head, which is on the left of the inference operator ⟵, and its body, which is on the right of the same operator. At the lexical level of Prolog4v, there are two different inference operators to be chosen by the programmer, either ":-" like in Prolog or ":=". The former operator obeys the Closed World Assumption[7] and makes use of the Negation as Failure[8]. This means that if the body of a rule results in *uu*, the ":-" operator makes the head of the same rule become *ff*. Similarly, if the body of a rule results in *ii*, the ":-" operator also makes the head of the same rule become *ff*. The latter operator ":=" is a contribution of mine, which obeys what I called the Open World Assumption in my PhD thesis[6] and it corresponds to the → operator described in table 1.

Briefly, if no clause unifies some given goal *g*, the answer of the query for *g* is *uu*, the *unknown* constant of Prolog4v.

The body of any rule is formed by a sequence of predicates with zero or more indexed parameters *t*, separated by the comma symbol (","), which in its turn corresponds to the ∧ operator of the four-valued logic that I introduced in table 1, in the introductory section. During the computation, each predicate $p_{j,}(t_{1,j}\ , \ldots , t_{u,j})$ corresponds to a new four-valued goal and to a new four-valued query.

Definition 3. There exist four predefined constants in Prolog4v, namely, *ff*, *tt*, *uu* and *ii*.

The above constants correspond to the four operands F, T, U and I, respectively, of the four-valued logic described in table 1.

Note that, in accordance with table 1, if any of those queries in a body results in *ff*, the computation of the whole rule results in *ff* regardless of the existence of any possible inconsistency or lack of information in the other queries of the body of the rule in question. The queries are performed from left to right like in Prolog, but it is easy to see that Prolog4v interpreter can carry out the computation in parallel and it canb even distribute the computation among a number of machines. Also from table 1, note that, a rule results in *tt* if and only if all containing queries result in *tt*, that is, the trivial and Boolean cases clearly must hold.

With respect to the ":=" inference operator, which in its turn corresponds to the → implication operator of the introduced four-valued logic, but containing the sides of the implication swapped, I could have chosen any pairs of operands of the → table whose results are all *tt*. However, the main diagonal of the → table is what makes sense in the real world, hence they are my choices. That is to say, *ff* → *ff*, *tt* → *tt*, *uu* → *uu*, as well as *ii* → *ii* all result in *tt* and therefore → operator is not only sound but also makes sense in the real world. During the computation, if the body of a rule results in *ii* the query for the whole rule results in *ii* and, in this way, the inconsistency is propagated, possibly to the level of the user, such as a mathematician.

However, any query with the negation operator can be treated as a unity. That is, although the four-valued logic introduced in table 1 contains the "not" operator ¬, the system might not make use of it. Instead, the not operator can be part of the query as well as it is part of the unification algorithm, i.e. the system tries to unify the predicate including the "not" operator. Furthermore, not uu does not result in *ii*, whereas not ii does not result in *uu* either. Instead, the system ought to propagate *uu* and also *ii*. Thus, not uu results in uu whereas not ii results in ii. These are the only two exceptions with respect to table 1. In other words, there are two different forms of negation.

In contrast with the negation in table 1, let us call the **not** operator in the definition 2 “abstract negation”. I had also called “abstract negation” in the three-valued Prolog. Here, the negation is a four-valued extension.

Finally, the ↔ operator in the above four-valued logic is simply not used by the system.

3.2 Examples

Consider the following example of a two-clause program in Prolog4v:

```
happy(ann). not
happy(ann).
```

Over the last thirty years, some proposals have been made for solving the inconsistency problem[9], such as setting priorities, possibly in some implicitly way, for all clauses. The

literature on inconsistency in deductive databases and logic programs is large[10] but I think that there is little references on abstract negation.

In the above example, a query like happy(ann) clearly results in *ii*. Accordingly, a query like not hapy(ann) also results in *ii*. In both cases, the system tries to prove both and, in accordance with the algorithm 1, implicitly makes two binary queries, for both the positive and the negative forms of the predicate.

Now, consider the classical non-flying bird example:

```
fly(X) := bird(X), not penguin(X).
not fly(Y) := penguin(Y).
bird(tweety).

penguin(Z) :- bird(Z), polar(Z).
```

To answer the query fly(tweety), the system unifies the goal with the head of the first rule, binding the variable X to the constant tweety. Then, the system finds the subgoal bird(tweety) which in turn unifies the third clause and that subquerry results in *tt*. Then, in the body of the first rule, not penguin(tweety), is the next subgoal to be explored. Note that, because of the inference operator chosen, penguin is the head of a closed-world rule. Then, the subgoal unifies the fourth clause binding Z to tweety. As the subgoal bird(tweety) had already been proven, the next subgoal is polar(tweety). To explore this subgoal, the system does not unify any clause and, because of this, this subquery results in *uu*. The body of the fourth clause results in *uu* since T ∧ U results in U in table 1. The subquery penguin(tweety) results in *ff* because of the closed-world assumption made by using the “:-“ operator. If one replaces “:-“ by “:=” in the fourth clause, the subquery penguin(tweety) in *uu* instead.

Now, consider a new clause

```
polar(tweety).
```

is asserted and the system places it at the end of the sequence of clauses. For the same query fly(tweety), the system now answers *ff*. That is, it learns. In comparison to a similar Prolog program:

```
fly(X) :- bird(X), not penguin(X).
bird(tweety).
penguin(Z) :- bird(Z), polar(Z).
```

The same query fly(tweety) would have resulted in *true* because the third clause alone ensures that only a polar bird is a penguin. That is, until the knowledge base is complete, the system sometimes gives wrong answers with respect to the real world. For instance, for a query such as fly(airplane), results in *uu* in Prolog4v program above, whereas the same query results in *false* in the three-clause Prolog program above. Following this, binary formal systems are clearly not appropriate to write mathematical truths.

As another example, suppose that I know that Berne is the capital of Switzerland and that each county has one capital only. In Prolog4v, one would write

```
capital(berne,switzerland).

not capital(X,Y) := capital(Z,Y), X <> Z.
```

where <> stands for the different from (≠) operator. A non-ground query, i.e. a query where there is some unbound variable, for instance capital(bern,X) (note the different spellings) would result in X = *uu*, whereas a ground query such as capital(zurich,switzerland) would definitely result in *ff* as follows: the corresponding goal would not unify the first clause but would unify the second one because the presence of the not abstract negation in its head does not fail during the computation of the unification algorithm. In this case, X is bound to zurich and Y is bound to switzerland. Then, the system tries to prove the subgoal capital(Z,switzerland) and unifies the first clause, i.e. the fact capital(berne,switzerland) binding Z to berne. Now, the system evaluates the expression X <> Z, which in turn results in *tt* as zurich is not berne. Since all premises of the rule are true, the body results in *tt* and the system concludes that the head not capital(zurich,switzerlan) is *tt* and, hence, that capital(zurich,switzerland) is *ff*, and that is the response of the query at the user's level.

4. CONCLUSIONS

There exists some four-valued formal system that is able to state all arithmetical truth. I think that its descriptions is comprehensive even for undergraduate student. Philosophy students can also understand the present article.

The computation by the referred formal system roughly takes the double the time of a typical binary formal system: some time for trying to prove that a goal is true and some additional time for trying to prove that the same goal is false. Therefore, its computation is polynomial.

ACKNOWLEDGEMENTS

Many thanks to my Master's course supervisors, namely, Pedro Sérgio Nicolletti and Hélio Menezes Silva.

REFERENCES

[1] Gödel, Kurt, (1931) Über formal unentscheidbareSätze der Principia Mathematica und verwandterSysteme I. MonatsheftefürMathematikunPhysik, Vol. 38, pp173-198.
[2] Ferreira, Ulisses (2000) uu for programming languages. ACM SIGPLAN Notices, 35(8): pp20-30.
[3] Ferreira, Ulisses (2004) A five-valued logic and a system. Journal of Computer Science and Technology, 4(3): pp134-140, October.
[4] Ferreira, Ulisses (2004) Uncertainty and a 7-valued logic. In Pradip Peter Dey, Mohammad N. Amin, and Thomas M. Gatton, editors, Proceedings of The 2nd International Conference on Computer Science and its Applications, pp 170-173, National University, San Diego, CA, USA, June.
[5] Belnap Jr, Nuel, (1975) A useful four-valued logic. In J. Michael Dunn and George Epstein, editors, Proceedings of the Fifth International Symposium on Multiple-Valued Logic, Modern Uses of Multiple-Valued Logic, pp 8-37. Indiana University, D. Reidel Publishing Company.
[6] Ferreira, Ulisses (2004) A prolog-like language for the internet. In Veljko Milutinovic, editor, Proceedings of IPSI CAITA-04, Purdue, Indiana, USA.
[7] Reiter, R. (1978) On Closed World Data Bases in Logic and Data Bases, Plenum Press, New York, pp 55-76.
[8] Clark, Keith (1978) Negation as Failure in Logic and Data Bases, Plenum Press, New York, pp 293-322.
[9] Dung, P. M. & Mancarella P. (1996) Production Systems need Negation as Failure, Proceedings of the XIII National Conference on Artificial Intelligence, vol 2, AAAI Press and the MIT Press, pp 1242-1247.
[10] Seipel, Dietmar (1998) Partial Evidential Stable Models for Disjunctive Deductive Databases, in Logic Programming and Knowledge Representation, Third International Workshop / LPKR'97, Lecture Notes in Artificial Intelligence, vol. 1471, Springer, New York, October, pp 66-83

AUTHOR

The present author studied as a Master student at the Universidade Federal da Paraíba in Campina Grande, Brazil, and as a postgraduate student in the Department of Computer Science at the University of Edinburgh, and did some further research work at Trinity College in Dublin from 1998 until 2001.