AN INTERSEMIOTIC TRANSLATION OF NORMATIVE UTTERANCES TO MACHINE LANGUAGE

Andrea Addis[1] and Olimpia Giuliana Loddo[2]

[1]Infora, viale Elmas, Cagliari, Italy
[2]Department of Law, University of Cagliari, Cagliari, Italy

ABSTRACT

Programming Languages (PL) effectively performs an intersemiotic translation from a natural language to machine language. PL comprises a set of instructions to implement algorithms, i.e., to perform (computational) tasks. Similarly to Normative Languages (NoL), PLs are formal languages that can perform both regulative and constitutive functions. The paper presents the first results of interdisciplinary research aimed at highlighting the similarities between NoL (social sciences) and PL (computer science) through everyday life examples, exploiting Object-Oriented Programming Language tools and an Internet of Things (IoT) system as a case study. Given the pandemic emergency, the urge to move part of our social life to the digital world arose, together with the need to effectively transpose regulative rules and constitutive rules through different strategies for translating a normative utterance expressed in natural language.

KEYWORDS

Programming languages, Normative languages, Constitutive rules, Regulative rules.

1. INTRODUCTION

According to Jakobson, translation is a form of interpretation, where interpretation is the "transposition of a sign into alternative signs having the same meaning". He also claimed that there are three ways one can interpret the verbal sign; "it can be translated into other signs of the same language, into another language, or into another nonverbal system of symbols" [1, p. 60]. He called them respectively intralingual translation or rewording, interlingual translation or translation proper, and intersemiotic translation or transmutation. The intersemiotic translation does not involve merely different languages, but it is the interpretation of a sign part of a semiotic system with another sign part of a different semiotic system. More precisely, intersemiotic translation is generally understood to mean the transfer of verbal texts into other systems of signification, such as visual, oral, gestural. The definition proposed by Jakobson is clear and schematic, but it also oversimplifies the mechanism that leads to the final product of the intersemiotic translation. There is rarely a direct relation "sign to sign" from a code to another code. In fact, the source sign and the target sign belong respectively to extremely different semiotic systems, making the task uneasy for the intersemiotic translator to find strategies that preserve the most relevant part of the original meaning. The impossibility to cross a linear path while performing a translation is verifiable, even interpreting different natural languages or dialects, notwithstanding their structural similarities[1]. The translation of a normative meaning expressed in a natural language into a bitcode is an effective and exemplifying way to illustrate

[1] E.g., the hypothesis in modern linguistics of the existence of a certain set of structural rules that are innate to humans and independent of sensory experience [2].

DOI: 10.5121/ijwest.2022.13101

the mechanism of intersemiotic translation because it highlights the needs to diverge from a unique set of encoding algorithms, i.e., the requirement of a bridge language, in our case a "High-Level" Programming Language. A High-Level PL allows to convey the concept from natural language to a human like representation with a sufficient abstraction from the details of the computer hardware (from which the adjective high-level), allowing to cross the pure semantical translation. Programming languages can explicitly represent deontic modalities (for example, prohibition and obligation through state machines).

The transposition of numerous social and legal events in the IT field has made it necessary to focus on the correspondence between what happens in the legal field and what can happen in the digital world. Can a social reality be digitized without taking into account some essential characteristics? Legally relevant forms of interaction, such as digital contracts and smart contracts, are already taking place. For this reason, it is important to analyze the different translation techniques into a programming language that considers the categories (such as constitutive and regulative rules) identified by the social ontology. As far as we studied, however, among the many examples of inter-semiotic translation examined by semiotics and linguistics scholars, PL is rarely mentioned in the translation of natural languages into machine languages. For the aims of the philosophy of normative language, a fascinating element of the translation of a deontic proposition expressed in natural language into a programming language consists in the fact that the programming language is essentially normative. Normally, the main purpose of the programming language is not to let people know but to get things done. Programming language has not a descriptive function but a regulative and constitutive function. The paper will focus on the feasibility of intersemiotic translation between normative language and digital language. To do that, we will study how to code a set of rules through a programming language. We will show several case studies and perform some experiments exploiting both the game of chess which is a phenomenon deeply analyzed both in computer science and in the philosophy of normative language[2] and DomoBuilder [3], an Internet of Things (IoT) system devoted to building complex home domotics environments combining simple physical devices and intuitive sets of rules. The paper is organized as follows: Section 2 will illustrate the reasons for choosing the bridge language and tools to analyze and show how to code norms, section 3 will explore the process to translate regulative and constitutive rules. Finally, section 4 will draw the conclusions.

2. PROGRAMMING LANGUAGES AS TOOLS FOR CODING NORMS

Programming languages (PL) are a subclass of formal languages comprising a set of instructions used in computer programming to implement algorithms, i.e., to elaborate information (e.g., input data) and produce output (e.g., analysis data, new information, and/or perform a task). PL can be categorized in several ways that set side by side their different strategies for implementing algorithms, as for instance, low-level languages that bind the developer to write code that matches the internal representation of the machine architecture, and high-level languages that allow expressing the code through a more abstract and human-like representation of the algorithm. Another classification compares declarative programming languages and imperative programming languages[3].

This classification shows how computer scientists and software engineers tend to add more layers of semantic abstraction to write more readable and maintainable code, thus releasing the developers from specifying how the program should achieve the result. In fact, Imperative

[2]The rules of chess are among the most exploited examples of constitutive rules.

[3] Please note that these two definitions carry an intrinsically different meaning from the one used in the investigation on normative language.

Programming Languages focus on the use of imperative utterances to change a program's state and describe how it operates in a way similar to the expression of commands in natural language. At the same time, Declarative Programming Languages allows the developer to describe what the program should accomplish, focusing on its target [4]. We will explore in this context the imperative paradigm because it is particularly popular and mirrors the hardware implementation of almost all computers. Among the imperative languages sub-classes, the Object-Oriented Programming Language (OOP) gained rapid growth in the '80s [5], its concept built on "objects" abstracted through "classes", (a) provide natural support for software modelling of real-world objects or the abstract model to be reproduced; (b) allows easier maintenance of large projects; (c) grants more organized code in the form of classes and concepts, favouring modularity and code reuse. Within this paper, we will exploit Java, a general-purpose OOP language designed to have as few hardware implementation dependencies as possible [6]. In Java, we will analyze the path to set a norm through a programming language and check if the logical structure of the norm affects the programming code.

3. Regulative Rules vs Constitutive Rules

John Searle [7] points out that not every rule has the same logical structure and draws the famous distinction between regulative and constitutive rules.

According to Searle, regulative rules "regulate antecedently or independently existing forms of behaviour; for example, many rules of etiquette regulate interpersonal relationships which exist independently of the rules" [7, p. 34]. There is not any ontological relation between the rule and the form of behaviour that the rule regulates. The regulative rule has no impact on the concept of what it regulates or its individual instances. Are examples of regulative rules:

(i) "It is prohibited to smoke."
(ii) "It is obligatory to wear a mask."
(iii) "It is prohibited to walk on the grass."

Usually, regulative rules establish implicitly or explicitly when they are binding. Regulative rules characteristically have the form or can be comfortably paraphrased in the form "Do X" or "If Y do X".

So it is possible to make the mentioned rules explicit as follows:

(i) "If you are in a public library, don't smoke."
(ii) "If you are in the park, do not walk on the grass."
(iii) "If you are in a public space, wear a mask."

Indeed, wearing a mask is a brute fact, such as walking, eating, swimming, turning on the light. (Regulative) rules can regulate all these behaviours; however, their existence is independent of the existence of any rule, and they cannot be valid or invalid.

According to Searle, unlike regulative rules, "Constitutive rules do not merely regulate, they create or define new forms of behaviour" [7]. Constitutive rules establish a relationship with what they regulate that is different from the relation between regulative rule and regulated behavior. There is an ontological connection between the constitutive rule and what the rule regulates.

In the following paragraphs, we will try to check how to translate regulative rules and constitutive rules in Java. To better understand these aspects, we will also exploit DomoBuilder, an Internet of Things (IoT) system devoted to building complex home domotics environments that allow depicting an operative environment composed of real entities: objects (devices) and actors (users). Every object is a "device" that only needs to describe itself within the environment through the PME paradigm. It shares, in fact, a set of (P) properties deemed relevant for the users, (M) methods allowing to change the state of its properties or perform tasks, and (E) events allowing the system to be asynchronously informed about occurring changes of states. Let us consider a trivial, though clarifying, example to highlight this concept: A light bulb.

```
public class LightBulb extends Device {                                     (i)

       public LightBulb() throws DeviceException {

       set("description", "This is a lamp,
       it can be switched on or off");                                      (ii)

       putProperty(new DeviceProperty("state",                              (iii)
              "Shows the status of the Light Bulb",
              String.class.getName(),
              "on|off", "off"));


       putMethod(new DeviceMethod("set",
              "Set the status of the LightBulb, i.e., (ON|OFF)",
              String.class.getName(), "ON|OFF"));                           (iv)
              [...]
       @Override
              public void onMethod(String name, String value) {
              [...]                                                          (v)
```

This is all we need to know to describe and make available a new device. Let's explain what is happening.

(i) LightBulb inherits all the properties and constraints of a generic DomoBuilder device. According to the OOP paradigm of Inheritance, the LightBulb will derive most of its features "extending" the generic DomoBuilder Device (e.g., the capability to describe itself).

(ii) We set a property of the LightBulb common to all devices: the description. This will make the description of its features available to the users or other devices.

(iii) We create a new property for light bulbs; in this case, we can't directly access it like the description already described for any generic device; we need first to define it. We make explicit its type and the allowed values.

(iv) In a similar fashion, we create a new method, an action that the device can execute. In this case, the LightBulb can turn on the light. This is done according to the encapsulation and obfuscation paradigms of OOP, i.e., to conceal the mechanism of the internal code.

(v) Here the developers implement the internals of the methods. In this case, an utterance will simply be sent by serial communication through a relays system to an electrical switch.

The ability to mash up heterogeneous devices and combine their functionalities gives rise to complex systems enriching them with new powerful features and, eventually, brand new components underlying new concepts. This is possible through an additional constituent of the paradigm: (R) rules. Rules allow encoding high-level behaviours with trivial devices, allowing complex interactions between components and users.

In the next paragraph, we will show how simple it is to model such an architectural paradigm given a PME(R) substrate.

3.1. Programming/Translating a Regulative Rule

A machine can operate in a non-digital environment and control other technological devices to fulfil regulative rules. This is possible if the programmer sets the machine to make possible the perception of the external environment and the other devices. To do so, the programmer has to create new classes of objects through a schema very similar to Searle's formal description of constitutive rules. According to Searle, constitutive rules can be formalized as follows,

```
X counts as Y in the context C
```

Where X is a brute fact, and Y is an institutional fact. However, a programmer who has to teach a machine to follow the rules in a non-digital environment must create digital (symbolic) classes that correspond to the material objects that the machine has to control (e.g. a lamp, an alarm clock, a washing machine).

In this sense, the programmer follows a schema that can be described as the reverse of the formula of constitutive rules. Where X is a digital object, Y is a material object with specific functions, and C is a software development context.

According to Searle, "regulative rules characteristically have the form or can be comfortably paraphrased in the form `"Do X" or "If Y do X"` [7, p. 34].

Regulative rules aim to be fulfilled, so they are addressed to a subject or a more or less wide class of subjects. Regulative rules can be addressed to machines, and, following the form "if Y do X," it is possible to ask a machine to perform an action that exists independently from me and from the machine, such as "if (conditions) then turn on the light". Where "turning the light on or off" is an action that preexists logically and chronologically this rule.

Let's exemplify this regulative rule inside a software development context. A user desires the light to turn on when s/he's entering a room. This rule is hard-coded inside the platform, i.e., in a meta-programming language fashion:

```
If a user enters the room, then turn on the light!
```

In Java language, we need to codify the meaning of "entering a room" given a sensor (e.g., a webcam installed in the room) and code the action of triggering a light switch. The implementation could be written as follows:

```
while (true) {
       if (webcam.movement_detected) {
              light_switch.turn_on(); } }
```

Where webcam and light switch are object instances of their corresponding classes, i.e., a representation of real devices, the code is encapsulated in an infinite control loop (`while(true)`). Nevertheless, in a real implementation context, an event-driven paradigm (asynchronous) is preferred to this strategy (synchronous). This is possible by subscribing the light switch to the event caused by the movement detection (i.e., its change of state), acting consequently by triggering an action, therefore relieving the processor from a continuous and computationally onerous check.

This is the reason why DomoBuilder implements a PME(R) paradigm. As we have seen, the Light Bulb shown above is a particularly simple device, it trivially exports its property of being on or off, and its events will be triggered correspondingly after the transition of its state from on to off and vice versa. Note that the method set we defined allows us to let users or other entities (objects, devices) in the system to toggle its state.

Similarly, for the webcam, we will have a few properties. The webcam will internally collect images corresponding to what happens in the environment. An internal algorithm will set to true the property indicating that someone is in the room (i.e., trivial algorithms of movement detection will update its properties as needed and triggering the corresponding events).

```
Public MovementDetector() throws DeviceException {

        set("description", "Checks if is there anybody

        out there");

putProperty(new DeviceProperty("movement",

        "Asserts if there is movement",

        boolean.class.getName(), "true|false", "false"));

putProperty(new DeviceProperty("presence",

        "Asserts whether somebody has been

        in this room recently",

        boolean.class.getName(), "true|false", "false"));
```

Note that implementation internals are not discussed in this context, but the philosophy of DomoBuilder is to make available very simple devices that implement very simple tasks so that under the hood, there are often very few lines of code. The system complexity rises from rules that combine behaviors.

The code to detect the movement is very simple; in fact, it just checks the camera input frame by frame, detecting differences among them and updating the status of the device if a sensibility threshold is passed.

It is possible to code a rule inside a machine establishing that at certain environmental conditions (e.g., the presence movement), the machine will perform an action (turning the light on or off) that is possible independently from any (social) regulation. In this sense, the act of turning on the light is different in kind from the act of moving the bishop during a chess game. Thanks to its architecture, DomoBuilder allows us to build a regulatory rule in a declarative form, further abstracting the imperativeness of the Java language on which it is implemented. The rule can be defined as follows:

```
<rule>

       <id>Automatic Light Example</id>

       <conditions>Webcam.movement_detected==true</conditions>

       <action>Light_Switch.toggle(ON)</action>

</rule>
```

Which expresses very clearly what is the outcome we expect from the user's interactions with the home environment[4]. Here it is clear how the presence of events that allow the asynchronous handling of data alerting the system when a change has occurred mirror the concept of logic of change [8].

During the Covid pandemic in 2020, many new regulative rules have been created, and many existing ones have been strengthened or modified[5]. The most consequences concerned online gaming, which has dramatically risen in popularity, leading to the need for increasing controls to maintain healthy and fairly competitive Internet chess online platforms. For example, improving the Artificial Intelligence (AI) of the bots (software applications that run automated tasks over the Internet) that automatically check the fairness of the players based on the deviation from their usual game style.

It is, in fact, possible to program a machine to strengthen or check if people follow regulative rules. For instance, let's think about the regulative rule "you must wear a mask". A video-scanner can notice if I wear a mask (to protect me and others from COVID19), and If I don't, it can remind me of my obligation, or lock the door until I wear it, or even call the security service. Or also, for the same aim, a scanner can check people's temperature to prevent their access if it is outside the accepted range.

In DomoBuilder we would code the rule as follows:

```
<rule>
       <id>Covid Security Door Locking</id>
       <conditions>Webcam.movement_detected==true
       AND Webcam.face_withmask_detected==true
       AND Clock.time > 0800, < 1800 (working time)
       </conditions>
       <action>Door.toggle_lock(OFF)</action>
</rule>
```

In this case, a rule is composed of a set of conditions that must be simultaneously fulfilled in order to unlock the door.

[4]Note that further conditions and temporal constraints can be implemented in the same rule.

[5] For example, during the Tata Steel Chess Tournament in 2021 a new rule allowed the organizers to change the logistics of the game tables, in some cases raising a huge controversy in some really critical game moments. See the Firouzja Controversy.

3.2. Programming/Translating a Constitutive Rule

Constitutive rules are different in kind from regulative rules because whereas regulative rules regulate preexisting behaviour, constitutive rules create what they regulate[6]. A typical example of constitutive rules in the game of chess is:

"The chessboard is composed of an 8 x 8 grid of 64 equal squares alternately light (the `white' squares) and dark (the 'black' squares)."[7]

In Java, we could code the chessboard as a matrix:

```
String[][] board = new String[8][8];
```

In this case, we represented each tile of the chessboard with an Integer Number (an integer number will, e.g., represent if a tile is occupied and by which piece), each piece may be placed only within the 8x8 grid corresponding to the instantiated matrix (an array of arrays).

Keeping in mind that chess coordinates are expressed as 0-based integers (i.e., a..h→ 0..7, and 1..8→0..7) let us implement the function that asserts whether a tile is white or not. In natural language, we have asserted that the tiles are alternately light and dark, but the concept of alternative must be made explicit to a machine through an unambiguous algorithm. One way to implement such a trivial algorithm is to check each tile through a hand-compiled list of dark boxes:

```
If ((x==0 && y==0) || (x==0 && y==2) || (x==0 && y==4)...
      || ((x==1) && (y==1)) || ... (and so on for every tile)
return true; else return false;
```

However, this solution is extremely verbose, computationally inefficient, and not scalable (if the size of the board changes, the list must be re-edited by hand).

In the same way, I would not create an algorithm to sum two numbers by writing a list of all infinite possible sums. So we resort to stratagems (the above-mentioned bridge), in this case, by simply writing:

```
Boolean is Dark(int x, int y){
return (x+y)%2==0; }
```

This function, not immediately intuitive, translates the constraints asserting that if the sum of the coordinates passed to the function (integers x and y) is even, a dark square occupies those coordinates on the grid. The concept of even is expressed as "the remainder of the division by 2 (indicated with the operator module "`%`") of the sum of the coordinates is equal ("==" operator is an equality check) to 0". Let's try, for example, to check the e4 square of the chessboard: *e4 is Dark?* Translates to: `is Dark(4,3)`; which will be computed because7 is odd, as `7%2==1`, so that the function returns false. Therefore, e4 is not a dark but a light tile.

[6] According to A.G. Conte [9] there are different kinds of constitutive rules. In fact, a constitutive rule can be a condition for the existence of what it regulates (e.g. the rules of chess) or can establish conditions for the validity of what it regulates (e.g. the signature of a contract or of a will). In this paper, we will focus on the rules that are a condition of the existence of what they regulate.

[7] Article 2.1. Laws of Chess.

As you can see, in order to translate in code a constitutive rule, we were forced to find an alternative way to represent the rule, still in near natural language but subject to the constraints of semantic correctness and computational efficiency.

From a semiotic point of view, these rules build the meaning of the specific term. For instance, an example of a constitutive rule is the following:

```
"A bishop moves any number of vacant squares diagonally as many open
squares as you like. The Bishop must remain on the same color square as
it started the game on."
```

We cannot say "this is a bishop, and it (may) move(s) diagonally" because there is no bishop before the "moves diagonally" rule.

Here also, the programmer could list the possible combinations of the moves of the chess pieces on the chessboard, but this would make the software particularly inefficient. The programmer must take into account one of the most important characteristics of the eidetic constitutive rules: the eidetic constitutive rules can not be violated. If one does not act in accordance with the constitutive rules, one can perform an act, even one with meaning, but it is in any case different from the activity whose concept is constituted by the rules[8].

From a software point of view, a way to obligate this behaviour is to check the old and the new position of the instance of a bishop on the chessboard, i.e., given a 8x8 chessboard represented with the same matrix we used earlier, we can implement the check of a legal bishop move as follows:

```
if ( (math.abs(bishop.old_y - bishop.new_y)) !=
   (math.abs(bishop.old_x - bishop.new_x)) )
     throw new InvalidPositionException();
```

Where `math.abs(...)` is a call to a math library function that returns the absolute value of the arguments passed inside the parenthesis, and the `!=` operator means "is different from". In fact, if the absolute differences (unsigned) of the x and y coordinates of the previous and current positions are not equals, the bishop didn't move diagonally. So an exception must be thrown, or, as usually happens in a software chessboard, the move is not allowed at all[9], which is translated in the interface as the impossibility to move the bishop on that target tile. This rule is part of the definition of the object called bishop.

Without this rule, the bishop would be unthinkable as bishop, and it would not be a bishop.

In addition to this, in programming languages such as Java, the creation of objects corresponding to instantiating a class, is logically analogous to a constitutive eidetic rule. This is a kind of rule that can not be violated. It would "not compile", i.e., the program could not be

[8]In the final chapter of the novel, "The Man Who Watched the Trains Go By" Georges Simenon describes an interesting exemplification of what this means. The protagonist of the novel, Popinga, is finally defeated and put in a mental institution. During a game of chess in the asylum garden, he takes the queen off the board and drops it into a cup of coffee. This episode is mentioned by the narrator to show Popinga insanity, since he is not making just an odd violation of a rule, he is out of the game like he is out of the society, he stopped playing chess when he made an impossible move (dropping the queen in a cup is, in fact, not a legal move).

[9]Please note that for sake of simplicity checks about the chessboard boundaries or blocking pieces on the bishop trajectory are omitted in this example.

translated into bitcode. It is quite evident in online chess that I can move the bishop only in accordance with the constitutive rule. If you "do not follow" these rules in a physical chessboard and move random chess pieces, you are not violating any rule; you are simply "not playing chess".

It is interesting to note that in an online game web platform, constitutive rules are usually implemented server side i.e., the backend implements the logic which will be made accessible via an Application Programming Interface (API) for display on any device that implements the interface (frontend).

Given their structure, digital systems allow the creation of constitutive rules from scratch in a very simple way. For example, many chess platforms provide small variations on conventional rules, creating new game scenarios such as the Fischer random chess, also known as Chess960, the Atomic Chess, the King Race, and so on. This happens in DomoBuilder too, where the programmer not only sets instructions to the machine but also creates new simplified semantic entities in order to produce a representation of the world understandable to the machine that has a logical form extremely similar to the one proposed by Searle for constitutive rules. Starting from these new semantic entities, a new set of regulative rules allows to obtain a new application behaviour given the same components.

4. CONCLUSIONS

To date, some programming languages that translate normative utterances (such as contractual clauses) are developed to produce immediate legal effects (e.g., a money transfer in a smart contract) and did not distinguish between violable regulative and inviolable constitutive rules. However, this distinction is part of the social reality.

We tried to outline the limits and potential of this approach, starting from extremely simple examples that are not strictly legal, but which take account of the typological diversity that exists between the norms. It will be desirable to verify the applicability of such strategies to the legal field in the future.

Norms can be expressed both in natural language and formal language. One of the debated problems in deontics consists in finding a semantic that combines constitutive rules and regulative rules. In this paper, we focused on a subclass of formal language, programming language showing that in imperative OOP programming languages, it is possible to translate both constitutive rules and regulative rules. Moreover, the combination between constitutive rules and regulative rules is not only possible, but it is sometimes the optimal solution to implement a system. Likely, an inflexible prescriptivist, a person that thinks that rules are only regulative and there is nothing in the world such as constitutive rules, would be a bad programmer. The activity of a programmer is mainly focused on rules. The programmer is indeed an emblematic example of a subject that creates to rule [10] precisely, she/he often creates new forms of action through constitutive rules. The programmer not only sets instructions to the machine but also creates a representation of the world understandable to the machine. To do so, he creates new simplified semantic entities that have a logical form extremely similar to the one proposed by Searle for constitutive rules. However, these rules operate on reality in different ways. These rules sometimes have a direct impact on an existing behaviour, but they are often a precondition for the existence of that behaviour, i.e., they establish a condition that the user or the programmer has to satisfy in order to validly perform an action. It is indeed possible to conceive a more articulate typology of (constitutive) rules, and in this sense, some interesting attempts have been made ([11];[12]; [13]; [14]; [15]). A future challenge for this research is to distinguish different forms of constitutive rules and different strategies of translation. Clearly, the programmer deals with rules that mainly regulate the interaction between man and machine. However, when they are used to creating a digital social environment (e.g., a digital platform where different subjects who are recognized by the machine as belonging to different classes interact), then their social relevance becomes evident.

ACKNOWLEDGEMENTS

This research is part of the project “Legal Acts, Images, Institutions: The Form of the Legal Act in the Era of Legal Design”, funded by Fondazione Sardegna in 2019.

REFERENCES

[1] Jakobson, R. (1971), “On linguistic aspects of translation”, in: *Selected Writings, Mouton*, New York.

[2] Cook, V. and M. Newson (1996), “Chomsky’s universal grammar: an introduction,” Blackwell Publishers, Oxford, OX, 2nd [updated] edition.

[3] Addis, A. and G. Armano (2010), “Domobuilder: A multiagent architecture for home automation,” in: A. Omicini and M. Viroli, eds., *Proceedings of the 11th WOA 2010 Workshop, Daglioggettiagliagenti*, Rimini, Italy, September 5-7, 2010, CEUR Workshop Proceedings 621, pp. 1–2.

[4] Sebesta, R.W. (2016), “Concepts of programming languages,” Pearson, Boston, eleventh edition.

[5] Black, A. P. (2013), “Object-oriented programming: Some history, and challenges for the next fifty years,” *Information and Computation* 231, pp. 3–20.

[6] Arnold, K., J. Gosling and D. Holmes (2005), “The Java programming language,” Addison Wesley Professional.

[7] Searle, J.R. (1969), “Speech Acts: An Essay in the Philosophy of Language,” Oxford University Press, Oxford.

[8] Wright von, G. H. (1963), “Norm and Action,” The Humanities Press, New York.

[9] Weinberger, O. (1970), “Die Norm als Gedanke und Realität,” *Österreichische Zeitschrift für öffentliches Rechtes* 20, pp. 203–216.

[10] Zelaniec, W. (2013), *Create to rule: studies on constitutive rules*, LED, Milano.

[11] Conte, AG (1986), “Fenomeni di fenomeni,” *Rivista internazionale di Filosofia del diritto* 63, pp. 29–57.

[12] Azzoni, G. M. (1988), “Il concetto di condizione nella tipologia delle regole,” CEDAM, Padova.

[13] Conte Conte, A. G., “Regole eidetico costitutive e regole anankastico costitutive,” in: G. Lorini and L. Passerini Glazel. Lorenzo, eds, *Filosofie della norma*, Giappichelli, Turin, 2012 pp. 107–117.

[14] Lorini, G.(2017), *Anankastico in deontica*, LED, Milano.

[15] Sun, X. and L. van der Torre (2014), “Combining constitutive and regulative norms in input/output logic,” in: F. Cariani, D. Grossi, J. Meheus and X. Parent, eds., *Deontic Logic and Normative Systems*, Springer, New York, pp. 241–257.

AUTHORS

Andrea Addis Founder, Senior Developer in Infora: a ICT startup with experience in research and innovation. Former researcher of the University of Cagliari. Research interests: Information Retrieval, Text Categorization, Multi Agent Systems

Olimpia G. Loddo is a post-doc fellow at the Department of Law, University of Cagliari. Her main topic is intersemiotic legal translation. She has been working on the normative language, phenomenology of law