Signata
Annales des sémiotiques / Annals of Semiotics

12 | 2021
Sémiotiques de l'archive

An adaptive computational framework for sign production

Une perspective computationnelle adaptative pour la production de signes

Andrea Valle

Electronic version
URL: http://journals.openedition.org/signata/3266
ISSN: 2565-7097

Publisher
Presses universitaires de Liège (PULg)

Printed version
ISBN: 9782875622693
ISSN: 2032-9806

An adaptive computational framework for sign production

Une perspective computationnelle adaptative pour la production de signes

Andrea Valle

1. Introduction

1 Given the not so usual features of the following contribution in relation to semiotic literature, it might be relevant to point to out some general epistemological aspects at its origin. First of all, the presented research is intended as a reflection on some foundational semiotic concepts/constructs, in particular in relation to Eco's semiotic theories. Second, the paper mostly describes a *Gedankenexperiment*: it aims at defining a very simple semiotic model that can be studied empirically but always *in vitro*. Third, as the model is a formal one, it can be implemented computationally: in this sense, the paper may also be read as an application in computational semiotics, or, maybe better, as an example of a possible computational approach to some aspects of semiotics. Finally, as the research has been developed in close contact with a computational implementation, it is also a tentative demonstration of the semiotic richness of programming languages and their usefulness as tools for semiotic theoretical research, in the perspective of a constructionist approach (Harel & Papert 1991) to—so to say—"theory-making".

2 The observations that I will try to develop have been prompted mostly by five theoretical sources. The first is Umberto Eco's theory of modes of sign production (Eco 1976). Such a theory of sign production takes into account, rather then already established semiotic constructs, the ways in which such constructs are created: in short, it deals with semiotic dynamics, i.e. the way signs are produced. In order to do so, the theory defines various ways in which expressions are generated and related to contents. To this aim, Eco describes four parameters: i) the "physical labor" required to produce expression ("recognition, ostension, replica, invention"); ii) the "type-token ratio" (be it *facilis* or *difficilis*); iii) the "continuum to be shaped", whether or not it is

shared between expression and referent ("homo-" vs "heteromaterial"); iv) the "mode and rate of articulation" ("grammatical units" vs "texts"). From such a theoretical proposal, some elements are particularly relevant:

- sign is properly intended as a "sign function", that is, something that relates two planes as a result of a pragmatic effort;
- the four parameters provide a compact organisation for a sign function typology, that can be further explored;
- the focus is not on cognitive operations or on the description of contents (cf. Greimas), but specifically on how expressions are produced in relation to contents;
- recognition is a semiotic labor. This is interesting but paradoxical: in recognising an imprint, the recogniser has not produced it. On the other side, when speaking of ostension, replica, invention, Eco's theory assumes the perspective of the producer: the latter is the subject responsible of sign production. This requires to explicitly double the perspective. As I tried to show elsewhere (Valle 2007, 2017a), recognition properly acts as a *metalabor*, a framing activity to be presupposed in order to access the other three labors.

3 The second source I will take into account is again by Eco, in this case the essay *Generazione di messaggi estetici in lingua edenica*.[1] Here Eco proposes a *Gedankenxperiment,* starting (and departing) from the Project Grammarama by Miller. Eco discusses a basic language made up of only two symbols (A, B) and a grammar (in the form: X, *n*Y, X) that allows to validate well-formed strings (e.g. ABBBBBA, BAB, etc). Eco takes into account various possibilities in word creation, as allowed by the AB grammar, in relation to various states of a similarly simplified world. In this way, he is able to discuss many typical, *lato sensu*, rhetoric usages of the language that emerge even in such a simplified model. Particularly striking features in Eco's essay are:

- the use of simple strings and a formally defined grammar to define a language;
- the idea of a model *in vitro* to feed a *Gedankenexperiment.*

4 The third source is Eco's most relevant late contribution to semiotics, *Kant and the platypus* (Eco 1997). The work begins with an essay *On Being* (nothing less!). But even when wandering in such a philosophically ponderous ontological district, Eco does not renounce his inclination to abstract mental models. In the essay (cf. section 1.8, "A model of world knowledge"), Eco proposes a simplified model of a World that is thought of as a set of "stoicheia" to be represented by a set of "symbols" in a Mind. As the Mind represents the World, the work of the former is properly interpretation. Given that, Eco takes into account various (quantitative) relations between these two sets, the stoicheia that make up the World and the symbols the Mind is built of: as an example, their respective cardinalities (the number of elements), and the ratios between cardinalities of the two sets. Some interesting suggestions from *On Being* are:

- as in the case of the AB language, this is a *Gedankenexperiment* and it is performed by using discrete units, both on the World and the Mind side;
- Eco considers the dualism implicit in the model just as a simplification. Properly, the Mind is made up of the same stoicheia of the World: the consequence is that the World represents itself (Eco adds: in animals, vegetables, and maybe also in minerals, i.e. in the "silicon epiphany of the computer", 1997, p. 38). This is a sort of Peircean monism, that turns into dualism not from an ontological perspective but from a functional one, where "function", one could say, properly refers to sign function, coupling symbols and stoicheia;
- Eco is still allergic to the problem of the origin of sign. Yet, he touches such a delicate point in various essays. In *On Being* too, this Mind/World model seems to open the very issue:

rather than in historical or neuro-cognitive terms, by posing the question of how a semiotic dynamics may be described.

5 The fourth source is *Hidden Order* by John Holland (1995), the father of the so-called genetic algorithms. In this essay recollecting his Ulam lectures, Holland describes various adaptive computational models, all based on a string formalism. That is, all elements are described by means of strings of symbols and their relations by means of operations on these same strings. Holland begins with modelling communication between agents and finally proposes a simplified model of an ecological (multiagent, adaptive) environment. While this final model is far beyond the topics of my discussion, Holland first discusses communication among agents and the exchange of information with the environment. Interestingly:

- the model is a way to computationally perform a *Gedankenexperiment;*
- Holland works exclusively with strings of symbols, to gain formal control over all operations;
- the author provides various way to deal with interpreting rules, meant as matching patterns between a state of the world and a state of symbols.

6 Finally, the following considerations can be seen as reflections on "worldmaking" (Goodman 1978). Worldmaking is the constant activity that generates the multiplicity of worlds we live in, through the descriptions that are generated by various human practices and cultures. In relation to this multiplicity, Goodman observes that "the issue between monism and pluralism tends to evaporate under analysis. If there is but one world, it embraces a multiplicity of contrasting aspects; if there are many worlds, the collection of them all is one" (1978, p. 2). Moreover, there is a recursive movement in worldmaking, as the latter "always starts from worlds already on hand; the making is remaking" (1978, p. 7): in the making process existing worlds are fed back to generate new ones.

7 The research strategy that I pursued and that I will describe in the next sections has been based on three main assumptions:

1. define simplified models using strings, still capable of capturing some aspects of semiotic theory;
2. use interactively programming as a way to experiment, by trials and errors;
3. write everything in a programming language. Comments in the code, added as "semiotic insertions" (Valle & Mazzei 2017), allow to include observations written in natural language. As a consequence, most of this text has been originally written directly in the program files. This approach can be seen as a variation on the so-called "Literate programming" paradigm (Knuth 1984).

8 The programming language that I chose is Python,[2] a simple, powerful, well-known, high level language, whose syntax closely mirrors the so-called "pseudo-code".[3] I will not include in the following all the code that has been written to implement the observations (thus, I will not adhere to a strict literate programming methodology), but I will still provide some code samples together with some console outputs.[4] These excerpts will be printed with a monospace typeface.

2. The World and hence the Mind

9 Assuming a monist principle (even if in the light of Goodman's *caveat*), we have to define Eco' stoicheia so to be able to use them to create the world and at the same time to create an interpretation. The first assumption is that our world is made of characters, combined in a string. An interpretation is thus something associated to pieces of the world (i.e. substrings) and made up of the same elements, i.e. characters.

10 Hence, in such a model the interpreter is something that puts together pieces of the world. This is a classic assumption in semiotics, as it mirrors the correlation between the two Hjelmslevian planes of expression and content (Hjelmslev 1961). Reading back Hjelmslev, for Eco (1984a, p. 53, cf. Valle 2007, p. 411), the semiotic function locally couples the purport, and the two resulting poles can be defined as "expression" and "content" (hence on, E and C). The definition of an interpreter is typically outside a Hjelmslevian perspective. In our context, the Interpreter should be made up with the same pieces of the World, otherwise it would be literally alien to the world itself. As the experiment is performed in the Python language, it could be interesting to use its building blocks. Like the vast majority of programming languages, Python is written in the ASCII character system, that uses a 7 bit encoding, reserving a subset (first 32) for control (non printable) characters while providing 95 printable characters.[5] Python is thus "literally" made up of this 95 characters. They can be obtained by evaluating the following code:

```
stk = [] # stoicheia
for c in range(95):
   stk.append(chr(c+32))
```

Stoicheia stk are thus:

```
! " # $ % & "" ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A
B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \\ ] ^ _ ` a b
c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

11 The first is the character representing the blank space. To make things easier (but not different) we skip some special characters that disturb string construction, such as ", ', \

```
stk = [ st for st in stk if st not in ["\"", "\'", "\\"]]
```

12 A starting assumption is that a world is a random (as far as we know) sequence of stoicheia. As an example, a small world of 100 elements ca be something like:

```
L%pjVgx6HDFLTZEb+)([PmUJV<(nZ&Ml/^&-rvm%P]Y7,1:#_6@MUnrI4NwXg}
^`8O3QOuN0ld`Jy>{
ib9g>MM_!.j|s`- *OEgj
```

13 Just to provide an example of how to obtain such a result, the following code creates the previous world:

```
import random
def makeWorld(st, dimension, seed = 1932):
   random.seed(seed) # seeding the random generator
   world = []
   for i in range(dimension):
     world.append(random.choice(st))
   return world
```

```
def aggregateWorld(world):
   agg = ””
   for st in world:
     agg = agg+st
   return agg
world = makeWorld(stk, 100, 1932)
print(aggregateWorld(world))
```

14 The first line imports a module required for random generation. Then, two functions are described. The first (makeWorld) returns a random list of characters. Characters are to be taken from st, dimension is the string length, and seed is a way to tune the pseudo-random generator so that it can reproduce exactly the same pseudo-random sequence, for sake of persistence. The second function (aggregateWorld) aggregates a world list (i.e. a container of separated elements) into a single string. The last two lines respectively call the makeWorld function so that it returns a 100-character sequence from stk, seeded by 1932, and then print the list in a string-like fashion. I will not enter into code details any more, first because the code itself is neither advanced nor particularly elegant, secondly because the code outside the context of a program interpreter would simply clutter the text. The previous discussion was just intended as a methodological example.

15 A first observation is that the world is made up of the same stuff (stoicheia) of the metalanguage we are using to describe it: the latter is a specific arrangement of the first.

16 Once we have the world, it is possible to think about its representation. As we cannot properly exit the world, a second constraint is that interpretation should happen within the world itself, by means of the E/C coupling mechanism discussed before: that is, a part of the world is to be used as expression for another part of the world. Classic cryptography provides a useful starting example. Substitution cipher, dating back to Julius Caesar (and being at the core of E.A. Poe's *The Gold-Bug*), is a basic cryptographic technique that associates a letter from an alphabet to another letter of the same alphabet (Biggs 2008). Typically, there is a shift rule that compactly describes the operation, based on alphabetical order: so, taking into account the Latin alphabet, in a shift by 3, an a becomes a c.

17 But let us suppose a substitution in which each character is associated to another character. For sake of readability, let us suppose from now on that stk includes only alphabetical lower case characters in alphabetical order.

`abcdefghijklmnopqrstuvwxyz`

18 A representation r can be a random permutation over stk:

`kbtrnwfjiszghevyaqlcmdopux`

19 A code here can be simply thought as a mapping from r onto stk by associating each character in one list to the character in the same position in the other one. So, if this is a possible world:

`kbtrxnrvfjijxkmoipzcbbolsmknzygbsyxobksdozbctusxbloynfycdgaxpfhkmstjelvnrwopflelluldsqpkvhwrpf`

this would be its representation in r:

`zbcqpeqdwsispzhviyxtbbvglhzexufblupvbzlrvxbtcmlpbgvuewutrfkpywjzhlcsngdeqovywgnggmgrlayzdjoqyw`

20 This cryptography-inspired example is oversimple, nevertheless it has some interesting features:

- it is a typical example of code meant as an association between two lists ("s-codes" in Eco 1976);
- it is a typical example of a specific labor: replica. Each character type allows to generate tokens by perfect replication;
- it is a typical case of *ratio facilis*. As Eco (1976) says, a machine working by *ratio facilis* can produce expressions even without having any information concerning the content side. It could properly work by itself, generating expressions that do not match any state of the world. A random string from r is a possible expression that can be generated without taking into account the world. We could say that it is an expression of a possible content;
- there is a perfect isomorphism between representation and world (cf. Eco's notion of "total reversibility", 1976, p. 23). In fact, it is evidently possible to define a reverse function that returns the world (content) from its expression.

21 In our case, stk represents content types and r expression types. In fact, they can have various occurrences (tokens) in the world and in the representation. An evidently open issue is that we are mixing referent and content. To put it better, properly there is no difference between the two elements. At the same time, this implies that there is also no difference between expression and referent. Expressions, like contents, are parts of the world. If not, the monism principle would not have been respected. The cryptography scenario deals with one of the scenarios discussed by Eco in *On Being*, the simplest one: the two sets—stoicheia and symbols—have the same cardinality and there is a one-to-one mapping among elements on the two sides of world and representation. As Eco underlines, a "real" situation (even if made up of characters) seems indeed much more complex. In order to address such a complexity issue, these considerations can be further expanded by introducing some aspects discussed by Holland (1995). Holland has mostly worked on adaptive systems, that is, on various formalisms capable of describing dynamic systems that change in relation to a certain environment by increasing their performances. In the same vein, in *Hidden Order* Holland has proposed a framework in which an agent receives messages from the environment or other agents. For theoretical reasons, Holland's representation format is based on bits: all the world (and its representation) is thus encoded in terms of sequences of 0 and 1, that is, by means of a binary alphabet. As the environment is encoded as a binary string, the agent is provided with a set of rules that allow the latter to extract information from the former. Properly, the agent detects blocks of the world as "messages": in this sense, Holland proposes a theory of communication. In Holland's framework, a "rule" is a pairing that matches a message (better: a class of messages, i.e. a string of 0 and 1) to a certain reply (another string of 0 and 1). To put it with the biological inspiration, for a frog: if there is a fly (message) then extend your tongue (reply). It is worth underlining again that Holland formalizes this relation as a rule that couples two strings of bits: if a rule possessed by the agent matches a message (from the environment), then the associated behaviour is triggered. If the environment is defined exclusively in terms of strings, then the resulting behaviour at the end will be (monistically, one could say) a string. In relation to the previous discussion, it is apparent that such rules can be thought in terms of sign functions. For Holland, a crucial element for an adaptive (thus: dynamically evolving) system is "credit assignment". If the message is matched by a rule it triggers the behaviour but it also receives some kind of credit. Credit assignment

indicates that the rule is adequate to the message. Credit assignment is instrumental in favouring good matching rules, i.e. adaptive ones. By this, a certain agent can adapt its set of rules to the environment over time. In this sense, adaptive systems are intrinsically dynamics: they deal with behaviour changes. With the aim of implementing credit assignment, Holland proposes to add to his bit alphabet ({0,1}) the # sign, what he calls a "don't care" symbol (hence on: dontcare, to indicate its metalinguistic nature). This important addition indicates that in the # position every character can be matched. As I will not strictly adhere to Holland's formalism (and, moreover, # indicates a comment in Python), in the following I will replace the # with the dot (.), still an ASCII symbol. So, an expressive type like:

```
agh.uj
```

matches all the messages (coming from the environment or from other agents: properly it is the same) having whatsoever character in the . position while having the specified characters in the other positions, like aghduj, aghouj, aghruj, etc. The dontcare symbol is instrumental in implementing in the model a notion that can be compared to relevance in Prieto's terms, and to the selection of distinctive features in Jakobson's structural phonology (Prieto 1975; Jakobson & Halle 1956), as it allows to differentiate types from tokens.

22 Back to the basic cryptographic example, we can consider the World as a set of encrypted messages to be decrypted by an agent, the Interpreter, by means of its Code. The Code is a set of sign functions that assign types (expression types, or forms in a Hjelmslevian parlance, e.g. k) to these messages (expression tokens: there can be many k tokens), map these types to a different set (content types, in the cryptographic example: a), and result in a certain behaviour (the generation of content tokens: various a occurrences). In short, the World provides expressions to be traced back by the Interpreter to expression forms, and expression forms are associated to content forms that lead to actual contents, as described in the Code. In a character-based world, the only possible behaviour to be triggered is indeed character generation. The cryptographic world is twice simple: first, it encodes characters one-to-one and, second, it is totally deterministic. By disrupting these two constraints, we may gain in complexity. A code can thus match multiple character strings by making use of dontcares, so to add a certain indeterminacy. So, the following:

```
cnf. : hdnl
```

is a sign function, in which the : stands for the semiotic relation between planes. It defines two types. The expression type is able to match all the four character strings beginning with cnf and ending with whatsoever character: they are mapped onto the content type hdnl, and results in the generation of a string hdnl.

23 It is possible to extend the formalism by proposing e.g.

```
cnf. : h.nl
```

24 This would mean not only to match all the expressions before, but to generate one of the possible tokens described by the type h.nl, e.g. honl, hfnl, hznl, etc.

25 Applying dontcares to the right side of a rule is a feature not present in Holland's theoretical framework, because in the latter the matching rules (with dontcare symbols) are used to trigger only specific behaviours. But the idea can be semiotically generalised as a sort of double matching system. From a semiotic perspective, it can be observed that such a system still implements reversibility between expression and content, and treats the two planes in the same way: after all, they are all pieces of the

same world as they are made of the same stuff. Yet, an indeterminacy feature is added, that differentiates types from tokens.

26 Starting from these observations, a *code* can be defined as a dictionary coupling *Ef* and *Cf*, i.e. expression and content types (forms, *f*), into sign functions. In code notation, the . character indicates a dontcare symbol. Hence on, rather than single characters, I will use triplets for *Ef* and *Cf*, i.e. sequences of three characters. This is a design choice that has proven to be apt and that can be easily generalised. An example of a code, implemented in Python, is the following:

```
code = {
# expression form : content form
'awa':'bbf',
'z.z':'g.g',
'k.k':'c.c',
'l..':'ooo'
}
```

27 Given a world and a code, it is thus possible to define an interpretation. The latter is the result of all the matches on the world: for each match, a new content token is generated accordingly. As a last code example, the following is a possible implementation:

28
```
def makeInterpretation(world, code):
 interpretation = []
 for d in code:
  matched = re.findall(d, aggregateWorld(world))
  for m in matched:
   cnt = code[d]
   interpretation.append([m, exrex.getone(cnt)])
 return interpretation
```

29 The function makeInterpretation takes a world and a code, for each *Ef* (left side of the code) it searches all the possible matches in world, and for each match it generates a content from the relative *Cf* (right side). So, given the code above and the previously created world, an interpretation is the following, where on each line the left side is a match and the right side is a generated content:

```
[['awa', 'bbf'],
 ['zrz', 'ghg'],
 ['zfz', 'geg'],
 ['kgk', 'cjc'],
 ['lsm', 'ooo'],
 ['loy', 'ooo'],
 ['lvn', 'ooo'],
 ['lel', 'ooo'],
 ['lul', 'ooo'],
 ['lir', 'ooo'],
 ...,
 ['lkk', 'ooo']]
```

30 From these first attempts some relevant features emerge from the model:

i. the presence of dontcare symbols differentiates types. Both *Ef* and *Cf* may contain or not dontcares. We could say that a type without dontcares is a closed type. Conversely, a type with at least a dontcare symbol is an open one. A closed sign function has both closed types.

A partially closed sign function has one closed type (on E or C). Thus, the cryptographic interpreter has all closed types, both on E and C, i.e. all closed sign functions. It represents the maximally deterministic interpretation, what is generally considered as the "rigid" model of "code". We may call such a code a "c-code" (closed code) vs an "o(pen)-code";

ii. the replica labor is still at work. There is no motivation between E and C apart from their established relation. Speaking with Eco, we do not need semantic instructions to generate expressions (in fact, we can generate expression tokens from an *Ef* without taking into account the associated *Cf*);

iii. a definition of interpreter results: an *interpreter* is a set of sign functions (the *code*) plus an *interpretation procedure* (an algorithm like makeInterpretation);

iv. recognition is indeed the task of the interpreter, that is, a metalabor allowing the replica one;

v. closed and open types allow to differentiate the mode of articulation. If we stick to E (but the same considerations apply to C), a closed type exhaustively describes the expression: it is an allography in Goodman 1968's terms. If the type is open, there is a variable amount of freedom in expression. So, in this model the axis allography vs autography is a continuum (or better in this case: a gradatum). Properly, such a feature, let us say *openness*, may even receive a score, as it is enough to count the number of dontcares. If openness is = 0 (no dontcares), then the type is strictly allographic (closed type), but in our case openness can be 1 or even 2 or 3, that is, there is a variable degree of allography.

31 Let us explore openness by taking into account some extreme cases. In this sign function:

`… : wow`

32 *Ef* is fully open. This means that it matches everything. And it generates as a content: wow. One might call such a sign function "stupor mundi". By matching all the expressions, everything can lead to interpretation, it is significant, so to say. A match always happens, but intuitively it works poorly. Following another suggestion by Holland, we can consider what we have called openness as a measure of the "specificity" of a type, measured as the number of characters vs dontcares, and we can assign a score to each type. So, the … has specificity = 0. Indeed, the same situation may occur for C type. Let us suppose we have:

`aaa : ...`

33 A certain expressions leads to whatsoever content. This is an oversimplified but not necessarily wrong description (at least, in terms of our model) of various interpretative phenomena addressed in literature as deconstruction or overinterpretation: given a specific expression, any content can be produced. *Cf* has a specificity = 0. From this, we can propose an interpretation score *is* for the sign function resulting from the product of E and C specificity *s*, i.e. $s_e \times s_c$. In both our previous cases, the total score is, respectively, $0 \times n$ and $n \times 0 = 0$. More generally, "catch-all" … types (both *Ef* and *Cf*) result in an interpretation score = 0. To push forward the exploration, the most bizarre theoretical case is the sign function "anything goes", i.e.

`... : ...`

34 It matches everything in the world and it can produce everything as its interpretation. Its score is $0 \times 0 = 0$. Intuitively, it is not very useful in order to deal with whatsoever environment.

35 If it is possible to compute an interpretation score *is* for a sign function, then the same notion can be applied to a complete interpretation of a world. The makeInterpretation

function can be easily updated so to store the score for each interpretation, giving a result like this for the code and the world presented before:

```
[['awa', 'bbf', 9],
 ['zrz', 'gzg', 4],
 ['zfz', 'gfg', 4],
 ['kgk', 'csc', 4],
 ['lsm', 'ooo', 3],
 ['loy', 'ooo', 3],
 ['lvn', 'ooo', 3],
 ['lel', 'ooo', 3],
 ['lul', 'ooo', 3],
 ['lir', 'ooo', 3],
 ...,
 ['lkk', 'ooo', 3]]
```

36 By looking back at the code, it is easy to see that for each matching in the world the associated score is the result of computing the specificity of the relative sign function. The relevance of dontcares in favouring matches is also apparent. Simply by summing up all the sign function's *is*, we obtain an overall *is*, the interpretation score for a given world given a certain code.

37 The purpose of the previous formalisation is to provide a credit assignment mechanism for interpretation. Of course, such a mechanism is useful only in comparing different interpretations. Before following this path, another look to modes of sign production is required.

3. Ostension and invention

38 If we look back to the model proposed up to here, it might be of some interest to discuss how it can deal with the various modes of sign production. First of all, recognition is here intended as a metalabor: properly, it is the framing activity of the interpreter, as it describes the whole matching procedure that applies a code to a world. The replica labor is always at play in the model, as a code works in a (variably) allographic regime, by recognising E tokens through *Ef* and by generating C tokens according to *Cf*. Yet, the replica mechanism is strongly weakened by means of dontcare symbols, so that, while the code is still a coupling of s-codes, at the same time it takes into account a variable degree of determinism. *Ef* and *Cf* thus select a variable amount of relevant, distinctive features (that is: of specificity), ranging from a strict deterministic behaviour to a catch-all one.

39 What about ostension and imprint? For Eco, these two labors, together with replica, complete a typology of sign functions. A shared feature is that these two labors imply a sort of "thick" relation with the world, theoretically difficult to disentangle. Ostension "occurs when a given object [...] is 'picked up' by someone and shown as the expression of the class of which it is member" (Eco 1976, p. 225). A typical aspect of ostension is "homomateriality" between the expression and the possible referent, says Eco: in ostension the object is "viewed as an expression made with same stuff as its possible referent" (Eco 1976, p. 224). Ostension and homomateriality are coupled in Eco's typology, as the only sign function type to feature homomateriality results from

ostension labor. How to define ostension in our simple model, in which properly there is no referent at all? Let us consider this sign function:

```
aab : aac
```

40 *Ef* and *Cf* share two characters in the same positions. The two-character chunk aa stands (E) for the same chunk aa (C). This "sameness" is a possible way to describe homomateriality in our model. Sameness must include position in order to provide a stronger constraint, as in a small stk set the simple presence, regardless of position, would indeed be a poor characterisation. Same position means that possibly the same character substring can be found in both types at the same place.[6] Like in the case of replica, ostension can be turned into a degree. The ostensive degree can be defined as the number of shared characters in the same position. Thus:

```
abc : def has a degree = 0
abc : aef has a degree = 1
abc : abf has a degree = 2
```

41 By the way, by using dontcares a sign function like

```
abc : de.
```

may have a certain ostensive degree or not, depending on the resulting token (e.g. dec). This means that a sort of accidental ostension component could occur.[7]

42 The maximal case of ostension happens when the two types *Ef* and *Cf* coincide, as in:

```
abc : abc
```

43 Is there a viable example of such an "identity" sign function? The expression is mirrored in the content. While bearing in mind that we are still discussing a simple, abstract model, let us consider the situation in which an object stands exactly for its class. A sample of an object (e.g. a scarf shown in a shop window) represents exactly its class (the abstract scarf type). In this sense, in the model a code made up of only pure ostensions mirrors exactly that part of the world that it represents. Debate on ostension and its limits seems to be aptly represented by such a paradoxical situation. Like in the replica case, a characterisation of ostension as a degree allows to weaken the purity of the notion. A small piece of cloth representing the cloth type, to be sold in a variable size, does not mirror the cloth in itself. This case might be represented in the model with a notation like this:

```
abc : abd
```

44 To characterize explicitly ostension, the following specific notation could be proposed:

```
abc : O.__
```

45 Here, the O character is a special character (like .) that specifies that the following type is (partly) an ostension. The character O is followed by the projection description, in which another special symbol, _, requires to copy the character in that position from *Ef* into *Cf*, resulting in the previous case in:

```
abc : .bc
```

46 Special notation clearly differentiates explicit ostension from the resulting, converted sign function above here. The latter can be conceived as an implicit form of ostension. It is evidently possible to define a simple string replacement function that converts ostension into a standard type. The call of such a function—let us say expandOstension —would require to pass *Ef* and *Cf* (the latter in the ostension format). In the previous example:

```
expandOstension("abc", "O.__")
```

will output the *Cf* .bc, as required. I will rediscuss this rewriting procedure later. Before, let us consider the remaining labor—invention—by passing through imprints.

47 If ostension has a strong theoretical link with homomateriality, the case of imprint is the pivotal element in discussing the so-called *ratio*, the type-token relation. Ratio can be *facilis* or *difficilis*. It is *facilis* if there is no specific (recognised) relation between the two types *Ef* and *Cf*. On the other side, in case of *ratio difficilis*, the expression is "motivated" by the content. Motivation has a twofold explanation in Eco:

i. there is a causal link. A fact of the world is recognised as an imprint (E) if it can be traced back to a causal framework (C). An imprint is the sign of the (past) presence of an "imprinter", so to say with a barbarism;

ii. by considering such a backtracking problem, that applies to imprint, Eco proposes a sort of Turing test, hypothesizing the involvement of a *machina semiotica*:

- in the case of *ratio facilis*, "objects could be produced by a suitably instructed machine which only knows expressions, while another machine could assign to each expression a given content, provided that it was instructed to correlate functives" (1976, p. 219);
- on the contrary, in the case of *ratio difficilis*, "a machine instructed to produce these objects should be considered to have also received semantic instructions. One might say that since it is instructed to produce expressions, it is being fed with schematic semantic representations" (1976, p. 219). Eco adds that in *ratio difficilis* "provided that the projection rule is constant, the results obtained by manipulating the expression are diagnostic or prognostic with respect to the content" (Eco 1984a, p. 45, transl. by me, as the excerpt is absent from the English edition).

48 So, the process of inferring the ratio may be described by the following algorithm (Valle 2017):

i. collect the objects as expression-units;

ii. abduct a production rule for expression (the *modus operandi* for the Expression Generator);

iii. manipulate the abducted Expression Generator;

iv. check if there are or have been (prognosis/diagnosis) changes in the content of the resulting sign-function;

v. in positive case: it is a *machina difficilis*, where the Content Generator shares some features of the type with the Expression Generator; otherwise it is a *machina facilis*, where Content- and Expression Generators have each their own type.

49 While imprint is a type of sign function, invention is a labor. But Eco says: "if [...] imprints (even if accidentally replicated rather than recognised) were not been classified as straightforward transformations under the heading of inventions, this was for good reason. In the case of an imprint the content-model already exists. [...] When replicating an imprint one is mapping from something known" (Eco 1976, p. 249).

50 To sum up:

- inventions and imprints share a feature: content and expression share (part of) the type in terms of an indexical relation;
- imprints are conventional, inventions not (yet).

51 In our model, there is at the moment no place for type creation *ex nihilo*, that is: for learning, that is: for emergence of new sign functions, that is: for adaptation. I will thus use imprint as the general term for invention (a too generic term, in my opinion). To describe this type of sign function, I propose as a feature the presence of a procedural mapping between the types on both sides.

52 In short, there is a rule (a generative procedure) that allows to get from *Ef* to *Cf*: *Ef* becomes "diagnostic" of *Cf*. A procedure that creates a *Cf* can be thought as a generation, extraction or mapping algorithm linking it to *Ef*. The introduction of such a feature is theoretically straightforward but makes the notation much more difficult than in the case of ostension. In fact, properly *any* algorithm can connect E and C types. How to notate it? Let us consider this case:

```
abc : bcd
```

53 Here the rule connecting *Ef* to *Cf* is obvious. Each character in *Ef* is replaced by the next alphabetic character in *Cf* (a shift by 1 in a Caesar cipher). As an example, if *Ef* is ab. then, the following so-called list comprehension notation in Python applied to ab. outputs the desired result (in form of a list to be concatenated into a string) while leaving dontcares unchanged:

```
[chr(ord(x)+1) if x != ”.“ else x for x in ef]
```

54 Once concatenated, the output *Cf* would be bc..

55 Assuming for convention that each character is represented by c and the type by eF we can represent an imprint sign function like the one before with something like:

```
ab. : I[chr(ord(c)+1) if c != \”.\" else chr(ord(c)) for c in eF]
```

56 As we did with O in ostension, *Cf* is prepended by a special character, I. The notation is concise but very complex and, above all, strictly related to Python implementation. But, as properly any algorithm can link *Ef* and *Cf*, a specific notation is required to encode automatically a mapping procedure into a short string. The relevant point here is that, as in ostension, such a notation requires a replacement procedure to get the actual *Cf* (bc.). And, exactly as in ostension, it is easy to define a function—say expandImprint—to be passed the *Ef* and the *Cf* in explicit imprint notation (as a procedure), getting back the actual, implicit *Cf* (bc.).

57 To sum up, we have three plus one labors:

1. *recognition* works a framing metalabor, i.e. the labor assigned to the interpreter that properly extracts contents from the world;
2. *replica* acts as an association between types, with no special relation apart from association itself;
3. *ostension* includes a subset (position matters) of *Ef* in *Cf*;
4. *imprint* (as a generalized invention) is described by a procedure that maps (in various ways) *Ef* into *Cf*.

58 An interpreter is described as i) an interpreting procedure plus ii) a code that maps expression to content.

59 In order to have a working procedure for our model, I have proposed two conversion functions turning ostension and imprint notation into a standard, replica-style one. Such procedures have been initially devised only as technical ones, that is, in relation to implementation. But this expansion of ostensions and imprints into an implicit form can be projected onto the theoretical level. At the end, the converted, implicit, sign functions only include replicas. And, in our example, the two resulting *Cf*s have exactly the same structure (bc.). Of course, the required relations between *Ef* and *Cf* are still present, but while in the sign functions with O and I notation they were explicit, in the converted code they become implicit, embedded into a standard replica model. Has this difference a theoretical value? It may look like the interpreter is provided with various

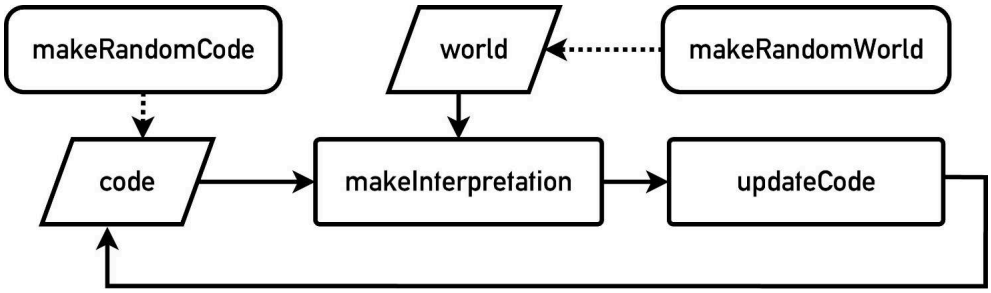
ways to describe *Ef*/*Cf* relations, according to Eco's typology of labor. This is relevant because it explicitly describes various, possible relations between E and C to produce interpretation. They are part of the competence of the interpreter, rules to create content, and this feature may provide some hints in the context of sign function creation. On the other side, the converted code demonstrates the basic notion of sign: something stands for something else, as in the classic motto *aliquid stat pro aliquo*. This notion is indeed captured by replica. Not by chance, semiotics as a discipline has extensively worked on such a paradigmatic case. In our model, the dontcare symbol weakens the deterministic strength of the replica and "opens" the interpretation (as multiple outputs -contents- are possible, while matching multiple inputs, expressions). Thus, such a conversion from O and I forms, that can be called "replica-reduction", is a framework that takes into account the basic concept of sign, still allowing the implicit encoding of ostension and imprint. In short, the model allows the description of a sort of semiotic dualism, opposing a basic semiotic working mode, embodied by replica, and a set of (at least two, following Eco) operations leading to such a (final, implicit) form. Eco says that the imprint is a convention: it could be argued that it can turn into a perfect conventionated replica exactly because, notwithstanding its generative construction, in order to operate it must be replica-reduced. Replica-reduction is thus a framework that takes into account the basic concept of sign, still allowing the implicit encoding of ostension and imprint. For this reason, hence on I will take into account only replicas: an easier move indeed, but hopefully not totally unjustified.

4. Adaptive hypotheses

60 Credit assignment is useful only in a dynamic paradigm, that is, as a way to evaluate a code in order to increase its performance. This is the meaning of "adaptive": to increase a system performance by taking into account an evaluation of its output in relation to an environment. Can we introduce a positive feedback loop so to increasingly generate new, better rewarding *Ef*s? That is, can we turn sign production from a description into an adaptive system, *à la* Holland? By assuming credit assignment as discussed before, we can only measure the system capability to match expression: developing this idea, for the moment I will not consider the obtained contents.

61 The basic idea is to modify the code after each interpretation by taking into account the most productive (i.e. with highest scores) sign functions, while replacing the less productive with new ones. Figure 1 shows a simple algorithm. First, a random world and a random code are generated parametrically by two algorithms (makeRandomWorld and makeRandomCode): these functions work only at initialisation step (hence the dotted lines), that is, at the beginning of the process. Then, the makeInterpretation function takes code and world in input, computes an interpretation and passes it to updateCode. The latter retains the best performing *n* sign functions (by taking into account the relative specificity in relation to matches), while replacing the *dim* - *n* worst performing ones, where *dim* is the number of sign functions in code and *n* is a settable parameter. The updated code is thus fed back recursively into makeInterpretation and the interpretation process starts back, world being immutable.

Figure 1.



An adaptive algorithm, first design.

62 Thus, given a certain world, the interpreter is initially provided with a randomly generated code and performs an interpretation: the best sign functions in terms of interpretation performance are kept in the code while the worst are removed, and replaced by new random ones. Two points are relevant in this hypothesis:

i. the interpreter starts with a random code, that is, in pitch dark. It is not provided with any clue in creating expressions that match the world. Similarly, new sign functions are again created randomly, in a trial and error process;

ii. code has a fixed size, it can improve its performance but without growing in terms of number of sign functions;

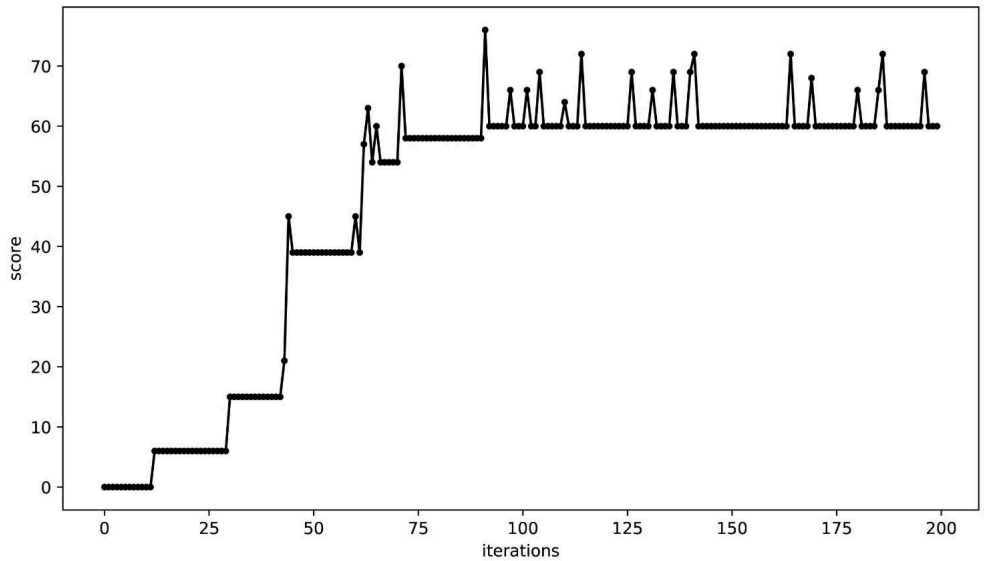
63 Both points are based on fundamental biological facts. First, there is no teleological guidance relating the world to its interpretation by the interpreter. Second, the latter has to store the code somewhere: resources are always limited, so it is the interpreter's memory. Hence, the need to replace rather than to add, assuming that we are already operating at the interpreter's full memory capacity.

64 An example of a random code (effectively generated in the implementation) with *dim* = 3 is the following:

```
{... : ..., pzv : .h., ..j : r..}
```

65 Interestingly enough, it opens with the "anythingGoes" function. Such a sign function may be generated, but—its score being = 0—it should be soon deleted in the dynamic process assigning credits.

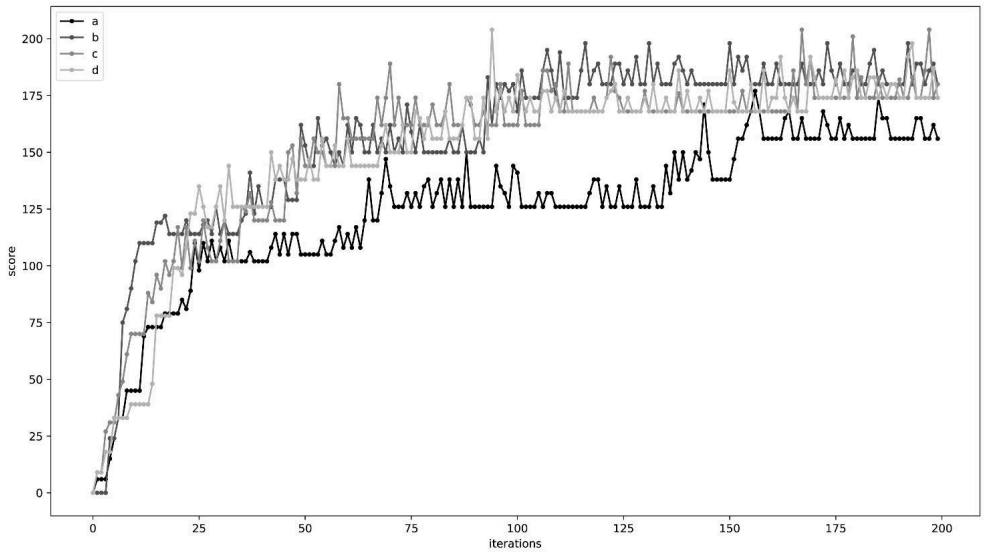
Figure 2



Performance of the adaptive code algorithm for a fixed world over 200 iterations.

66 Figure 2 shows the overall score *is* against the number of iterated interpretations. In this case, the process starts with a random world and a random code made up of 4 sign functions, with one sign function being replaced at each iteration, for a total of 200 iterations. A typical behaviour of the algorithm is clearly visible: the score starts from a variable number of scores = 0, then it grows almost linearly, finally it oscillates around various plateaus, eventually it raises to a new one. Peaks show a feature related to replacement working at full capacity: the worst sign functions are always replaced, but there is no guarantee that new ones will perform better, at least immediately. In fact, it can be seen that the score decreases after the peak, while typically increasing in the long run (but no more at the end). Figure 3 shows a further example, depicting 200 iterations for four runs (*a*...*d*) with the same world and the same initial code, in this case made up of 11 random sign functions. This means that the same code at initialisation has been processed by evaluating its performance in four different runs, and, for each run, at each iteration the worst 3 sign functions have been randomly replaced. The four runs show an overall similar behaviour, and they tend to align towards the end, with *a* scoring a worse performance for most iterations. This alignment does not mean that the final codes are the same: rather interestingly, while being mostly very different, they still show some similarities, all moving toward a better interpretation of the world. The final codes (after 200 iterations) for the four runs *a*...*d* are reported in Figure 4, where left side is the *Ef*, and right side a list (between square brackets) containing the *Cf* and the specificity score (as an example, in *a*1 *Ef* contains a dontcare, *Cf* does not, so specificity = 2 *x* 3 = 6). Sign function *a*1 has also been developed in *b*, as *b*5. The same happens for *b*3 and *c*5. Other sign functions are very similar, like *a*3 and *d*6, or *b*1 and *d*1, as they match parts of contiguous chunks of the world (the latter being a unique, single string).

Figure 3



Four runs from the same code over 200 iterations with a fixed world.

67 The system looks adaptive as it keeps on generating better scores. We can say that it learns by trials and errors, collecting the best results, reaching different codes that represent with the same performance the world and that share some similarities. In the model, a simple interpreter is provided with an interpretation algorithm plus a memory: something not far from the characterisation of many living beings. The interpreter learns by storing good matches, introducing new sign functions while discarding less performing ones. Loosing sign function is also something that sounds not inappropriate both in cognitive terms and in cultural ones. The renewal of culture, be it at the level of the single interpreter or the society, implies (if not requires) an exercise in oblivion. Indeed, many possible variations on the model can be easily added. As an example, rather than discarding a sign function for each iteration, one can think of discarding a sign function only if its score = 0. So each iteration adds a new rule, which is discarded in the following iteration if its score = 0. This would lead to a progressive increase in the dimension of the code and could be an interesting characterisation of what typically happens in cultures. Colin Renfrew has discussed the grow in cognitive/cultural richness caused by the Neolithic revolution: more material richness has implied more symbolic richness, that has prompted a "tectonic" (i.e. constructive) cognitive phase (Renfrew 2007).

Figure 4

	a	b	c	d
1	vh. : [dji , 6]	xh. : [aky , 6]	d.u : [ocd , 6]	.xh : [tvb , 6]
2	p.k : [pak , 6]	.fh : [dto , 6]	vb. : [uqk , 6]	aw. : [lmo , 6]
3	.ct : [iqw , 6]	.gc : [tzq , 6]	gp. : [esz , 6]	ul. : [oxq , 6]
4	.km : [var , 6]	hx. : [net , 6]	bc. : [jfc , 6]	o.f : [awt , 6]
5	.os : [lnu , 6]	vh. : [anj , 6]	.gc : [bef , 6]	cv. : [nec , 6]
6	.rk : [fyt , 6]	ya. : [jsd , 6]	.kd : [zji , 6]	ct. : [liq , 6]
7	.ca : [fyc , 6]	vs. : [ucf , 6]	u.j : [jpz , 6]	.od : [ngl , 6]
8	sr. : [vzg , 6]	s.p : [yqc , 6]	.ms : [mex , 6]	ol. : [ijp , 6]
9	cya : [o.x , 6]	obd : [iad , 9]	pux : [zjx , 9]	.yt : [.ei , 4]
10	clg : [jnd , 9]	azh : [xck , 9]	jcl : [fwb , 9]	hbx : [.oy , 6]
11	iry : [.ua , 6]	kzl : [foz , 9]	gne : [ymr , 9]	czd : [uac , 9]

Codes a…d after 200 iterations.

68 The main issue in this evolutionary modelisation is that it does not take into account the generated content. This is contradictory in terms of assumptions: if content is simply discarded, like residing in a totally different space from expression, then the monistic approach is turned into a dualism, in which the interpreter and the world stay on opposite sides (moreover: where does the generated content go?). It is also contradictory in relation to the model itself: the interpretation score, that is, the parameter leading the whole process, is computed by taking into account the content (as a factor in specificity) but then the same content is not used at all.

5. On content

69 Content should thus be included into the previous bare-bone mechanism in which sign functions are constructed by summing random guesses and storing inductively good results. The main point at stake here is that the model provides a characterisation in terms of E/C, however it does not say anything about content per se. It just deals with expression matching. To speak with Hjelmslev (1961), such a semiotics is monoplanar, content here being simply a sort of appendage of expression. Our starting assumption was that expression and content are made literally of the same stuff. As properly there is no difference between E and C, there is no space in this simple model for considering the latter as a mental/cognitive item (as it mostly happens in linguistic studies). On the other side, here content is something related to a fact in the world: simply, the reply to an expression. In biosemiotics (Emmeche & Kull 2011, but one could also think about Peirce), content can be seen as a response to something that triggers it (the expression). Talking with a friend, I hear her/him speaking, and I reply by speaking. On the other side, expressions are material, but they can be literally made of the stuff that dreams are made of: while dreaming, expressions and contents are evidently present in a single semiotic context that requires a complex interpretative work. But what to do with content in our model? Eco comes again to help with a very famous suggestion, dealing with semiotics as a discipline. As an interpretative practice, the latter is not to be thought of as the exploration of the sea, "where a ship's wake disappears as soon as it has passed", rather as the exploration of a forest: this means that, technically speaking, the interpretation leaves "footprints" and "cart-trails" that "modify the explored landscape" (Eco 1976, p. 29). This does not apply only to the epistemological level. In Eco (1976) the famous Model Q (owing its name to an early essay on semantic networks by Quillian) is meant as a regulative model of the Encyclopaedia, and is structured like a complex graph connecting cultural entries with multiple inputs and outputs. The difference between E and C is simply positional, that is, in graph parlance, it differentiates the starting vertex from the ending vertex in a direct graph. Eco strongly suggests that every interpretation modifies the graph. How this formally happens is not specified but the suggestion is relevant (cf. Valle 2017b). It is also apparent that:

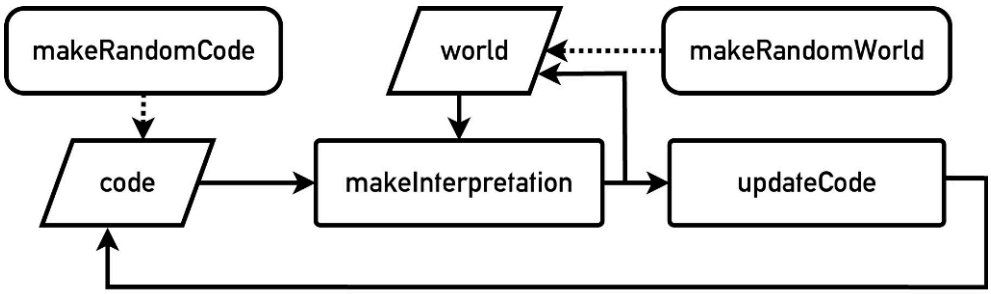
- we live in a semiotic world, where objects and their interpretations are intertwined;
- semiotic behaviour has evidently material effects. One may think today of Anthropocene. But the "human imprint" is far more distant in the past, at least since the discovery of fire. More generally, all living beings shape the world constantly with their behaviour (from cyanobacteria responsible for the Great Oxidation Event to "engineering animals", like elephants, cf. Chelazzi 2013).

70 To sum up: the only difference between E and C lies in their respective positions, and each interpretation (E → C) is an act or a fact in the world. Interpretation, rather than a passive operation, coincides with the active movement of sign production. A possible solution to implement these features in our model is to plunge back content into the world. Let us consider a sign function like:

```
ahb : qux
```

71 When *Ef* matches ahb, *Cf* is available as a generating template. It is the response of the interpreter to E, and a new string qux is generated. Interpretation modifies the world. Properly, the world is recreated (partially) by the interpreter, in the same way in which the interpreter has to adequate to the world. Phenomenologically, this semiotic intertwining is a "structural coupling" between the interpreter and the world (Basso 2002). So, in our case, every matched expression generates a new content that can be added (in various ways) to the world. As Goodman observes, "worlds are made *from other worlds*" (Goodman 1978, p. 6, italics in the text). Figure 5 represents an updated design with respect to Figure 1, where interpretation is used both to assign credit to sign functions (as before) *and* to generate content to be added to the world. In this design, while the code has a fixed size, the world gets "tectonically" larger at each interpretation.

Figure 5



Adaptive algorithm with world update.

72 Long story short: such a simply revised design, as (interestingly) shown by implementation, simply does not work. We may get high scores or at the same time 0 score. So, there is properly no learning, just contingency. The main flaw is that each successful interpretation (i.e. expression matching based on *Ef*) throws into the world new chunks (contents, based on *Cf*) that are unrelated to the code's *Ef*s, thus going unmatched at the next iteration. The production of content (i.e. its injection in the world) worsens the interpreter's situation by decreasing the interpretation score. In short, an adaptive behaviour in the iteration *n* results in a (variably large) modified environment in the iteration *n+1* which is unrelated to the same adaptation. Actually, we are polluting the world by adding unmatched character chunks. While this is not so unsound (cf. again Anthropocene), indeed it does not favour adaptation. It is almost like beginning from scratch, like in initialisation phase, at each iteration.

6. Motivation (partly) vindicated

73 If contents are plunged into the world, in order to match them, opportune sign functions need to be in the code if the goal is to increase the interpretation score. Given that, a possibility could be to intrinsically generate matching while updating the code. In this case, new types *Ef*/*Cf* should not be random, rather they should be generated following existing ones. This is not a totally ad hoc solution as it may seems at first glance. At the end, it deals with the description of unknown things by means of known ones. Random association between *Cf* and *Ef* seems to adequate the notion of arbitrariety as a classic Saussurean foundation for sign definition. It could be observed that such a foundation is probably biased by natural language (cf. Maran 2020): the typology of modes of sign production shows that actually a vast amount of sign functions are generated by motivation, i.e. by ostension and imprint (invention). In presence of something not conventionated (expression) the only resource to interpret it is to find something else (content) related in some way to it, that is, to generate content by motivation. Why abc should be operationally associated to something totally else like def? We know two ways to relate the two planes: by reproducing some part of the expression in the content (ostension) and by inferring a rule allowing to map expression to content (imprint/invention).

74 While in the model it has proven difficult to formalize imprint, ostension is not complicated. If we have abc as *Ef*, then by putting some characters with their respective positions into *Cf* we have an ostension. But, if an ostensive sign function is:

```
abc : abd
```

then injecting abd back into the world still does not favour a future match. Here, dontcares are the key elements. Let us consider:

```
abc : ab.
```

75 Injected content may be abc, and can be matched by the same *Ef*. As usual, the more dots, the more matching (and, conversely, the less interpretation score). Ostension thus may result in motivation that can be described as *Ef*/*Cf intra-connectivity*.

76 Another form of connectivity can be obtained by associating to a new (random) *Ef* an *Ef* from another sign function (becoming its *Cf*). If the code contains:

```
abc : def
```

then we could add a new sign function:

```
xyz : abc
```

77 Collected in the same code, the second sign function, when matching xyz, injects into the world abc, that is matched (at the next iteration) by the first one, generating def. To reach a content (abc) from an expression (xyz), then to move on to another content (def), as (technically) expressed by the first, and so on, is evidently a very well known cognitive and cultural procedure, and has been the model of various semiotic constructions (from connotation chains to unlimited semiosis, from semiosphere to encyclopaedia). In this case, there is *extra-connectivity*, that gives an account of connotation chains.

78 Connectivity (be it intra- or extra-) can be traced back to a slippery, yet pivotal, semiotic notion: analogy. In the context of a character-based modelisation, analogy may be seen as the generation of content that is adequate to expression in that it allows to find relations inside a sign function and outside it, in the code. Analogy, as

formalised by connectivity, represents the fact that—if there is not an already conventionated E/C association – then a "resemblance" among types is better than a blind guess.

7. A final model

79 For sake of simplicity, in the final model that I will discuss the world is slightly modified, as it is built by randomly assembled triplets, acting as atomic building blocks: thus, the world is not a sequence of characters but of 3-character chunks. Newly generated signs functions are assembled by analogy. In relation to extra-connectivity, each new sign function is obtained by coupling an expression type from the *Cf* set with a content type taken from the *Ef* one. In both types, each character can be replaced stochastically (i.e. according to a certain probability) by a dontcare. This general behaviour can be called "strict analogy".[8] In relation to intra-connectivity, the simple presence of dontcares allows a variable ostensive degree. Open types have lower scores but match more. As an example, if this is a code:

```
{.hc : m.j, wi. : l.x, acw : pdt, tm. : far}
```

a new sign function generated by strict analogy with a dontcare replacement probability = 0, could be:

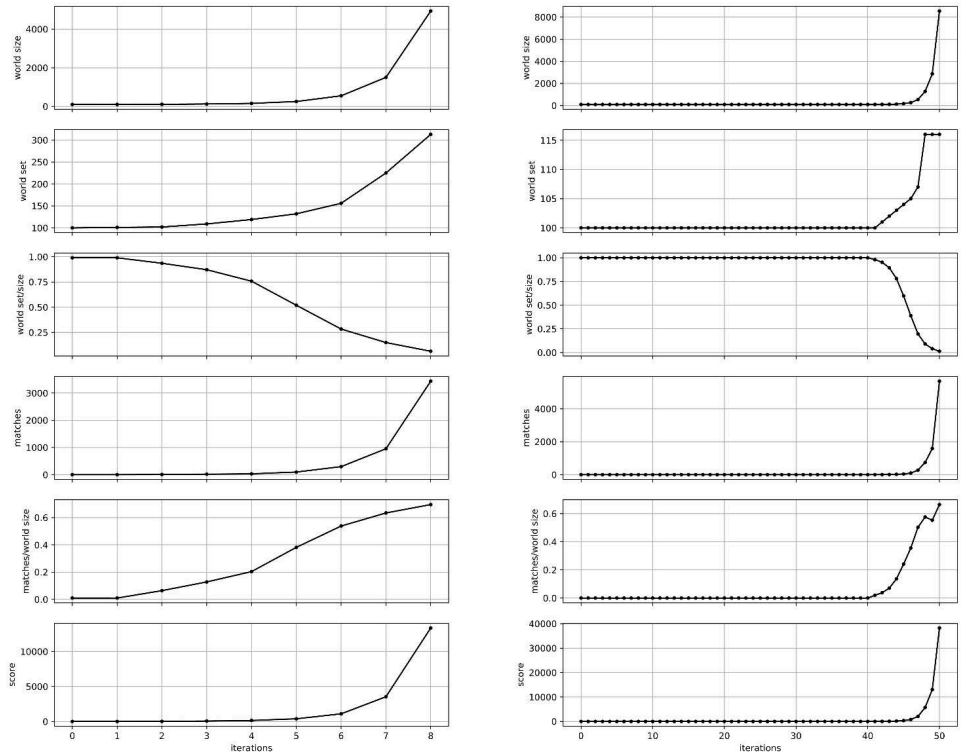
```
far : acw
```

while increasing the dontcare replacement probability we could get slightly more open types, e.g.:

```
f.r : .cw
```

80 Generated content (a triplet) is fed into the world, that keeps growing at each iteration: this assumes that semiotisation is properly a form of worldmaking, acting both by expanding the world and making it more interpretable for the interpreter.

Figure 6.



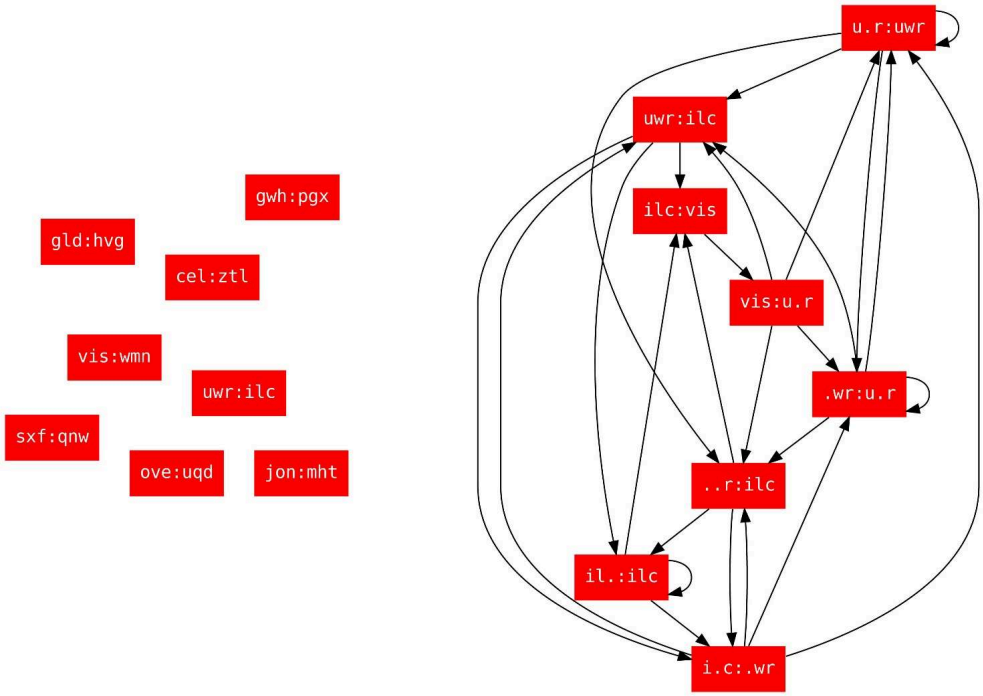
Dimensions of the adaptive process in two runs.

81 Figure 6 shows the behaviour of two runs of the implemented model starting from different random worlds and codes. In both cases, the world is made up initially of 100 triplets and the code contains 8 sign functions, with 3 replacements at each iteration. From top to bottom, world size indicates the number of triplets in the world: the latter increases if new content is generated as a consequence of matching. World set is the cardinality of the triplet set of the world: so the number of *different* triplets. World set/size is the ratio between the previous factors. Matches is the number of successful matches, i.e. how many triplets from the world are matched by code's *Ef* side. Matches/world size is a ratio indicating how much of the world is captured by the code. Finally, score is the interpretation score. All dimensions are plotted over iterations. In both cases, the generative process ends by design after the world exceeds 5,000 triplets. In case left, 8 iterations are enough to make the world grow over this value. World size starts from 100, while world set is 100 too, meaning that the random generation process resulted in 100 different triplets. While world size increases, the same happens to the number of different triplets, yet the ratio world set/size falls from 1 near to 0: this means that, even if there are more different triplets, the world contains many more occurrences (tokens) of those same triplets. In short, redundancy in the world is growing much faster then world variety. The number of matches measures the capability of the code to represent the world but also to increase the latter's dimension, as each match generates new content for the next iteration. The ratio matches/world size thus indicates how much of the world can be interpreted by the code. It starts from 0 (no matching), and a value > 1 (not reached here, but not infrequent) means that triplets in the world are statistically matched by more than one sign function: this means that at iteration *n+1* the new content to be added to the world will be greater

than the world size at iteration *n*. Figure 6 represent two extreme cases: they show the same behaviour, with curves having different slopes. Case left is almost linear, and in just 8 iterations the world size is greater than 5,000 triplets. Case right shows a long inertia in the beginning, meaning no luck in interpreting the world for many iterations. This history of tectonical failure results in 40 iterations with 0 matches, and the world consequently remains confined to the small 100-triplet one from initialisation. Then, the process quickly realigns with the behaviour shown in case left.

82 The whole dynamic process can be described by taking into account the first state (initialisation) and the last iteration from the perspective of both the code and the world, to see how they relate each other.

Figure 7.



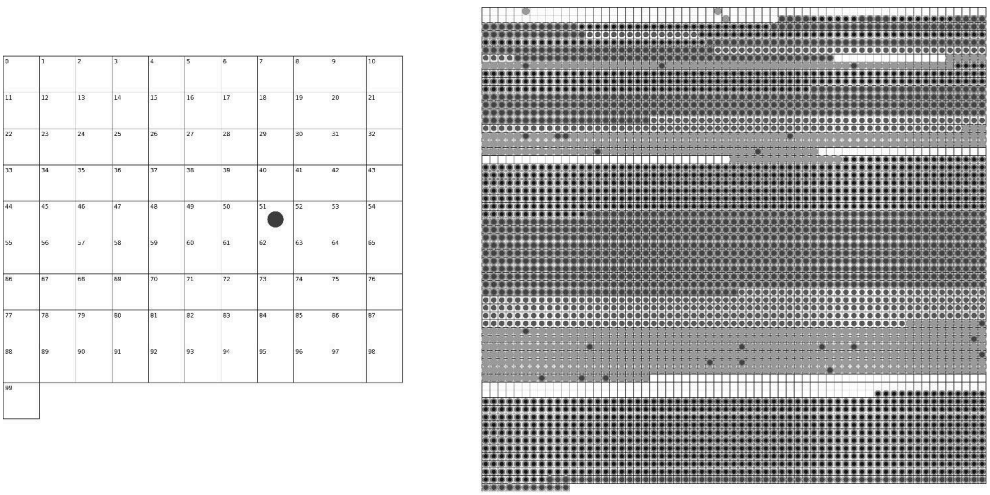
Code graphs at beginning state and after 11 iterations

83 In relation to code, connectivity provides a suggestion on how to visualise it. With a classic move, code can be represented as a graph (one may indeed think about the various models by Eco), in which vertices represent sign functions while directed edges link two sign functions if and only if the content of the starting function is matched by the expression of the ending one. In this sense, the graph represents the code in terms of connotation chains. Figures 7 and 8 show a case with a code including 8 sign functions and a world from 100 to maximum 4,000 triplets. Figure 7, left, shows the code graph for initialisation (state 0): being randomly generated, there are no connections at all (even if they might occur). Figure 7, right, shows the code graph for iteration 11. Now the updated graph is densely connected: in this particular case, every newly generated content will thus be matched in the next iteration by at least one, and typically by more, sign functions in the code. This can be verified by looking at the right side of each starting vertex and the left side of each ending one. The code also

includes three open identity functions (i.e. including dontcares), as shown by looping arrows. The number of sign functions is kept constant (8).

84 The situation can be explored symmetrically by looking at the world side. Figure 8 shows a visualisation of the world in the same two states (0 and 11) of Figure 7. The linear sequence of triplets making up the world is folded into a square space, from top left to bottom right, for sake of readability. In Figure 8, left, each triplet is shown as an indexed square: at initialisation, the world is by construction made up by 100 triplets (0-99). Each match by a sign function is shown as a circle in the triplet square. In this case, there is already a given match (more typically, for this parameter configuration there is none). In match visualisation, circle dimension and grey level are coupled (the smaller, the darker) and arbitrarily related to the sign function index in the code (that is: there are 8 different circle sizes, each associated with a grey level, one for each sign function). While state 0 is almost empty (1 match), in iteration 11 the world is made up of 3,854 triplets (not numbered for sake of readability): most triplets are matched, and many of them by more than one sign function (i.e. overlapping circles with different grey levels).

Figure 8.



World in relation to matching at initialisation state and after 11 iterations.

8. Conclusions

85 In this final adaptive model, quite abstract and preliminary, implementation details may be crucial in tuning the behaviour of the whole systems. Still, some more general semiotic features are worth underlining:

- the interpreter starts in pitch dark and tries to make sense of the world;
- sign functions exploit successfully type openness (by means of dontcares), both on E and C side: rather than an obstacle, indeterminacy (cf. Prieto's relevance) is a resource to make sense;
- the interpreter's behaviour modifies the world by making it more understandable. Interpretation is production, and vice versa;
- interpretation is properly a worldmaking activity. Operations in worldmaking indicated by Goodman (1978) are intrinsic to the model. In relation to "composition and decomposition"

and "deletion and supplementation" it can be observed that the code and the world add new information, while at least the code is subject to sign function deletion, and is constantly reorganized ("ordering"); "deformation" can be found in the role of motivation in ostension and imprint; "weighting" is encoded in credit assignment;

- redundancy keeps growing at a fast rate and proves to be a pivotal feature in meaningfulness (world interpretability);
- the interpreter is limited (it stores a limited information), the world is not (it keeps growing). Of course, as discussed, if an increase in code dimension could model the increase in cultural complexity, on the other side a constraint on dimension is ecologically required. Conversely, the world too drains resources from a reservoir that should be considered ecologically finite. Yet, in the model there is a strong asymmetry between these two poles. The assumption is that, first, the interpreter is a part of the world, and, second, its activity is in some sense faster than overall world change (hence the interpreter has been immediately considered at full capacity while the world can expand).

86 Many questions are triggered by such a simple model. As an example, shall we think of introducing random perturbations in the world? How much robust would the model be in relation to these? Conversely, should we introduce random perturbations also in code (cf. mutations in genetics)? Shall we assume—ecologically—the world to be limited, so to "loose pieces", maybe older triplets, even if at a slower rate than the iteration one? And what happens if we start with more than one interpreter in a parallel fashion, each one producing new contents to be added to the world (i.e. by running the same interpretation procedure but with different codes): would their codes align?

87 Finally, as a methodological note, all the previous observations (regardless of their usefulness) have been prompted by working interactively in a formal symbolic environment (i.e. programming on a computer). This has proven instrumental in verifying the actual behaviour resulting from theoretical principles, failures included.

BIBLIOGRAPHY

BIGGS, Norman L. (2008), *Codes: An Introduction to Information Communication and Cryptography*, London: Springer.

BASSO, Pierluigi (2002) "Sul percetto tracciato e sulle tracce di una coimplicazione. Estetica e semioticadell'esperienza", *Rivista di estetica*, 42 (21), pp. 3-23.

CHELAZZI, Guido (2013), *L'impronta originale. Storia naturale della colpa biologica*, Torino: Einaudi.

ECO, Umberto (1962), *Opera aperta*, Milano: Bompiani; new. ed. 2009.

ECO, Umberto (1971), *Le forme del contenuto*, Milano: Bompiani.

ECO, Umberto (1976,) *A theory of semiotics*, Bloomington: Indiana University Press.

ECO, Umberto (1984a), *Semiotica e filosofia del linguaggio*, Torino: Einaudi.

ECO, Umberto (1984b), *The Role of the Reader*, Bloomington: Indiana University Press.

ECO, Umberto (1997), *Kant e l'ornitorinco*, Milano: Bompiani (English transl. *Kant and the platypus*, New York and San Diego: Harcourt Brace, 2000).

EMMECHE, Claus & KULL, Kalevi (eds., 2011), *Towards a Semiotic Biology. Life is the action of signs*, London: Imperial Collegge Press.

GOODMAN, Nelson (1968), *Languages of Art*, Indianapolis: The Bobbs-Merrill Company.

GOODMAN, Nelson (1978), *Ways of Worldmaking*, Indianapolis: Hackett.

HAREL, Idit & PAPERT, Seymour (eds., 1991), *Constructionism*, Norwood: Ablex Pusblishing.

HJELMSLEV, Louis (1961), *Prolegomena to a Theory of Language*, Madison: University of Wisconsin Press.

HOLLAND, John H. (1995), *Hidden Order*, Reading: Helix Books.

JAKOBSON, Roman & HALLE, Moris (1956), *Fundamentals of Language*, The Hague: Mouton.

KNUTH, Donald E. (1984), "Literate Programming", in *The Computer Journal*, 27, 2, pp. 97-111.

MARAN, Timo (2020), *Ecosemiotics. The Study of Signs in Changing Ecologies*, Cambridge: Cambridge UP.

PRIETO, Luis J. (1975), *Pertinence et pratique*, Paris: Minuit.

RENFREW, Colin (2007), *Prehistory: The Making of the Human Mind*, London: Weidenfeld & Nicholson.

VALLE, Andrea (2007) "Cortociruiti: modi di produzione segnica e teoria dell'enunciazione", in PAOLUCCI (ed.), *Studi di semiotica interpretativa*, Milano: Bompiani, pp. 349-424.

VALLE, Andrea (2017a) "Modes of Sign Production", in BEARDSWORTH & AUXIER (ed.), *The Philosophy of Umberto Eco*, Chicago: Open Court, pp. 279-304.

VALLE, Andrea (2017b) "Paths – On the Formation of the Subject in *A Theory of Semiotics*", in THELLEFSEN & SØRENSEN (eds.), *Umberto Eco in His Own Words*, Boston/Berlin: De Gruyter, pp. 93-103.

VALLE, Andrea & MAZZEI, Alessandro (2017), "Sapir–Whorf vs Boas–Jakobson. Enunciation and the Semiotics of Programming Languages" in *Lexia*, 27-28 ("Aspettualità"), pp. 505-525.

NOTES

1. A neglected essay by Eco, originally in *Le forme del contenuto* (1971), reprinted in the 2009 Italian edition of *Opera aperta* (Eco 1962). An English version is available in *The Role of The Reader* (Eco 1984b). In 2009 Eco still considers this essay a major work (1962, p. VIII).
2. https://www.python.org.
3. That is, "a plain language description of the steps in an algorithm or another system. Pseudocode often uses structural conventions of a normal programming language, but is intended for human reading rather than machine reading" (https://en.wikipedia.org/wiki/Pseudocode).
4. The code is publicly available here: https://github.com/vanderaalle/semioMod.
5. en.wikipedia.org/wiki/ASCII.
6. Ostension is here described as a projection between expression and content. Goodman (1978) describes the quality of a sample in terms of "fairness or projectibility, [that,] rather than requiring or guaranteeing agreement between the projection made and an actual feature of the

whole or of further samples, depends upon conformity to good practice in interpreting samples" (1978, p. 135).

7. One could say that this variable ostension degree component is $n_{dontcare}/lenght_{stk}$, where *n* is the number of dontcare in a type and *length* the cardinality of the alphabet *stk*.

8. In *relaxed analogy*, *Ef* and *Cf* can be taken from the set *f* resulting from the union of sets *Ef* and *Cf*. In this case, properly new sign functions do not immediately match existing ones (left-side chaining, like in connotation), rather they increase the overall connectivity of the code graph (like the interpreter is creating "internally" analogies). From preliminary results, the two notions seem to lead to similar results. So, the model presented here sticks to strict analogy.

ABSTRACTS

The article presents a computationally-oriented framework based on some general semiotic concepts from Eco's theory of sign production. The paper proposes a formalization of Umberto Eco's typology by taking into account various abstract models introduced by the Italian semiotician. These are complemented by observations on adaptive systems as developed by John Holland. Following a constructionist methodology, the resulting string-based formalization is implemented computationally, leading to an abstract, but operating, dynamic model of semiosis. Design options and resulting outputs are discussed in the light of semiotic theory.

L'article s'appuie sur des concepts sémiotiques généraux formulés dans la théorie de la production de signes d'Umberto Eco, afin de proposer une perspective computationnelle de la signification. L'article propose une formalisation de la typologie d'Eco en prenant en compte divers modèles abstraits introduits par le sémioticien italien. Ceux-ci sont complétés par des observations sur les systèmes adaptatifs tels que développés par John Holland. Suivant une méthodologie constructionniste, la formalisation résultante basée sur les chaînes de caractères est implémentée de manière computationnelle, menant à un modèle dynamique abstrait, mais opérationnel, de la sémiose. Les options de conception et les résultats qui en découlent sont discutés à la lumière de la théorie sémiotique.

INDEX

Mots-clés: sémiose, sémiotique, modélisation, épistémologie, méthodologie
Keywords: semiosis, semiotics, modelisation, modelling, epistemology, methodology

AUTHOR

ANDREA VALLE

Università di Torino
Email: andrea.valle[at]unito.it