Why zombies cannot write significant source code: The Knowledge Game and the art of computer programming

Gobbo, F.; Benini, M.

DOI
10.1080/0952813X.2014.940142

Publication date
2015

Document Version
Final published version

Published in
Journal of Experimental and Theoretical Artificial Intelligence

Journal of Experimental & Theoretical Artificial Intelligence, 2015
Vol. 27, No. 1, 37–50, http://dx.doi.org/10.1080/0952813X.2014.940142

Taylor & Francis
Taylor & Francis Group

Why zombies cannot write significant source code: The Knowledge Game and the art of computer programming

Federico Gobbo[a,b]* and Marco Benini[c1]

[a]*Amsterdam Center for Language and Communication (ACLC), University of Amsterdam, Spui 210, Amsterdam, The Netherlands;* [b]*Dipartimento di Studi Umanistici, University of Torino, via S. Ottavio 20, Torino, (TO), Italy;* [c]*Dipartimento di Scienze Teoriche e Applicate, University of Insubria, via Mazzini 5, Varese (VA), Italy*

(*Received 15 July 2012; accepted 15 March 2013*)

This article analyses the knowledge needed to understand a computer program within the philosophy of information. L. Floridi's method of levels of abstraction is applied to the relation between an ideal programmer and a modern computer seen together as an informational organism. The results obtained by the mental experiment known as the Knowledge Game are applied to this relation, so to explain what a programmer should know in order to program a modern computer. In particular, this analysis clearly shows why non-conscious agents have no hopes to write significant programs. Two main objections are then presented and attacked by corresponding counterobjections.

Keywords: philosophy of information; method of levels of abstraction; Knowledge Game; consciousness in programming

1. Introduction

What does it mean to *know* a computer program? In other words, what kind of knowledge does a programmer need to write the source code of a program? In the last years, one of the main contribution by L. Floridi to the epistemological side of the philosophy of information (PI) is a clear and well-defined notion of *knowledge*, which is distinct and richer than the notion of information (see Allo, 2011; Floridi, 2011a). In particular, the challenge to artificial intelligence (AI) posed by Dretske (2003) about the nature of self-conscious agents (AGs) is attacked via a mental experiment known as the Knowledge Game (KG), that explains the relation between the nature of the AG and the notion of consciousness (Bringsjord, 2011; Floridi, 2011b, ch.13).

However, the KG does not explicitly refer to computer programming, which is a key process in the philosophy of AI – at least the logic-based variety. Here, the results obtained by the KG are applied to computational informational organisms (inforgs), i.e. every inforg whose computational part is a modern Von Neumann's machine (VNM), which requires to be programmed by an AG in order to function. The goal of the argumentation is to explain why neither a conscious-less artificial AG (e.g. a robot) nor a conscious-less biological AG (e.g. Spinoza's automata or Dretske's zombies) can be real computer programmers. In particular, we explain what kind of consciousness is needed to be a computer programmer through non-trivial examples of programming techniques known in the literature.

*Corresponding author. Email: f.gobbo@uva.nl

The plan of this article follows the structure of the argument. In Section 2, definitions are provided as the necessary tools used during the argumentation. At first, it is shown what an intertwining of VNM-based machines, which defines the interpretation of modern computers, and in particular why the adjective 'modern' is important. Then, we explain how the method of levels of abstraction (LoAs) is used within a computational inforgs. Finally, the correct place of consciousness within the computational inforgs is given, after some consideration of the KG after Dretske's question.

Section 3 applies the method of LoAs to the *process* that shapes the computational inforgs, namely the act of programming. How the strata of a computer program – from the source code to machine code – should be represented in terms of LoAs? What kind of information do they carry?

Section 4 shows the key argument: how the knowledge needed by the AG cannot be reduced to the features describing the computer program itself, but on the contrary a high degree of consciousness is required to write a significant computer program – the adjective 'significant' being crucial.

Section 5 introduces some well-known examples of non-standard, 'deviant' computer programming habits that are analysed in terms of LoAs within the computational inforgs. These examples are used to explain why only *humans* can be computer programmers since humans only are able to give non-predictable, creative solutions to computer programming problems, an unavoidable requirement in the light of Section 4.

Section 6 presents predictable objections to our argument and their respective counter-objections, while Section 7 concludes the article.

2. Definitions

Computational inforgs are instances of inforgs where the human part has – at least – a counterpart made by a VNM-based machinery, i.e. a standard, modern computer based on the original architecture by Von Neumann, which is the *de facto* standard implementation of the Universal Turing machine (Floridi, 1999, 44–46). The method of LoAs can be used to describe this kind of artefacts.

It is important to identify three different, complementary types of levels describing an artefact (Floridi, 2011b, ch. 3): the proper LoAs, the levels of organisation (LoOs) and the levels of explanations (LoEs). LoOs can be constructively defined as the observable knowledge about the artefact – e.g. the software source code – while the *correspondent* LoAs are in the mind of the human counterpart of the inforg – e.g. the software specification. Finally, LoEs are the pragmatically motivated epistemologies of LoAs, and their motivation relies in the correspondent LoA (Floridi, 2011b, 69).

The history of modern computers – roughly, from ENIAC in 1945 (Ceruzzi, 2003) – shows an increasing number of LoAs while the corresponding LoOs become more complex. Gobbo and Benini (2013) discuss how the architecture of modern computers can be described as intertwining VNMs, where the increasing number of LoOs is motivated by the *externalisation* of LoAs to allow the coherent hiding of LoOs' information to the human part of the inforg. For example, an operating system is an LoO that hides the information previously input by human operators by hand. This process of externalisation – from LoAs to LoOs – is central in the development of modern computing. In fact, it allows the ability to conceive a computer as an abstract entity performing complex tasks, instead of just a complicated electronic equipment – ultimately, justifying why computers are useful.

As a side effect, new epistemologies (i.e. LoEs) could rise exactly because of externalisation of LoAs in LoOs. For example, it is hard to conceive the end-user's LoE without the LoA of

software applications supported by the graphic interface known as the desktop metaphor found at Xerox Palo Alto Research Center in the 1970s (Ceruzzi, 2003, 257–263): for instance, the elementary action 'clicking a button' requires a sophisticated support from the graphical libraries and from the computer hardware – a support provided by the LoA's implementation in a number of LoOs; at the same time, the graphical aspect of a button invites to click it, referring to a metaphorical analogy well known to the users – which is part of the LoE of the graphical interface. Analogously, it is hard to conceive the computer programmer's LoE without (a) the externalisation of the process of compiling into a new LoA, i.e. the compilers, conceived by Böhm and Jacopini (1966) and (b) the externalisation of operators into operating systems – from Multics and especially Unix (Donovan, 1974).

It is important to notice that the LoEs are *always in the human part* of the computational inforgs. This fact is explained by the levels of consciousness needed to conceive 'different epistemic approaches and goals' (Floridi, 2011b, 69), applying the results of the KG.

In the original formulation by Floridi (2011b), the KG is a mental experiment which involves three kinds of populations that may be logically extended to four – see Table 1. The populations are classified by two Boolean variables: their nature (biological vs. non-biological) and their level of consciousness (present vs. absent).

The two populations at the extreme – i.e. human beings and strong AI AGs – are easily explained: human beings are members of the species *Homo sapiens*, they are naturally biological and show a considerable level of consciousness. As put by Dretske (2003), the point is not whether we are conscious – this can be given as granted; the point is *how* we are conscious. In fact, there is nothing in what we perceive through our senses that tells us we are conscious, but rather consciousness is revealed by the awareness of things we perceive – e.g. in the case of sight, some examples are points of view, perspectives, boundaries and horizons. The other extreme, i.e. strong AI AGs, is here only for logical completeness: there is no foreseeable realisation of conscious, non-biological AGs: 'the best artifacts that artificial intelligence (AI) will be capable of engineering will be, at most, zombies' (Floridi, 2011a, 3992).

On the contrary, the two populations in the middle – robots and zombies – need more clarification. The population of robots (and artificial AGs) is *currently* composed by engineered artefacts, where at least a *programmed*, VNM-based computer is always present as a control device. They can show '*interactive*, *autonomous* and *adaptable* behaviour' (Floridi, 2011b, 291) – and, sometimes, even *adaptive* behaviour, meaning that the machine tracks its interaction with the human being using it, and adapts its behaviour according to the internal model it builds up of the user, see Acquaviva, Benini, and Trombetta (2005) for a more technical view, whereas Dretske's (2003) zombies are biologically identical to human beings, except for consciousness. The main difference between robots and zombies, apart from being biological, is the ability to do counter-factual reflection. They are the ideally perfect behaviourists: every action is performed in the same way as humans, only without any awareness – without consciousness. A multi-AG system made by zombies may even pass the final KG-test (Floridi, 2011b, ch. 13).

Table 1. Possible populations of the KG.

Name of the population	Nature	Level of consciousness
Human beings	Biological	Conscious
Robots and artificial AGs	Non-biological	Non-conscious
Dretske's zombies	Biological	Non-conscious
Strong AI AGs	Non-biological	Conscious

Bringsjord (2011) rightly stresses that the key point in the KG relies in the notion of consciousness, which should be refined. Following the 'standard terminological furniture of modern philosophy of mind' (Bringsjord, 2011, 1363), in particular, Block (1995) distinguishes three levels of consciousness: *access-consciousness* (abbreviated as *a*-consciousness, which is equivalent to 'environmental consciousness', in Floridi's terminology); *phenomenal consciousness* (abbreviated as *p*-consciousness) and finally *self-consciousness* (abbreviated as *s*-consciousness). Table 2 shows that the only level of consciousness shared by all populations ('1', means presence; '0', absence; '?', unknown) is *a*-consciousness, i.e. the capability to represent a state and adapt the behaviour according to the inputs from the external world.

By contrast, *p*-conscious properties can be described as experiential (Bringsjord, 2011) or having the experience of 'what it is like to be' an AG in that state, while *s*-conscious properties additionally require introspective awareness (Floridi, 2011b, 292). It is worth noticing that *p*-consciousness is inferred from *s*-consciousness in the final move of the KG: there is a logical priority of *s*-consciousness over *p*-consciousness (Floridi, 2011b, 314). Both are internally oriented.

All terminological tools are now ready for the argumentation. In sum, we will deal with computational inforgs, *programmability* being the crucial feature – after all, that's why Turing called his machine 'Universal'. The explanation of computer programming in terms of the method of LoAs requires the distinction between: (a) proper LoAs; (b) LoOs being their *de re* counterparts and (c) LoEs being the collection of epistemological reasons behind. We have seen that the structure of the computational inforgs is shaped by the LoAs, which tend to be externalised into new LoOs implementing a process of information hiding, with the important consequence that new LoEs can emerge. However, the KG gives us the populations of AGs – logically four, pragmatically three – that can be part of computational inforgs, and a tripartite view of the notion of consciousness, where the internally oriented *p*-consciousness and *s*-consciousness are available only to human beings (Bringsjord, 2011; Floridi, 2011a), as far as we know.

3. What is a computer program?

As soon as compilers have become available (Ceruzzi, 2003, 201–203), the corresponding LoA allowed the creation of new programming languages – from here, simply 'languages'. In the sequel, we take into account only computational inforgs with compilers (or interpreters, but this variant is not influential for our line of reasoning). The programmer writes his algorithms in the form of source code, a piece of text which is then translated into a format which can be directly executed by the machine. This process hides information about the internal representation of data and its detailed manipulation, simplifying the programmer's task.

The usual expression *high-level* languages refer to the fact that a series of LoAs, which configure themselves as a *gradient of abstraction* (GoA) in the method of LoAs. In fact, each level is available to the programmer, being discrete and deputed to provide an abstract description (set) of manipulation tools for a coherent class of features. At the lowest level, the LoO is the hardware, with CPUs, buses, wires, etc. The logical interpretation of electric signals in terms of zeros and

Table 2. Populations and their level of consciousness.

Population	*a*-Conscious	*p*-Conscious	*s*-Conscious
Humans	1	1	1
Robots	1	0	0
Zombies	1	0	0
Strong AI AGs	1	?	?

ones – a major result by Shannon (1940) – is the first LoA of the GoA of computers programs; then, a stratified set of LoAs is provided, each one implemented by a suitable LoO, forming the so-called operating system; in modern computers, further LoAs are present, to cope with the graphical user interface, the networked services, etc. These LoAs are usually implemented as system libraries and services, which are the corresponding LoOs. Beyond the highest LoA, there is the pragmatic reason of creating a definite programming language:

> Just as high-level languages are more abstract than assembly language, some high-level languages are more abstract than others. For example, C is quite low-level, almost a portable assembly language, whereas Lisp is very high-level.
>
> If high-level languages are better to program in than assembly language, then you might expect that the higher-level the language, the better. Ordinarily, yes, but not always. [. . .] If you need code to be super fast, it's better to stay close to the machine. Most operating systems are written in C, and it is not a coincidence. (Graham, 2004, 150–151)

There is always a trade-off between usability (for programmers) and efficiency (for machines) in the design of a new programming language. In fact, the farther the language is from the LoOs close to the physical machine, the higher is the quantity of LoOs that are used to realise the language constructions. In this sense, the C language is very close to the machine, and it comes equipped with a minimal set of standard libraries; however, a language as Java is very far from the machine, and it comes with abundance of packages to interact with almost any conceivable system service, from web applications to databases. The choice of the exact point where to collocate the language is a main part of the LoE conceived by the language programmer(s).

The result of a process of programming is the program, written into the appropriate language – 'appropriate' in the sense of the LoE of the language. This result is a text that can be compiled and run into the computer system. As reported by Eden (2011), an ontological dispute about the ultimate nature of computer programs raised in the late 1990s, as they have a dualistic nature: the *program-script* represents the static, a-temporal entity written in the language, while the *program-process* is the dynamic, temporal entity which is put into the machine via the compiler, as illustrated earlier.

Eden and Turner (2011) provide a top-level ontology of programs, where *metaprograms* are the collection of the highest level abstractions, as algorithms, abstract automata (in particular, Turing's formalisms) and software design specification, including informal descriptions. However, metaprograms are not defined in that ontology (Eden & Turner, 2011, section 2.4). In the terms of the method of LoAs, followed here, metaprograms should be defined more precisely: if the specifications are described through some programming formalism, they are a proper LoA part of the GoA and a bridge to programs. On the contrary, if metaprograms are informal specifications such as 'program x translates French into English' (Eden, 2011, table 4) they do *not* belong to the GoA but rather to the pragmatic reasons, i.e. the rationale behind the program. In other words, they are part of the LoE. In the following, the term 'metaprogramming' will be used in the restricted sense, i.e. where it is part of the GoA.

4. The knowledge of computer programs

Traditionally, programs are thought of as formal description of a coordinated set of algorithms in a specific language, in order to solve a well-defined problem. The analysis made so far give a more complex and richer picture: the first result of the programming act is an LoO (the running program) which corresponds to an LoA (the program's features deputed to solve the problem); apart, there is an LoE which defines the purpose of the running object (the problem and its solution), which is the ultimate reason to write programs.

Let us assume that the programmer and the end-user of a program are different persons, with different abilities and ways to put into relations with the computer machinery. This assumption is quite commonsensical – at least since the advent of PCs in the 1980s (Ceruzzi, 2003, ch. 8).

The analysis just given is suitable for the end-user of the program, since it gives the physical object (LoO), the way to operate it (LoA) and the reason to use it (LoE), as a link between the user intention and the purpose of each part of the program as perceived by the user. In this sense, the *knowledge* about a program is the ability to perform a *conscious* interpretation of the purposes of each part of the program to obtain a result which satisfies the user's intention.

To some extent, the described picture applies also to the programmer: he or she operates on an abstract machine, given by the language and the system services, which constitutes the LoA; this abstract machine is implemented via the compiler, the operating system, the system applications and libraries, which constitute the LoO; finally, the programmer has a mental model of the various parts of the abstract machine, as the knowledge about the language and the various functions of the system, in the sense that he knows how to use them to implement algorithms, i.e. to express algorithms in the language so that they can compute the desired results, eventually interacting with the system services.

However, there is another point of view about the programmer that is quite distant and interesting for our purposes. In fact, the programmer has to write a piece of source code– program-script, in Eden (2011) – meeting some given specification, usually subject to a number of constraints. For example, he or she has to calculate the income tax to pay, given the balance of a firm in the form of a spreadsheet table with a fixed structure; the specification defines what the program is expected to do – and not to do, see the distinction between liveness and safety properties, introduced by Lamport (1977) – while the constraints declare some technical aspects which must be met, such as interacting with a specific spreadsheet. In the terms of the method of LoAs, the programmer has to conceive an LoA to solve the problem whose specification is (part of) the LoE, and the work should be completed producing an LoO, the running program, which realises the LoA. Moreover, the two LoEs are intertwined: the user's LoE roughly corresponds to the content of the end-user documentation of the program, while the programmer's LoA corresponds to the technical documentation of the program.

Therefore, the programmer's task is to build another LoA over the abstract machine he or she uses, to meet a goal corresponding to the LoE of the not-yet-existing LoA. To accomplish this task, a programmer has to understand the specification, devise a formal model of the world the specification lives in, design the algorithms operating in this hypothetical world to meet the specification, and, finally, code the algorithms in the language to obtain a source code program which can be successfully compiled and executed. Then, the final product, i.e. the compiled program, has to be checked against the specification to verify its correctness, that is, that it is really the LoO corresponding to the designed LoA, eventually going again through the whole process to solve the mistakes. The complete and detailed analysis of the whole process is beyond the scope of this article, and the reader is referred to a good textbook on software engineering, e.g. Peters and Pedrycz (2000).

The first step in the above-mentioned process is to understand the specification: if the specification is clear and well defined, the knowledge needed by the programmer would be the ability to read it and correctly interpret it. In principle, this ability can be performed by an artificial AG – in the sense explained in Table 2 – only *a*-consciousness is needed. In fact, there is an entire branch of theoretical computer science devoted to study the formal synthesis of programs (see, e.g. Benini, 1999), which deals exactly with the knowledge needed to write a program meeting a formal specification, which has no ambiguity with respect to a prefixed reference model. In principle, a formal specification in a suitable mathematical theory is enough

to write a program satisfying it: the requirement is that the theory should be powerful enough to convey in the program specification enough computational content to enable the mechanical extraction of a program that satisfies the specification. Unfortunately, the so-derived program is rarely 'good' with respect to the context: in most cases, even a poor human programmer is able to write a more efficient program code. Reality of life is that few specifications are formal, or even close to being formalised, making the initial step in the programming process far from trivial. This point is very important, so let us delve into it for a moment.

Even apparently formal specifications are not enough to determine a program meeting them: for example, it is easy to write a logical formula in the theory of orderings to express the fact that an array is sorted, making the specification 'given an array, sort it!' a perfectly clear statement. Nevertheless, this kind of specification is still part of the LoE, as it is too vague to be part of an LoA. In other words, that specification is not precise enough to provide an essentially unique solution to the problem. In fact, an entire book in the monumental work by Knuth (1998a) is dedicated to sorting algorithms. Consider the fact that different algorithms have different computational complexities, e.g. *insert sort* is $O(n^2)$ in time, while *heap sort* is $O(n\log n)$, with n the size of the array. But this by far not the only property to be considered. Interestingly, sorting algorithms which are more inefficient may perform far better if the arrays to sort belong to a restricted class, e.g. *insert sort* on quasi-sorted arrays operates on an average time of $O(n)$; the same algorithm can be coded in greatly different ways, e.g. recursive versus iterative *quick sort*. The abstract machine the programmer uses makes the choice even harder, as some operations may be significantly slower than others: for example, *2-way heap sort* may be quite inefficient on a large array in a virtual memory system, while *n-way heap sort* with $n > 2$ acts more 'locally', reducing memory swaps, thus being faster, even if this is not clear from a pure algorithmic analysis – most of these rather technical analyses can be found in Knuth (1998c) and in Peterson and Silberschatz (1985).

The point we want to make here is that, even in the case of a formal specification, a variety of choices is available to the programmer, due to the *context* where the program has to operate. The context is made of the programmer's abstract machine and the whole problem behind the specification. These aspects of the world's problem that are not explicit, and therefore they are part of the programmer's LoE. In the program-script, this is exactly what distinguishes a generated source code from a *significant* source code. In fact, a 'good' programmer is able to imagine the program in the context when it will be used, a clear instance of *p*-consciousness. A possible objection would be to overcome the knowledge of the *p*-consciousness by putting more and more information in the specification statement, to formally encode the whole set of constraints the programmer derives from his knowledge of the world where the program in supposed to operate in. As it is easy to understand, a formal specification which contains so much information may exceed by far the size and complexity of the final program implementing it – a phenomenon well-studied for mission-critical systems, such as aircraft control software, as reported in Mackenzie (2001) – while the final program-script becomes an unsolvable mystery in terms of maintenance.

As we have seen, specifications are not exhaustive in the sense that they are not strict enough to determine a unique piece of program-script source code as their solution. But there is another fact to be considered. Usually, specifications are not precise but rather an approximation of the required solution, since the problem itself is well understood or stated neither by end-users nor by programmers until the program-script is actually realised. For example, it is common practice to write web applications by refinement, validating an intermediate release with end-users: what is perceived as unsatisfactory or wrong is corrected in the next release and the features which are absent in the current version but required, even if not in the very beginning of the development process, are eventually introduced. This refinement process is unavoidable, as the requirements

are not really known in advance, and it is part of the life cycle of every software system, identified in literature as the maintenance phase (Peters & Pedrycz, 2000). This phenomenon proves beyond any doubt that knowledge and information are distinct, non-equivalent concepts, supporting once more Floridi's approach to PI.

Moreover, a third fact should be considered now. A good programmer writes the source code of program anticipating *maintenance*, which is the pragmatic counterpart of the refinement process just stated – the program-script is changed along the requests that emerge, while releases can be seen as milestones in its evolution. All criteria behind maintenance are part of the programmer's LoE, among the others: clear technical documentation, an intuitive choice for the source code architecture, a clean organisation of code and data structures, adherence to style rules are some of the strategies to make easier to modify an existing source code program. A somewhat extreme example is given by 'interfaces' in object-oriented programming (Gosling, Joy, Steele, Bracha, & Buckley, 2012): from the functional point of view, these pieces of code have no state and they do not provide any additional functions to existing packages and classes; instead, they provide a uniform access to a series of classes, which are distinct also in their structure, but, to some extent, identifiable under a common syntactical pattern, which is exactly what the interface codes. There is no reason to use interfaces in object-oriented programming, except to organise a complex set of classes as a clean structure, with a uniform access schema – an action which makes sense only to support maintenance.

From a philosophical point of view, the efforts a programmer makes to *anticipate* maintenance reveal that the programmer is *s*-conscious. In fact, a human programmer wants to minimise the future work caused by maintenance, which is an implicit goal of programming, internal to the programmer. Moreover, maintenance is not only unpredictable – the programmer does not know in advance how the current specification will be modified – but also *open-ended* – the programmer knows that the life of his or her code can be very long, without a predictable bound. The strategies to facilitate maintenance are directed by the principle: 'when you write a piece of code, always imagine that you will be asked to modify it exactly an instant after you completely forget how it works!' (folklore says that all programmers receive such an advice in their early life). Accomplishing this principle requires *s*-consciousness by definition.

Let us recall the main arguments behind the knowledge of programs. First, we cannot avoid to consider end-user's and programmer's LoEs, which are inherently different. Second, it is not possible to reduce the knowledge of the program in the LoE in terms of formal specifications, because not only its complexity would grow tremendously. Third, the program's LoE is made by the intertwinement of the end-user's and the programmer's, which is a complex process made of refinement, and ultimately determines the program-script in terms of subsequent releases. An important part of the programmer's LoE is to anticipate maintenance, which requires *s*-consciousness.

5. Computer programmers as artists

Much of the philosophical problems analysed in this article come from the fact that programs are perceived as technological objects. This is certainly true, but limiting our vision of programming as a technological activity denies some aspects which are vital to software development and to the advancement of computer science as a whole. In the following, we want to discuss some of these aspects, through a series of examples, as they provide other reasons why zombies cannot be programmers.

When a program is designed, the starting point is to define a problem to solve –as we have already discussed, the problem statement may be informal and underspecified – that is, there is a

specific behaviour the program must exhibit to satisfy an intended usage. For example, a web mail system, such as Gmail™ (Google Inc.), is supposed to provide email access to users even when they do not use their own computers. The service saves emails somewhere, so that they are readable from any web-enabled device after authentication; also, emails can be sent from everywhere using the same service. The intended use of Gmail™ is to communicate via a standardised electronic substitute of the traditional mail system.

But users have found deviant and creative ways of using Gmail™. We illustrate two of them, based on our first-hand experience. The first example is an example of creative use by end-users. Since the storage space of Gmail™ is very large, users use it to backup important documents: the user sends a message to himself with the document as an attachment; the direct effect is that he can read his own message and retrieve his document from any place where Gmail™ can be accessed. But there is also a side-effect: the message, and, thus, the document, is saved in Gmail™'s storage. Since the Google system is robust, the storage is a safe place to keep the document, safer than a computer for personal use or an in-house server, as both may break or lose data with no possibility to recover as they constitute a single point of failure. The document is 'protected' by the redundant nature of Google network of servers, providing, for free, a reliable backup service which is more robust than any other a normal user can afford.

The second example pertains the LoE of system administrators – which by definition are not end-users. Only Google's system managers can access the mail service with administrative powers. For this reason, some system administrators developed automatic procedures to periodically send messages, summarising their system's logs, to a fixed Gmail™ mailbox.[1] The purpose of those messages is to provide legal protection to the system managers and their institutions. In fact, as those messages cannot be modified by the institution and their personnel – potentially, they can be altered only by Google's system managers – they provide legal evidence of what happened to the institution's systems, an evidence that cannot be manipulated by an intruder, too – unless he is able to break the strong security system of Google. Therefore, in a court, those messages provide facts which cannot be disputed about the state of the system in time, because the exact time when they have been received by Gmail™ is recorded beyond any possibility of manipulation.

Evidently, these two examples of deviant use of a world-wide service have not been part of the specification of Gmail™. They reveal a creative understanding of the service, which is used beyond the intention it was designed for. In terms of the methods of LoAs, the users adopting these deviant examples, have added different LoEs to Gmail™, distinct from the one provided by the programmers of the system, even if still associated with the same LoO and LoA that constitute Gmail™.

Similar examples abound in programming practice. We now give an example to explain that the original, intended LoE externalised in the program-script can be used together with a deviant LoE at the same time. The example is an historical instance about the `RTS` instruction of the 6502™ (Motorola Inc. trademark) microprocessor.[2] This microprocessor was very popular in the times of home computing – roughly, 1977–1984, see for instance (Tomczyk, 1984) – as it was key hardware in the Apple II™ and of the Commodore 64™. The microprocessor had an instruction set with a limited ability to reference memory; in particular, branches were absolute but not relative: it was possible to say 'go to the instruction at address 1000', or 'go to the instruction whose address is in the memory cell 1000', but there was no instruction to say 'go to the instruction which is four instructions ahead'. This limitation was a problem for programming: when the executable program is loaded into memory, it must have the 'right' addresses in the parameters of the jump instructions, thus it had to be loaded at a fixed address, known at compile time. Otherwise, the program must be 'relocated' before execution: every

jump instruction must be adapted by the difference between the loading address and the compiling address.

Since relocation was time-expensive on such a slow microprocessor and loading programs at a prefixed address was not always possible, programmers soon found ways to write code which was independent from the loading position. The idea is to use in a deviant way the pair of instructions `RTS` and `JSR`: the 6502™ microprocessor provides the `JSR` instruction ('jump to subroutine') that operates by putting the address of the instruction onto the system stack and then jumping to the address given as a parameter; the `RTS` ('return from subroutine') instruction reverses the action of `JSR` since it takes the address on the top of the stack and jumps to the instruction just after. The intended use of these pair of instructions is to implement 'subroutines', i.e. functions needed more than once in the program, that can be called from any point of the program itself.

However, since every computer has a resident part of the operating system in memory, programmers found a way to use these instructions and the resident system to write position-independent programs. In particular, the resident system has a well-known address where there is an `RTS` instruction – for example, this was in the Apple II system documentation. It suffices to start the program with the following code: first, a `JSR` to the well-known `RTS` was performed, obtaining the side effect that one position over the top of the stack there was the address of the `JSR` instruction; second, that address was stored in a variable (memory cell with a known address) which acted as the base; finally, whenever a jump had to be performed, it was coded as an indirect jump to the instruction whose address was the base plus a displacement equal to the number of instructions separating the destination from the beginning of the program, i.e. the initial `JSR`.

We see that the `RTS` trick allows to overcome a design limit of the 6502™ architecture – in its instruction set, in particular – by a deviant use of a pair of instructions. The use is deviant because out of the intended meaning in the original design, even if correct with respect to the manipulation of the internal state of the microprocessor. In this case, the programmer uses *two LoEs at the same time* when writing the source code: the one provided by the microprocessor's manufacturer and a second LoE which substitutes the relative jumps with an indirect jumps from a displaced base address. Thus, the source code contains instructions, the jumps, which have to be read according to a different interpretation: they stand for relative branches, even if coded as indirect absolute jumps.

The main consequence of the `RTS` example is that rarely the source code of a program can be read literally: there are multiple levels of interpretations of its parts, corresponding to distinct LoEs. Only the composition of those LoEs allows to understand the program-script as an expression of the programmer's thought. An extremely interesting set of examples of this kind can be found in the beautiful book by Warren (2003), where programming techniques fully developing the power of computer arithmetic are described in depth. Those techniques reveal how far can be a program – as the formal description of a sequence of instructions in a language – from its source code, showing how multiple LoEs may interact in the production of an experienced programmer.

Let us consider another example, which shows how a deviant LoE became part of the standard LoE of the Internet for system administrators, as it concerns the Internet Control Message Protocol (ICMP) by Postel (1981). This network protocol, part of the standard Transmission Control Protocol/Internet Protocol (TCP/IP) suite, the reference implementation of the Internet architecture, is deputed to provide the services which are used to report errors and problems in the Internet protocol, and to provide basic services to check the correct behaviour of the network. The peculiar aspect of ICMP is that the protocol implementation relies on deviant uses of the IP protocol. For example, the `traceroute` service, which tests whether a given

host is reachable tracing the sequence of hosts in between, shows how to systematically misuse the standard mechanisms of IP networking. Specifically, the time to live (TTL) field in the IP packet is used to give a maximal life to an IP packet: whenever the packet is received by a network node, its TTL is decremented; if the TTL becomes zero, the packet is discarded and an error, in the form of an ICMP packet, is sent back to the source node. The rationale of this mechanism is to prevent a packet to circulate forever in a network when its final destination is unreachable; moreover, the error packet informs the source node that a packet has been lost. This mechanism allows some kind of recovery strategy, which is used – e.g. by TCP – to construct reliable channels over an unreliable network, and eventually being part of the standard LoE that underpins the Internet. Moreover, the `traceroute` service uses the mechanism differently: first, it sends to the destination an IP packet whose TTL is 1, thus generating an error at the first node it crosses; after receiving the corresponding error packet, which contains the address of the discarding node as its source, it generates another IP packet to the destination whose TTL is 2. Eventually, after enough IP packets are generated with increasing TTLs, a packet reaches the final destination. There, it gets discarded, as its TTL will be zero, and an ICMP error packet is sent to the source. When this error packet is received, the sequence of error packets contains the route from the source to the final destination, which is very important in the standard LoE of system administrators of networks.

At a first sight, all the illustrated examples appear as 'tricks'. But their systematic use reveals a deeper truth: they are part of the knowledge of a programmer, a knowledge that arises from alternative but coherent interpretations of the available LoOs and LoAs, i.e. alternative LoEs. In other terms, there is a creative step a programmer has to perform to understand and, then, to actuate these 'tricks': he or she must develop a new LoE, which is coherent with an existing LoA and LoO – a step comparable to the creation of an art work, which represent a fictitious reality but it is perfectly plausible. In the case of programming, plausibility of the LoE means that also the machine has to act coherently to the alternative explanation, so that the user may 'believe' to what he or she observes. The reader is invited to rethink to our examples in comparison with movies: a 'good' movie is not necessarily depicting some existing world, but, for the time of its projection, a viewer must think to the depicted world as 'real'. Similarly, although the alternative interpretations we illustrated are not the standard ones, they 'work', which is exactly the ultimate meaning of coherence with LoAs and LoOs.

Although the relation between art performances and programming can be carried further, and constitutes a fascinating topic to explore, our point is made: not only a programmer must be *s*-conscious, but he must also be creative on need. Thus, zombies and robots both lack the fundamental capabilities to be good programmers: at most, they can be coders, translating, in a more or less mechanical way, specifications into a flat text that can be correctly interpreted by a compiler. Hence, only humans are able to write significant programs, following the whole software development process, as strong AI AGs are non-existent for the moment, and probably forever. The required abilities for programmers are self-consciousness, to consider themselves as actors in the software development process and to derive specifications from their role, and creativity, to imagine and realise new explanations to existing abstract entities and features, so to solve problems by reinterpreting the available abstractions.

6. Objections and counter-objections

Three objections can be envisaged: the first objection concerns meta-reasoning in programming, the second refers to language stratification, while the third objection addresses the problem of a federation of zombies-as-programmers.

The first objection can be described as machines that program themselves: there are many techniques of reflect, introspect, shift from the object-level reasoning to meta-reasoning and vice versa – for a survey, see Costantini (2002). A program that modifies its (or other's) program-script is doing programming; therefore, a robot or an artificial AG (in the sense previously defined, see Table 1) are programmers even if they lack some level of consciousness (see Table 2). The objection is based on the implicit assumption: programming is the act of writing source code. As we have seen previously, programming should not be limited to the implementation of some formal specification, being a far more creative – even artistic – activity. In fact, whatever technique of reflection is used or can be found in the foreseeable future, there is no way to simulate the LoE behind the meta-programming process, which is informal and bound to *p*-consciousness and *s*-consciousness. Moreover, creating new and alternative LoEs is part of programming and this activity shows no signs of being mechanisable or able to be carried on by unaware entities.

The second objection can be posed as follows: there are no new LoEs in the programming process such as those illustrated in Section 5; the so-called alternative LoEs are, in fact, the explanations of new LoAs which abstract over the given LoAs – in concrete terms, in the illustrated examples, the user/programmer is not using the original program/language, but a new language which resides inside the old language, thus stratifying the original language/program into two interacting entities. The counter-objection is that every LoA must have a corresponding LoO, which is evidently shared by the whole stratification in the objection. For example, recalling the `RTS` trick, the same assembly language and the same microprocessor are used – no difference in the observables can be found between the single LoA/multiple LoEs – our interpretation – and the multiple LoAs – the objection's interpretation. Thus, even if we may think of an auto-relocatable source code as more abstract than the source code with fixed jumps, this abstraction is absent in the source code – it lies entirely in the programmer's mind, with no counterpart in the computational side of the inforg. Therefore, since it keeps the one-to-one correspondence between LoOs and LoAs, our explanation is more 'economic' than the counter-objection, as it does not force a reinterpretation of Floridi's model.

The third objection can be expressed as follows: if we concede that the KG 'promotes an intersubjective conception of agenthood' (Floridi, 2011a), then a federation of zombies behaving as computer programmers could be effective. In other words, we can supersede the programmer by splitting the need of an LoE in an indefinite number of behaviourist programmers. Here, there are two important aspects of programming that should be considered in order to address this objection. First, even if it is true that real-world, complex program-scripts are rarely written by a single programmer, in principle there is no advantage in having many programmers instead of one only – beyond efficiency. In fact, the ultimate product of the act of programming is a *sequential* text: the merging process of the program-script pieces written by the many programmers is not trivial, and it can be performed efficiently only if there is at least a partial agreement between each programmer's LoE. Again, issues in refinement and maintainability are in charge: there is no way to externalise merging in pure terms of *a*-consciousness.

7. Conclusion

This article, starting from the provocative title, analysed the act of programming in the Floridi's framework of PI, using the method of LoAs and the KG. We argue that programming is a highly complex activity, which involves consciousness and creativity, and it cannot be reduced to mere coding.

Our analysis is partial, because of the vastness and complexity of the explored field, and initial, as many aspects have been touched but not developed in depth. To name just a few, creativity is a fundamental attribute of programming and it relates to art – it is not by chance that Knuth's masterpiece is named 'The Art of Computer Programming'; this relation has not be systematically and philosophically explored yet, and we believe that an 'aesthetics of programming' may clarify many aspects why people find more pleasant or useful some programming solutions or techniques. Also, the analysis of consciousness in the programming life cycle has been limited to prove an argument satisfying our thesis: it can – and it should – be carried further to analyse the psychology of programming, an unexplored field which may help to understand how to organise programs so to stimulate production and quality, two of the major concerns of software producers.

We want to conclude acknowledging the source of inspiration for this article: the words at the beginning of the preface of Knuth (1998b):

> The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.

Funding

Dr Benini was supported by a Marie Curie Intra European Fellowship [grant number PIEF-GA-2010-271926], *Predicative Theories and Grothendieck Toposes*, within the 7th European Community Framework Programme.

Notes

1. One of the authors implemented this feature, only to discover that, independently, other system manager did the same: he was unable to trace the first proposer, or an article documenting the feature.
2. This example comes from a practice which was folklore in the Apple II™ programmers' community, according to one of the authors, who was part of that community.

References

Acquaviva, M., Benini, M., & Trombetta, A. (2005). Short-term content adaptation in web-based learning systems. In M. H. Hamza (Ed.), *Web technologies, applications, and services (WTAS 2005)*. Calgary: Acta Press.

Allo, P. (Ed.). (2011). *Putting information first: Luciano Floridi and the philosophy of information*. Oxford: Wiley-Blackwell.

Benini, M. (1999). *Verification and analysis of programs in a constructive environment* (PhD thesis). Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Milano, Italy.

Block, N. (1995). On a confusion about a function of consciousness. *Behavioral and Brain Sciences*, *18*, 227–247.

Böhm, C., & Jacopini, G. (1966). Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, *9*, 366–371.

Bringsjord, S. (2011). Metting Floridi's challenge to artificial intelligence from the knowledge-game test for self-consciousness. In P. Allo (Ed.), *Putting information first: Luciano Floridi and the philosophy of information* (pp. 1310–1838). Oxford: Wiley-Blackwell.

Ceruzzi, P. (2003). *A history of modern computing*. Cambridge, MA: MIT Press.

Costantini, S. (2002). Meta-reasoning: A survey. In R. A. Kowalski, A. C. Kakas, & F. Sadri (Eds.), *Computational logic: Logic programming and beyond: Essays in honour* (pp. 253–288). Berlin: Springer Verlag.

Donovan, J. J. (1974). *Operating systems*. New York, NY: McGraw-Hill.

Dretske, F. I. (2003). How do you know you are not a Zombie? In B. Gertler (Ed.), *Privileged access and first-person authority* (pp. 1–14). Burlington: Ashgate.

Eden, A. H. (2011). Three paradigms of computer science. *Mind and Machines*, *17*, 135–167.

Eden, A. H., & Turner, R. (2011). Problems in the ontology of computer programs. *Applied Ontology*, *2*, 13–36.

Floridi, L. (2011a). The philosophy of information: Ten years later. In P. Allo (Ed.), *Putting information first: Luciano Floridi and the philosophy of information* (pp. 3942–4360). Oxford: Wiley-Blackwell.

Floridi, L. (2011b). *The philosophy of information*. Oxford: Oxford University Press.

Floridi, L. (1999). *Philosophy and computing: An introduction*. London: Routledge.

Gobbo, F., & Benini, M. (2013). The minimal levels of abstraction in the history of modern computing. *Philosophy and Technology*, 1–17. doi:10.1007/s13347-012-0097-0

Gosling, J., Joy, B., Steele, G., Bracha, G., & Buckley, A. (2012). *The Java™ Language specification* (Java SE 7 ed.). Redwood City, CA: Oracle.

Graham, P. (2004). *Hackers and painters: Big ideas from the computer age*. Sebastopol, CA: O'Reilly.

Knuth, D. E. (1998a). *The art of computer programming*. Reading, MA: Addison-Wesley.

Knuth, D. E. (1998b). *Fundamental algorithms. Vol. 1: The art of computer programming* (3rd ed.). Reading, MA: Addison-Wesley.

Knuth, D. E. (1998c). *Sorting and searching. Vol. 3: The art of computer programming* (3rd ed.). Reading, MA: Addison-Wesley.

Lamport, L. (1977). Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, *3*, 125–143.

Mackenzie, D. (2001). *Mechanizing proof: Computing, risk, and trust*. Cambridge, MA: MIT Press.

Peters, J. F., & Pedrycz, W. (2000). *Software engineering: An engineering approach*. New York, NY: Wiley.

Peterson, J. L., & Silberschatz, A. (1985). *Operating system concepts* (2nd ed.). Reading, MA: Addison-Wesley.

Postel, J. (1981). Internet control message protocol, RFC792.

Shannon, C. E. (1940). *A symbolic analysis of relay and switching circuits* (Master's thesis). Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, MA.

Tomczyk, M. S. (1984). *The home computer wars*. Greensboro, NC: Compute.

Warren, H. S. (2003). *Hacker's delight*. Boston, MA: Addison-Wesley.