DOI: http://dx.doi.org/10.26483/ijarcs.v10i5.6467
Volume 10, No. 5, September-October 2019
ISSN No. 0976-5697
International Journal of Advanced Research in Computer Science
RESEARCH PAPER
Available Online at www.ijarcs.info

ABOUT INSUFFICIENCY OF GRAMMARS FOR SEMANTIC VALIDITY OF COMPUTER PROGRAMS AND ONTOLOGIES AS AN ALTERNATIVE APPROACH

Laurence R. Ugalde
Project Creator, Formulae (formulae.org)
Toluca, Mexico

Abstract: The Rice theorem proves that semantic properties of computer programs are not decidable. In this paper it will be proved that grammars are not a sufficient mean to provide semantic validity on computer programs, first as a corollary of the Rice theorem and then as an independent theorem. Once that is proved that they are not sufficient, the use of ontologies are presented as a viable alternative to grammars and the additional benefits they offer. Finally, an implementation of a programming language is presented which is not based in any grammar, but in an ontology.

Keywords*:* Validation of programs, grammars, semantic validity, symbolic computation, ontology

I. INTRODUCTION

Grammars have been used extensively to validate the syntax of computer programs. Although the semantic validity of programs can be lifted adding more structures, for example with types; a complete mean of semantic validity cannot exist.

A. The Rice theorem

The Rice theorem [11] states that it is impossible to formalize a method to differentiate a single, no trivial semantic property to all computer programs. A trivial semantic property is a semantic property that is either always present or always absent in all possible programs. An example of no trivial semantic property could be "This program is harmful". As a consequence of this theorem a formal and deterministic malware detection program cannot be built [16]; most of them use heuristics for their detection procedures. The proof of this theorem shows that if there existed a method to differentiate such a semantic property, it could be used to solve the halting problem [15], which it is known to be impossible.

II. A COROLLARY FOR THE INSUFFICIENCY OF GRAMMARS FOR SEMANTIC VALIDITY

Corollary 1. *A computer language grammar cannot define the meaning of the programs or instances that belong to it.*

Proof. Let us define the property "This program does exactly what its author(s) wanted it to do" as a semantic property because it is evident that some programs have it while some others do not. According to the Rice theorem, there cannot exist a formal method to differentiate this property from any given program, including the use of grammars. ■

III. A THEOREM FOR THE INSUFFICIENCY OF GRAMMARS FOR SEMANTIC VALIDITY

Theorem 1. *A computer language grammar cannot define the meaning of the programs or instances that belong to it.*

The proof is based in the fact that if a program were semantically valid under a specific grammar, then there cannot be more than one such valid different meanings.

Before presenting the proof, let us see the following exercise from Douglas R. Hofstadter in his book "Gödel, Escher, Bach: an Eternal Golden Braid" [7], named the *pq-system*:

The *pq-system* contains the following three symbols:

$$p \;\; q \;\; \text{-}$$

There is the following definition: ***xp-qx*** is an axiom, whenever ***x*** is composed of hyphens only.

And there is just only production rule: Suppose *x*, *y* and *z* all stand for particular strings containing only hyphens. And suppose that *xpyqz* is known to be a theorem. Then *xpy−qz−* is a theorem.

According to this, the following are valid strings:

$$-p-q--$$
$$--p--q----$$
$$--p---q-----$$

While the following are not:

$$---p--q-$$
$$--p----q-------------$$

Afterward in the Hofstadter exercise, the reader is invited to find the associated semantic. This is, to define the meaning in the real world of the propositions created in the *pq-system*. Valid propositions must be paired with valid real-world facts, and conversely, non-valid propositions must be paired with invalid real-world facts.

The following **interpretation** arises almost immediately:

p	$\Leftrightarrow$	plus
q	$\Leftrightarrow$	equals
$-$	$\Leftrightarrow$	one
$--$	$\Leftrightarrow$	two
$---$	$\Leftrightarrow$	three
	$\vdots$	

So the string $- - p - - - q - - - - -$ means “2 plus 3 equals 5”.

We have a tendency to think that there exists a biunivocal correspondence between the *pq-system* and the semantic we have just found, but Hofstadter destroys it by showing the following, **alternative interpretation**:

p	$\Leftrightarrow$	equals
q	$\Leftrightarrow$	taken from
$-$	$\Leftrightarrow$	one
$--$	$\Leftrightarrow$	two
$---$	$\Leftrightarrow$	three
	$\vdots$	

Nonetheless, it can be argued that while there are two meanings, there exists an isomorphism between them and therefore they are equivalent.

In order to show an isomorphism, a rule of correspondence must be shown between the elements and operations of both structures. The hyphens representing numbers, are the same in both. It is evident that:

$$A + B = C$$

Is isomorphic with:

$$A = B \text{ taken from } C$$

And therefore they are equivalent, so they are the same semantics.

Proof. It is based in the previous scheme, this is, an incomplete grammar will be presented, in order to discover the meaning of missing symbols, and then a different non-isomorphic meaning is presented.

Let us suppose that some documents from a early discovered civilization are found. They show symbols that seem strange to us. Most of the symbols have been already deciphered. Two of the undeciphered symbols are:

- The symbol $\bowtie$, which is a binary operator, because it appears in the documents as $a \bowtie b$.
- The symbol $\star$, which is a unary operator, because it appears in the documents as $\star(x)$.

One of the documents is shown in Fig. 1.

Given

$$(a_1 \times a_2 \times \cdots \times a_n) \bowtie x$$

If we calculate:

$$\star(a_1) \Leftrightarrow A_1$$
$$\star(a_2) \Leftrightarrow A_2$$
$$\vdots$$
$$\star(a_n) \Leftrightarrow A_n$$

Then

$$\star(x) \Leftrightarrow A_1 + A_2 + \cdots + A_n$$

Figure 1. An example of a document found. It can be seen as a grammar.

In order to avoid confusion, the document is not shown in its “original” form, but it does in our notations and symbols. For example, the symbols $+$ and $\times$ represent usual addition and multiplication respectively, with the obvious exception of the $\bowtie$ and $\star$ symbols because their meanings are unknown.

Can the reader discover the meaning of the operations $\bowtie$ and $\star$, in such a way that we have a complete interpretation of the document, and therefore its semantic?

After remembering the logarithm laws we can state the following relation:

$\bowtie$	$\Leftrightarrow$	equality (=)
$\star(x)$	$\Leftrightarrow$	logarithm $log(x)$

In this relation, we could have chosen any valid base (a positive value excepting 1). For this specific case we chose the base 2.

Now let us build a specific case shown in Fig. 2 (if the document were a grammar one, this example would be a valid program).

Given

$$3 \times 4 \times 8 = x$$

If we calculate:

$$\log_2(3) \Leftrightarrow 1.5849625$$
$$\log_2(4) \Leftrightarrow 2$$
$$\log_2(8) \Leftrightarrow 3$$

Then

$$\log_2(x) \Leftrightarrow 1.5849625 + 2 + 3 = 6.5849625$$

Figure 2. An example of a “valid” program.

In order to find x, we have to do the inverse operation. The inverse operation of logarithms is exponentiation, so $x = 2^{6.5849625} = 96$.

So we can conclude that the document expresses the rule of logarithms that we enunciate as “The **logarithm** of a product **is equal** to the sum of the **logarithms** of the factors”, and therefore it is the meaning of the document.

Will there exist another interpretation? By way of explanation, will there exist another set of operations (different from equality and logarithms) that satisfies the same structure?

In number theory, exists the concept of **index** [8]. Suppose that we define any arbitrary base (for example 2) and we calculate how we can express all the numbers (under any modulo, for example 13) as powers of the base. We found that $2^{12} \equiv 1$, $2^1 \equiv 2$, $2^4 \equiv 3$, $2^2 \equiv 4$, $2^9 \equiv 5$, $2^5 \equiv 6$, $2^{11} \equiv 7$, $2^3 \equiv 8$, $2^8 \equiv 9$, $2^{10} \equiv 10$, $2^7 \equiv 11$ and $2^6 \equiv 12$ (all modulo 13). Expressly, 2 is a generator of the multiplicative group of integers modulo 13.

It is not true for any base. A base that shows this property is said to be a primitive root of the modulo. There are four primitive roots modulo 13: 2, 6, 7 and 11, and their correspondent indices are shown in the table 1.

Table 1. Indices modulo 13

Number (or residue)	*Primitive root 2*	*Primitive root 6*	*Primitive root 7*	*Primitive root 11*
1	0	0	0	0
2	1	5	11	7
3	4	8	8	4
4	2	10	10	2
5	9	9	3	3
6	5	1	7	11
7	11	7	1	5
8	3	3	9	9
9	8	4	4	8
10	10	2	2	10
11	7	11	5	1
12	6	6	6	6

The following valid relation can be established:

$$\bowtie \;\Leftrightarrow\; \text{Congruence, as in } a \equiv b \bmod 13$$
$$\star(x) \;\Leftrightarrow\; \text{Index, } \mathrm{ind}_{11}(x)$$

Any base of either 6, 7 or 11 could be used besides 2.

Building a specific case (a new "valid" program, for the same grammar) is shown in the Fig. 3.

Given

$$3 \times 4 \times 8 \equiv x \bmod 13$$

If we calculate:

$$\mathrm{ind}_2(3) \Leftrightarrow 4$$
$$\mathrm{ind}_2(4) \Leftrightarrow 2$$
$$\mathrm{ind}_2(8) \Leftrightarrow 3$$

Then

$$\mathrm{ind}_2(x) \Leftrightarrow (4 + 2 + 3) = 9$$

Figure 3. An example of another "valid program" for the same grammar.

In order to find x, we have to do the inverse operation. If we (reverse) lookup at the table 1, we will find that $\mathrm{ind}_2(5) = 9 \bmod 13$, then $x = 5$.

This new interpretation, corresponds to the rule "The **index** of a product **is congruent** to the sum of the **indices** of the factors".

Due to the similarity of indices with logarithms, indices are also known (outside theory of numbers) as discrete logarithms, and it is supposed to because they come from the same origin: the exponentiation in continuous and discrete domains, respectively.

However, the two semantics cannot be isomorphic, at least for the following reasons:

1. The sets that define the bases of both systems (logarithms and indices) are infinite but of different cardinality. The set of bases for indices is numerable while the set of bases for logarithms is not. Cantor proved that a rule of correspondence between them does not exist [3].
2. No algorithm for calculation of indices running in polynomial time is known[1]. If an isomorphism existed, it could be used to translate an index problem to a logarithm problem, which can run in polynomial time.

It has been shown an example of two different (non-isomorphic) meaning for the same grammar, therefore it cannot be generalized that a grammar necessarily defines a unique semantics. ■

IV. Ontologies as a mean of semantic validation of programs

An ontology [14] is a description of the world "as it is". It is a compendium of properties obtained under the criteria that they best describe the world, rather than being deduced from formal systems only. This last statement is not a trivial contempt to formal systems, it is because many properties of exact and non-exact fields of science cannot be obtained by deduction. As a very simple example, the value of the speed of light can neither be obtained exclusively from mathematical nor physical formulas, it requires observation and experimentation.

A. Ontologies and symbolic computation

Although ontology technologies exist, such as languages for ontologies, or documented ontologies for a wide scope of sciences, a different use of ontologies is presented here.

Symbolic computation works differently from the traditional one. It is based in rewriting rules, and therefore is a non-deterministic form of computation, because its behavior depends not only on the program and its inputs, but also on the available rewriting rules. It strongly relies on the principle of compositionality [9][10], derived of the fields of denotational semantics [12].

A set of rewriting rules define an ontology and hence a mean of semantic validity.

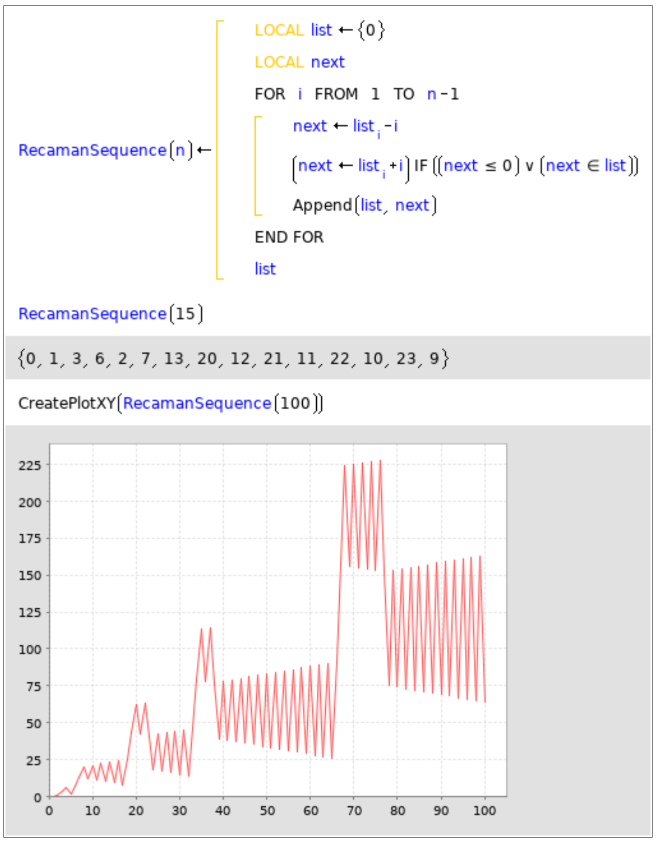
The expression 2 + 3 is semantically valid and it can be rewritten as 5 because there is a rewriting rule (addition of numbers) that can be applied. On the other hand, the following expression is an arithmetic division (the numerator refers to the planet, while the denominator refers to the chemical element):

$$\mathit{Mars} \div \mathit{Silver}$$

The expression is not semantically valid, there cannot be a rewriting rule, which everybody could define it as correct.

Rewriting rules provide both validity and functionality.

1 The ElGamal cryptographic scheme, and the Diffie-Hellman key exchange protocol are based on this property.

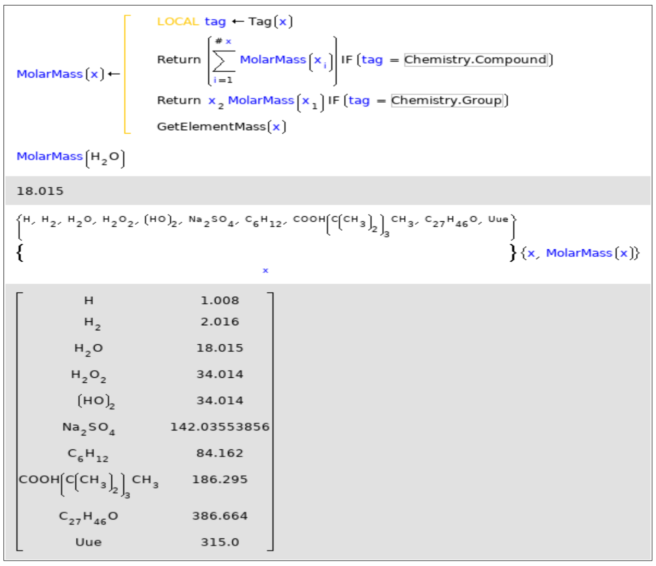


```
                         LOCAL list ← {0}
                         LOCAL next
                         FOR i FROM 1 TO n-1
                             next ← list_i - i
RecamanSequence(n) ←         [next ← list_i + i] IF ((next ≤ 0) v (next ∈ list))
                             Append(list, next)
                         END FOR
                         list

RecamanSequence(15)

{0, 1, 3, 6, 2, 7, 13, 20, 12, 21, 11, 22, 10, 23, 9}

CreatePlotXY(RecamanSequence(100))
```

Figure 4. Visual capabilites of the Fōrmulæ programming language.

B. *Benefits*

The usage of ontologies provides several benefits. The list does not pretend to be exhaustive:



```
                  LOCAL tag ← Tag(x)
                  Return [ Σ_{i=1}^{#x} MolarMass(x_i) ] IF (tag = Chemistry.Compound)
MolarMass(x) ←    Return x_2 MolarMass(x_1) IF (tag = Chemistry.Group)
                  GetElementMass(x)

MolarMass(H2O)

18.015

{ {x, MolarMass(x)} }
  x ∈ {H, H2, H2O, H2O2, (HO)2, Na2SO4, C6H12, COOH(C(CH3)2)3 CH3, C27H46O, Uue}
```

H	1.008
H_2	2.016
H_2O	18.015
H_2O_2	34.014
$(HO)_2$	34.014
Na_2SO_4	142.03553856
C_6H_{12}	84.162
$COOH(C(CH_3)_2)_3CH_3$	186.295
$C_{27}H_{46}O$	386.664
Uue	315.0

Figure 5. Chemical elements and compounds are first-class citizen in the Fōrmulæ programming language.

Dynamism. Grammars are static. Making changes to a programming language involves making changes in its underlying grammar and therefore in its tools such as compilers. Rewriting rules, on the other hand, are independent pieces that can however, interoperate with each other. A programming language based on an ontology built by rewriting rules is dynamic because their capabilities can be defined and evolve over time with consistent result, making such that language orthogonal [13].

- **Increasing validity**. As the number of rewriting rules increases, also the ontology does; and therefore the validation capacity of programs. Because ontologies are not formal systems their validation capacity are not limited by the Rice theorem.
- **Possibility of non-textual programs**. Computer languages are commonly expressed as plain text, because it is required by their underlying grammars. Using ontologies, text languages are not mandatory, creating the possibility of programs defined structurally instead or textually.
  Structural programming languages also opens possibilities for homoiconic languages. These are programming languages where the objects managed are defined in the language itself. It is highly desirable to create languages able to manage real world objects, such as mathematics elements directly in the language.

C. *Are ontologic languages necessary?*

Our knowledge is not perfect, it cannot be, there is a barrier imposed by the nature that prevent it. Kurt Gödel proved that no formal system can be both complete and consistent [5]. Stephen Hawking shows that the realism (scientific abstractions to explain natural phenomena) depends on the model used to its study [6].

Our knowledge is not static either. Gregory Chaitin asserts that the concept of truth is not absolute. It could either change with time, be sometimes random and even be accidental [2].

Going further, Paul Feyerabend in his work *Against the method* defines the epistemological anarchism which postulates that the scientific method, although it is valid, it is not the unique mean to promote the progress of the science. Other forms of non-scientific as holism (in opposition of reductionism), or even the serendipity, have produced great advances in science [4].

Nowadays, we have neural networks able to beat any human player in difficult games such as Chess or Go. Of course, there is a lot of theory and formal methods involved in that field of science, but these networks are able to do that because their neurons are in a state produced by the learning on millions of games previously played. The only formal elements initially introduced in these networks are the game rules. They are able to play (and play very well) "as is".

As a second example, in mathematics (a formal and exact science), what is the result of 0^0 ? There is not an agreement about the answer. For some fields, it is more appropriate the answer 1, and indefinite for others. The choice whether to define 0^0 is based on convenience, not on correctness [1].

V. FŌRMULÆ, AN IMPLEMENTATION OF A ONTOLOGY-BASED PROGRAMMING LANGUAGE

Fōrmulæ [17] is a programming language based in ontology. It has no grammar associated so it is not compiled but interpreted.

The associated ontology is not centralized in a work group or place. Rewriting rules can be written by anyone, thanks to a free, open source API is provided in order that they can be implemented in a regular programming language. Related rewriting rules are then grouped in units called packages that can be easily published, downloaded and installed. Because of it, the ontology is created by the community.

Capabilities of the Fōrmulæ language increase in time, it is orthogonal and homoiconic. Because it is not text based, its programs can be visualized in different forms, according to regional settings or user preferences, but mainly in mathematical notation.

Fig. 4 and Fig. 5 show visual capabilities of the Fōrmulæ programming language.

Corollary 2. *The Fōrmulæ programing language is Turing complete.*

Proof. It is enough to show that the Fōrmulæ language can be used to simulate a Turing machine as it is shown in [18]. ■

VI. Conclussions

The use of ontologies as a mean of semantic validation of computer programs is shown as an alternative to formal grammars. This opens new possibilities in language theories, such as the emerging of programming languages with dynamic capabilities, ortogonality and homoiconicity. Because the use of ontologies does not impose the use of textual programs, it allows the emerging of "visual" and structural languages and programs. The implementation also opens the possibility of using symbolic computing in multiple disciplines as a simple way to write computer programs.

VII. References

[1] Benson, D. C. (1999), "The moment of proof: Mathematical epiphanies," Oxford University Press, 1st. edition.

[2] Chaitin, G. J. (2007), "Thinking about Gödel and Turing: Essays on complexity 1970-2007," World Scientific Publishing.

[3] Ferreirós, J. (2007), "Labyrinth of thought: A history of set theory and its role in mathematical thought," Birkhuser, 2nd revised edition.

[4] Feyerabend, P. (2010), "Against Method," Verso, 4th. edition.

[5] Gödel, K. (1992), "On the formally undecidable propositions of Principia Mathematica and related systems," Oxford University Press.

[6] Hawkings, S. W. & Mlodinow, L. (2010), "The grand design," Bantam Books, 1st. edition.

[7] Hofstadter, D. R. (1999) "Gödel, Escher, Bach: an eternal golden braid," Basic Books, 20th anniversary edition.

[8] Nagel, T. (1981), "Introduction to number theory," Chelsea Pub Co, 2nd. edition.

[9] Pelletier, F. J. (1994), "The principle of semantic compositionality," Topoi, Vol. 13, pp. 11–24.

[10] Pelletier, F. J. (2001), "Did Frege believe Frege's principle?," Journal of Logic, Language, and Information, pp. 87–114.

[11] Rice, H. G. (1953), "Classes of recursively enumerable sets and their decision problems," Trans. Amer. Math. Soc., Vol. 74, pp. 358366.

[12] Scott, D. & Strachey, C. (1971), "Towards a mathematical semantics for computer languages," Oxford Programming Research Group Technical Monograph.

[13] Sebesta, R. W. (2010), "Concepts of programming languages," Addison-Wesley, 9th edition.

[14] Staab, S. & Studer, R. (2009), "Handbook on ontologies," chapter "What is an ontology?," Springer Publishing Company, Incorporated, 2nd edition.

[15] Turing, A. (1938), "On computable numbers, with an application to the entscheidungsproblem," A correction. Proceedings of the London Mathematical Society, Vol. 43, No. 6, pp. 544546.

[16] Ugalde, L. R. (2015), "Cooperative development and human interface of a computer algebra system with the Fōrmulæ framework," 21st Confer ence on Applications of Computer Algebra (ACA 2015), pp. 38–41. http://www.singacom.uva.es/ACA2015/latex/ACAproc.pdf.

[17] Ugalde, L. R. (2015), The Fōrmulæ website. http://www.formulae.org.

[18] Ugalde, L. R. (2018), The "Rosetta code" website, task "Universal turing machine", section "Fōrmulæ". http://rosettacode.org/wiki/Universal_Turing_machine#F.C5.8Drmul.C3.A6.