The Minimal Levels of Abstraction in the History of Modern Computing

Federico Gobbo[*]& Marco Benini[†]

Received: 13 April 2012 / Accepted: 3 December 2012

Abstract

From the advent of general-purpose, Turing-complete machines, the relation between operators, programmers, and users with computers can be seen in terms of interconnected informational organisms (inforgs) henceforth analysed with the method of levels of abstraction (LoAs), risen within the Philosophy of Information (PI). In this paper, the epistemological levellism proposed by L. Floridi in the PI to deal with LoAs will be formalised in constructive terms using category theory, so that information itself is treated as structure-preserving functions instead of Cartesian products. The milestones in the history of modern computing are then analysed via constructive levellism to show how the growth of system complexity lead to more and more information hiding.

Keywords: Epistemological levellism, Constructive levellism, Philosophy of Information, Computational interconnected informational organisms.
ACM class: K.2: History of Computing

[*]DISIM – Dep. of Inf. Eng. Comp. Science and Math., via Vetoio, 2, University of L'Aquila, IT-67100 L'Aquila, Italy, federico.gobbo@univaq.it

[†]Department of Pure Mathematics, University of Leeds, Woodhouse Lane, Leeds, LS2 9JT, UK M.Benini@leeds.ac.uk. Dr Benini was supported by a Marie Curie Intra European Fellowship, grant n. PIEF-GA-2010-271926, *Predicative Theories and Grothendieck Toposes*, within the 7th European Community Framework Programme.

1 Introduction

In the recent debate among philosophers of computing, it emerges that there are several formal theoretical concepts of information (Sommaruga, 2009). This is hardly surprising, as "the marriage of physis and techne" (Floridi, 2010) became more and more complex after the computational turn, whose historical origin can be traced back in 1936, when Church, Turing and Post obtained the major results about computability in their research about logic and the foundations of mathematics.

However, even if the universal Turing machine established the theoretical basis of modern computing, the advent of the Von Neumann machine (VNM) provided a real human-computer interaction: the more computing machineries became complex, the more intricate became the relations between operators, programmers, and users. During the last decades, physical layers of machines have been gradually replaced by abstract computing devices—even computers themselves can be fully made virtual nowadays, especially in the cloud computing paradigm. Philosophy of Information (PI) is the framework in which we analyse these complex relations, following Floridi's major contributions—for recent comments and discussions, see Allo (2011) and Demir (2012). In our approach, we borrow Floridi's concept of 'informational organism' (inforg), while the proposed method is a variant of the epistemological levellism, i.e., the philosophical view that investigates reality at different levels, where levels are defined by observation or interpretation. This variant is based on a rigorous, multi-levelled definition of information, where the levels are identified through the notion of abstraction (Floridi, 2011b, in particular, ch. 3), (Floridi, 2010, 2008), (Floridi and Sanders, 2004): the starting point considers numbers as symbols, thus providing an epistemological level of abstraction. Moreover, the underlying principle *aliquid (stat) pro aliquo*, something that stands for something else, can be carried on up to capture the complexity of modern computing systems.

While this kind of non-reductionist approach seems to be the right one in dealing with general inforgs, there are some practical disadvantages in using the method of levels of abstraction (LoAs) as such with *computational* inforgs. A computational inforg is an organism composed by (at least) a human being and by some kind of computing machine. Most often, the computing part is made of a VNM or some evolution: although other computational models beside Von Neumann's exist, in this paper we will limit ourselves to VNM-based computational inforgs, as this paradigm is by far the most important in the history of modern computing.

Because computational inforgs are *aliquid pro aliquo*, the history of VNM-based machines shows that the hiding of the computing technicalities to the human counterpart of the inforg and the growth in complexity of the inforg itself, both in the human and the machine sides, develop in pairs. For example, when the end-user types one or more keywords in the web page of a search engine like Google, what he or she expects as a response is a list of web pages related to the entered keywords, certainly not to know how the search engine was programmed to obtain that output. Even in the case of a computer programmer this fact holds: the programmer wants to obtain a sound answer when running the program itself, *independently* from the hardware and the operative system used to code his or her algorithms. So, in both cases of end-users and programmers, some essential pieces of information are hidden, and this very fact is what makes computer systems interesting for human beings, a fact already noticed by Turing (1950) dealing with Lady Lovelace's well-known objection.

So, a way to cope with the LoAs of computational inforgs in the case of hidden, implicit information should still be found. The method is based on the notion of *observables*, i.e. interpreted typed variables together with the corresponding statements of what features are under consideration; a LoA is the finite but non-empty set of the observables (Floridi, 2011b, 48). There are some examples of application of the method provided by Floridi, in particular: the study of some physical human attributes; Gassendi's objections to Descartes' Meditations; the game of chess; the semiotics of traffic lights in Rome and Oxford. None of these examples pertains to computational inforgs, which in our view is a clear limit; moreover, in that foundational paper Floridi declares that he "shall operate entirely within the boundaries of standard naïve set theory" (Floridi, 2011b, 50). In order to deal with computational inforgs and, in particular, with the phenomenon of information hiding, it is useful to put the variables and LoAs in a perspective that goes beyond set theory.

2 Abstraction within Constructive Levellism

Category theory (MacLane, 1998) can give a reasonably manageable and precise account of the growth in complexity of computational inforgs. The idea is to let information be a domain, represented as a mathematical category, so that abstraction becomes a map which preserves the inner structure of its domain, i.e., a *functor* in mathematical terms, and, thus, the LoAs are distinguished by what kind of information gets hidden to the human interacting with the machine.

If we adopt only methods and tools belonging to constructive mathematics, implicit information can be explained alongside explicit one, without the risk of being lost—at least to some extent. If we adopt a strict constructive attitude, for example P. Martin Löf's type theory as the reasoning framework, then we gain the ability to access implicit information on request; on the contrary, if we stick on a less demanding system, where choice principles are available, we may not have algorithmic access to hidden/implicit information, even if present in the system. However, even within less demanding systems, this fact does not imply that epistemological levellism cannot be used constructively. In fact, there is a recent attempt to apply it into the more general perspective of philosophical constructionism, which is a first step in this direction (Floridi, 2011a). What is stated here is that constructive philosophy or *constructivism*—i.e., the use of constructive mathematics to present philosophy in general and epistemology in particular—gives pragmatic advantages to the working philosopher of information, because it permits to distinguish very clearly information from knowledge, as rightly put by Primiero:

> [...] an epistemic distinction is needed among the notions of information and knowledge, and that it is possible and moreover natural to obtain it in a constructive framework. [...] In all the relevant explanations of the notions of knowledge and information, it has always been assumed that knowledge is explicit, and information is usually conflated with the

> content of knowledge. The relation of implicit/explicit containment is obviously essential in the understanding of the relation between premises and conclusion of an inference. [...] By referring to judgemental knowledge, it is instead possible to formulate implicit knowledge more clearly: [...] such an expression refers to what contained in an agent's knowledge frame every time something is explicitly asserted; this knowledge refers therefore to the collection of assertions required (but not necessarily expressed) by the meaningful assertion of a judgement (Primiero, 2008, 121).

The advantage of a constructive approach is that implicit knowledge is never lost but hidden, see Sambin and Valentini (1995) for the technical aspects, as in constructive mathematics the information content of statements is strictly preserved by proofs (Bridges and Richman, 1987).

There is another crucial point in epistemological levellism, i.e., how LoAs are related. In other words, what is meant by *abstraction*. While preserving the possibility to work in a purely constructive environment, category theory gives us a general and rigorous definition of abstraction[1]. We can understand how abstraction is treated within category theory through a very simple example:

$$\begin{tikzcd}
 & c_1 \arrow[loop above, "id"] \arrow[dr, "i"] \arrow[dl, dotted, "F"'] & \\
\cdot \arrow[loop above, "id"] \arrow[loop below, "\alpha"] & & c_2 \arrow[loop right, "id"] \arrow[dl, "o"] \arrow[ll, dotted, "F"'] \\
 & c_3 \arrow[loop below, "id"] \arrow[ul, dotted, "F"] &
\end{tikzcd}$$

Category A, on the left, has one object and two arrows: the identity (*id*) and α such that $\alpha \circ \alpha = \alpha$. Category C, on the right, has three objects and two arrows apart identities: i and o. It is possible to map each object in C into the only object in A in such a way that to every arrow in C corresponds an arrow in A, respecting identities and composition, as shown by the dotted arrows.

In general, given two categories A and C, A is *abstract* with respect to C, while C is *concrete* with respect to A, if there exists a functor $F\colon C \to A$. The key point is that the mapping of the objects of C into A preserves all the existing relations (arrows) within the source category, which is the meaning of F being a functor.

In the example it is evident that F is a functor, so A is abstract with respect to C, and C is concrete with respect to A: in fact, A hides the details of C by mapping the arrows i and o to α. If one thinks to C as the category representing the 'black box', then A is a representation of the monolithic computational paradigm: the arrow α, the

[1]It is beyond the scope of this article to explain why category theory is very close to constructive reasoning: it suffices to say that the natural frameworks to reason are *topoi*, sorts of 'complete' categories, and each topos comes equipped with an internal logic which is inherently constructive.

only relevant arrow of A, is the computational process, where the input, the processing, and the output are hidden. The functor F maps the input to α, the output to α, and their composition to α, which is the reason why α composed with itself must equal α. It is relevant to notice that F allows for recovering the concrete level from the abstract one, as the domain A is part of the definition of the functor. In traditional approaches, based on set theory, abstraction is some formal relation between concrete and abstract objects: in these approaches, arrows and a precise identification of domains and co-domains are both forgotten, preventing a constructive development from the very beginning, as pieces of information are, in principle, unrecoverable.

Actually, the category C is a faithful representation of the black box paradigm: the objects of C represent states in time, ordered by their indexes—c_1 is the initial state, c_2 represents the black box computing, while c_3 represents the state where the result is known. The arrows represent transitions: i feeds the black box with the input data, while o extracts the output data from the black box. Usually, a functional approach is preferred. As shown by Goldblatt (2006, 16), if we consider the input as the argument of a function and the output as its value, the black box itself is conveniently denoted as an arrow:

$$\begin{tikzcd} i \arrow[r, "\text{black box}"] & o \end{tikzcd}$$

When we limit ourselves to describe black boxes as categories, no *meaningful* information could be really found. In fact, in general, computing machineries are built with human beings as their final users and what we want to model is the relation between operators, programmers and users with computers, i.e., human-computer systems, or rather interconnected inforgs in Floridi's terms (Floridi, 2010, 2008). In this respect, we are interested in modelling the flow of information among the parts of a system rather than how computation is performed.

In the sequel of this paper, we address the modelling of computational inforgs starting from the concept of computer as a generic tool to perform computation, either concrete or abstract. We will use category theory as a guideline to describe abstractions, thus conserving in the abstraction functors hidden information, and we will proceed historically, i.e., from the ancient calculating machines to cloud computing. The aim is to find the *minimal* LoAs needed for modelling computing in different historical moments: so, we will show how evolution of modern computers has been shaped by abstraction, intended both as information hiding and the ability to recover the hidden information on need.

3 Minimal Levels of Abstraction in Modern Computing

The modern era of computing was born in 1936, when Church, Post and Turing showed the existence of universal machines, thus founding the future general-purpose computers on a solid theoretical basis. Later, calculation became the processing of a program, represented as a number in input, applied to some data, denoted by another number as input, eventually producing a number as the final result. The numbers are not numerical quantities anymore: they denote (sequences of) symbols forming the program, the input data and the final result. Here, an epistemological LoA can be found, as the machine works on numbers, but the human beings using that machine think of those numbers as programs and data, as the inner nature of a universal machine is to be 'generic'. It is important to notice how the LoA is entirely on the human side of the inforg, while the *inner* structure of the LoA is reflected on the corresponding Level of Organisation (LoO) in the machinery part of the computational inforg—'the system' (Floridi, 2011b, 69). A LoO is a

> structure in itself, or *de re*, which is allegedly captured and uncovered by its description, and objectively formulated in some neutral observation language (Floridi, 2011b, 69).

This view, inherited by classic AI (e.g., according to Newell's and Simon's views) is acceptable within epistemological levellism when there is a perfect correspondence between LoOs and LoAs. For instance, if an observer inspects the source code of a program written in some programming language, the observer will realise that many parts are deputed to facilitate this interpretation, i.e., type declarations, function prototypes, etc., imposing an organisation to the program. Similarly, also data are organised and their representation is highly structured. In other words, some LoOs have been hierarchically built *inside* the machine so that each LoA can be *externalised* by a correspondent LoO and, consequently, some information gets hidden. Hiding is only half of this process, which is really abstraction: the other half is the ability to recover the concrete representation from the data/program knowing the details of the abstraction. This second part is fundamental to enable many activities related to programming, e.g., debugging.

A consequence of the widespread use of LoOs is that inforgs involving modern computing start to become more complex: the general definition of information as *data + meaning* can be adequate only for the results of computation, but not for the whole process of information generation performed by the inforg, i.e., the human-machine system. In particular, *programmability* plays a crucial role, alongside computational efficiency and evolutionary adaptability (Conrad, 1995).

The act of programming is the act of symbolically representing algorithms as numbers. Hence, it is inherently an abstraction, where information gets partially hidden. The more inforgs grow, the more their pragmatic wills (from humans) and needs (to the machines) grow, the more LoAs (on the human side) and LoOs (on the machine side) should be found.

Figure 1 shows the model of the computational inforg of a VNM, from our general point of view in terms of category theory. The human-side LoAs are put at the top (with g indicating the goal), while the machine-side LoAs are put at the bottom (with M indicating the machine). The goal g is obtained in two steps: p represents the result of the act of programming, while data is encoded *supposing* the machine has been appropriately programmed $(M \times p)$. Now, the data can be executed $((M \times p) \times i \to o)$ to obtain the output o, which is *observable* (the dashed arrow) by the human-side computational inforg. Meaning is then obtained by the interpretation from the observation, e.g., at what degree the observer is surprised by the comparison between the output o and the goal g, following Turing's counter-objection to Lady Lovelace. Of course, the goal g cannot and should not be reduced to a generic notion of epistemic

$$\begin{tikzcd}
 & g \arrow[r, "\text{data encoding}"'] \arrow[d, "\text{programming}"'] & i & o \arrow[ll, dashed, bend right] \\
 & p & & \\
M & M \times p \arrow[l, "\pi_1"] \arrow[u, "\pi_2"] & (M \times p) \times i \arrow[l, "\pi_1"] \arrow[uu, "\pi_2"'] \arrow[uur, "\text{execution}"'] &
\end{tikzcd}$$

Figure 1: The computing model behind Von Neumann's architecture

surprise of the observer. However, it is out of the scope of this paper to investigate how this epistemic surprise is made, as it pertains entirely and only to the human-side of the inforg. Similarly, we do not discuss in this paper how computation is performed (the arrow $(M \times p) \times i \to o$), or how data and program are represented inside the machine (the arrows $(M \times p) \times i \to i$ and $M \times p \to p$, respectively). The diagram illustrates the fundamental steps in the interaction between human and machinery, which necessarily force a way to recover (the projection arrows π) the machine M, the program p and the input i from the executable object $(M \times p) \times i$.

It is interesting to notice that we used products to model coupling of machine, program and input data. In category theory, the product is a *limit*: it means that the product is *minimal* among the objects which allows to recover all its components via its projections, the πs arrows: in less formal terms, it means that no further elements beside the program, the machine and the input data are needed to perform the concrete computation; LoAs are responsible to provide the concrete elements to form the products from the abstract elements available to the human side of the inforg.

In order to show how VNM-based inforgs grow in complexity with the insertion of more and more levels of abstractions so to hide information, the next three sections will highlight the main points in history where the levels were inserted so to find the *minimal* number of needed levels.

4 From Operators to Operating Systems

The ENIAC and the other early computational machines, which are implementations of the Von Neumann's architecture, rely on human operators interacting with the machinery. These operators were responsible for coding the input i and for interpreting the output o according to the goal g. Also, engineers first and programmers later—from the 1950s onward—were responsible to write the program p deputed to process the data. Operators were not required to understand the internals of their computing machines, while programmers had to do so, thus different roles, or inforgs in our terminology, exist, each role with distinct competences. In particular, operators were expected to interact with some *hidden information* provided by programmers.

As machines were expensive, and a number of highly trained people were needed to operate them, both as programmers and as operators, there was a need to reduce the amount of persons and, consequently, the costs. This was done by writing programs to perform part of the duties of operators, a fact made possible by the birth of compilers: historically, this was the start of the mechanisation process which lead to the construction of the modern operating systems (Ceruzzi, 2003; Donovan, 1974).

Also, it was a waste of money to keep a machine idle when programmers and operators were preparing an execution. So, operators started to prepare 'jobs' for the machine, which were scheduled to execute one after the other with no pauses—it was called 'batch execution'. In this activity, they were assisted by the machine, which was able to take a job from a queue, execute it, signal that the output was finished and remove the job from the queue, restarting the execution cycle.

Soon, it was clear that machines were spending a large amount of time accessing devices and disks (or other storage devices). This was perceived as inefficient, as the machine as a whole was working, although all but one part of it were waiting. It would have been much more interesting and economical to use all the parts of the machine all the time to their maximum performance. So, special 'job control languages' were devoted to describe the scheduling task and their masters were the operators. But it was evident that operators, being human, were too slow to manage the interleaved executions of many processes. Hence, the need for an appropriate program became clear: its input was a series of jobs (pairs of programs and corresponding inputs), while its output was the results of the executions of the jobs. This special program had to find the optimal usage of the machine's resources and devices and, at the same time, to ensure that all the jobs were eventually executed. This kind of programs were the first operating systems.

The modern concept of 'operating system' (Donovan, 1974) can be seen as a new LoA: some tasks are hidden in an abstract machine operating on the computer system so that humans can forget them instead of manually perform their task as living operators: information gets hidden, without being lost. The hidden information is how to operate devices and how to optimally schedule computational tasks.

The side effect of operating systems was that the user could think to processes as being executed in parallel, as if each of them had a machine for its own purposes. Computing machineries were fast enough to convey the impression of parallel executions, even if no more than one process was really executed at any moment in time. These systems were called *multitasking systems*.

If we want to model multitasking systems, we must recognise two different abstractions: in the first place, we have a single physical machine M 'executing' a number of 'parallel' jobs. This abstraction, that enables us to use the quotes in the previous sentence, is the operating system. In the second place, each job is conceived to work inside an environment where the machine is fully dedicated to its execution. Again, this ability to isolate jobs from undesired interactions is the result of the operating system's action. But these abstractions are essential to write correct programs: programmers can safely assume to have the machine for their own purposes, without taking care of the possible interactions with other programs.

The resulting model of computation is depicted in Figure 2: it looks complex because we have all the abstractions at work—it is possible to simplify the diagram by cancelling arrows which can be obtained by composition, but they are useful in the fol-

$$\begin{tikzcd}
g \arrow[rr] \arrow[dr, "\pi_k"] \arrow[dd] & & i=\Pi_j i_j \arrow[d, "\pi_k"] & \\
& g_k \arrow[r, "\text{data encoding}"] \arrow[d, "\text{programming}"'] & i_k & o_k \arrow[ll, dashed, bend right] \\
p=\Pi_j p_j \arrow[r, "\pi_k"] & p_k & & \\
M_k & M_k\times p_k \arrow[l, "\pi_1"'] \arrow[u, "\pi_2"] & (M_k\times p_k)\times i_k \arrow[l, "\pi_1"'] \arrow[uu, "\pi_2"] \arrow[uur, "\text{execution}"] & \\
\Pi_j M_j \arrow[u, "\pi_k"] & \Pi_j(M_j\times p_j) \arrow[l, "\pi"'] \arrow[u, "\pi_k"] \arrow[uul, "\pi"] & \Pi_j((M_j\times p_j)\times i_j) \arrow[l, "\pi"'] \arrow[u, "\pi_k"] \arrow[r, "\text{execution}"] \arrow[uuuu, bend right, "\pi"'] & o \arrow[uuu, "\pi_k"'] \arrow[uuuulll, dashed] \\
M \arrow[u, "\pi"] & M\times p \arrow[l, "\pi_1"'] \arrow[u, "\pi"] \arrow[uuul, "\pi_2"] & (M\times p)\times i \arrow[l, "\pi_1"'] \arrow[u, "\pi"] \arrow[ur, "\text{execution}"'] \arrow[uuuuu, bend right, "\pi_2"'] &
\end{tikzcd}$$

Figure 2: The computing model behind multitasking architectures

$$\begin{tikzcd}
g \arrow[rr, dotted] \arrow[dd, dotted] \arrow[dr, dotted, "\pi_k"] & & i = \Pi_j i_j \arrow[d, dotted, "\pi_k"] & \\
& g_k \arrow[r, "\text{data encoding}"] \arrow[d, "\text{programming}"] & i_k & o_k \arrow[ll, dashed, bend right] \\
p = \Pi_j p_j \arrow[r, dotted, "\pi_k"] & p_k & & \\
M_k & M_k \times p_k \arrow[l, "\pi_1"'] \arrow[u, "\pi_2"] & (M_k \times p_k) \times i_k \arrow[l, "\pi_1"'] \arrow[uu, "\pi_2"] \arrow[uur, "\text{execution}"] & \\
\Pi_j M_j \arrow[u, dotted, "\pi_k"] & \Pi_j(M_j \times p_j) \arrow[l, dotted, "\pi"'] \arrow[u, dotted, "\pi_k"] \arrow[uul, dotted, "\pi"] & \Pi_j((M_j \times p_j) \times i_j) \arrow[l, dotted, "\pi"'] \arrow[u, dotted, "\pi_k"] \arrow[r, dotted, "\text{execution}"] \arrow[uuuu, dotted, bend right, "\pi"] & o \arrow[uuu, dotted, "\pi_k"] \arrow[uuuulll, dotted, no head, shift left] \arrow[uuuulll, dotted, shift right, "\pi_2"] \\
M \arrow[u, dotted, "\pi"] & M \times p \arrow[l, dotted, "\pi_1"'] \arrow[u, dotted, "\pi"] \arrow[uuul, dotted, "\pi_2"] & (M \times p) \times i \arrow[l, dotted, "\pi_1"'] \arrow[u, dotted, "\pi"] \arrow[ur, dotted, "\text{execution}"'] \arrow[uuuuu, dotted, bend right, "\pi_2"] &
\end{tikzcd}$$

Figure 3: Multitasking: single process abstraction

lowing explanation. If we consider the top and the bottom lines, they form the physical von Neumann's inforg, as previously described. This inforg operates on the set of goals g, inputs i and programs p, so to produce the output o.

In fact, if each goal is considered individually, focusing on just one of them, g_k, it produces the internal (Figure 3) Von Neumann's inforg, which is the programmer's abstraction, that operates on the M_k 'virtual machine', eventually producing the o_k result. The 'parallel execution' of the set of jobs is the picture in the mind of the operative system's designer, see Figure 4: in fact, he is part of an inforg which can be appropriately retrieved by composing projection arrows – i.e., the πs in the diagram. As before, projections take care of hiding most of the work behind the scenes of the operating system. Obviously, the concrete machine is still present, as the reader can see in Figure 5, and it wraps the preceding abstractions via appropriate projections. In this respect, it is interesting to notice that projections take care of 'implementing' the relations among the various LoAs, an intuition we borrowed from Floridi.

In the diagrams, the dashed and double dotted arrows refer to interpretations—the act of matching the output of a machine with the corresponding goal; dotted arrows show the part of the multitasking system that are hidden in the considered inforg.

$$\begin{tikzcd}
g \arrow[rr] \arrow[dd] \arrow[dr, dotted, "\pi_k"] & & i = \Pi_j i_j \arrow[d, dotted, "\pi_k"] & \\
& g_k \arrow[r, dotted, "\text{data encoding}"] \arrow[d, dotted, "\text{programming}"] & i_k & o_k \arrow[ll, dotted, bend right] \\
p = \Pi_j p_j \arrow[r, dotted, "\pi_k"] & p_k & & \\
M_k & M_k \times p_k \arrow[l, dotted, "\pi_1"'] \arrow[u, dotted, "\pi_2"] & (M_k \times p_k) \times i_k \arrow[l, dotted, "\pi_1"'] \arrow[uu, dotted, "\pi_2"] \arrow[uur, dotted, "\text{execution}"] & \\
\Pi_j M_j \arrow[u, dotted, "\pi_k"] & \Pi_j(M_j \times p_j) \arrow[l, "\pi"'] \arrow[u, dotted, "\pi_k"] \arrow[uul, "\pi"] & \Pi_j((M_j \times p_j) \times i_j) \arrow[l, "\pi"'] \arrow[u, dotted, "\pi_k"] \arrow[uuuu, bend right, "\pi"'] \arrow[r, "\text{execution}"] & o \arrow[uuu, dotted, "\pi_k"'] \arrow[uuuulll, dashed] \\
M \arrow[u, dotted, "\pi"] & M \times p \arrow[l, dotted, "\pi_1"'] \arrow[u, dotted, "\pi"] \arrow[uuul, dotted, "\pi_2"] & (M \times p) \times i \arrow[l, dotted, "\pi_1"'] \arrow[u, dotted, "\pi"] \arrow[uuuuu, dotted, bend right, "\pi_2"'] \arrow[ur, dotted, "\text{execution}"'] &
\end{tikzcd}$$

Figure 4: Multitasking: operating system abstraction

$$\begin{tikzcd}
g \arrow[rr] \arrow[dd] \arrow[dr, dotted, "\pi_k"] \arrow[dr, dashed] & & i=\Pi_j i_j \arrow[d, dotted, "\pi_k"] & \\
& g_k \arrow[r, dotted, "\text{data encoding}"] \arrow[d, dotted, "\text{programming}"] \arrow[rrddd, dashed, no head] & i_k & o_k \arrow[ll, dotted, bend right] \\
p=\Pi_j p_j \arrow[r, dotted, "\pi_k"] & p_k & & \\
M_k & M_k\times p_k \arrow[u, dotted, "\pi_2"] \arrow[l, dotted, "\pi_1"'] & (M_k\times p_k)\times i_k \arrow[l, dotted, "\pi_1"'] \arrow[uu, dotted, "\pi_2"] \arrow[ruu, dotted, "\text{execution}"] & \\
\Pi_j M_j \arrow[u, dotted, "\pi_k"] & \Pi_j(M_j\times p_j) \arrow[u, dotted, "\pi_k"] \arrow[l, dotted, "\pi"'] \arrow[luu, dotted, "\pi"] & \Pi_j((M_j\times p_j)\times i_j) \arrow[u, dotted, "\pi_k"] \arrow[l, dotted, "\pi"'] \arrow[r, dotted, "\text{execution}"] \arrow[uuuu, dotted, bend right, "\pi"'] & o \arrow[uuu, dotted, "\pi_k"] \\
M \arrow[u, dotted, "\pi"] & M\times p \arrow[l, "\pi_1"'] \arrow[u, dotted, "\pi"] \arrow[luuu, "\pi_2"] & (M\times p)\times i \arrow[l, "\pi_1"'] \arrow[u, dotted, "\pi"] \arrow[uuuuu, bend right, "\pi_2"'] \arrow[ru, "\text{execution}"'] &
\end{tikzcd}$$

Figure 5: Multitasking: the concrete Von Neumann's machine

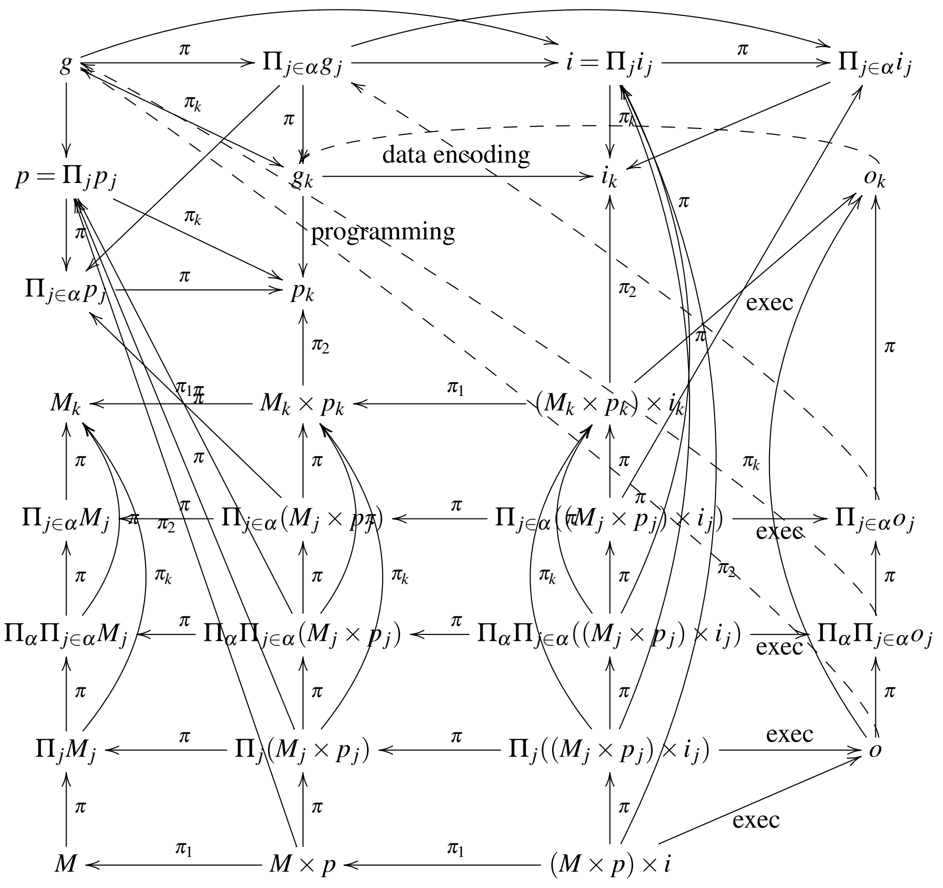
g
$\Pi_{j\in\alpha}g_j$
$i=\Pi_j i_j$
$\Pi_{j\in\alpha}i_j$
$p=\Pi_j p_j$
g_k
data encoding
i_k
o_k
programming
$\Pi_{j\in\alpha}p_j$
p_k
exec
M_k
$M_k\times p_k$
$(M_k\times p_k)\times i_k$
$\Pi_{j\in\alpha}M_j$
$\Pi_{j\in\alpha}(M_j\times p_j)$
$\Pi_{j\in\alpha}((M_j\times p_j)\times i_j)$
$\Pi_{j\in\alpha}o_j$
$\Pi_\alpha\Pi_{j\in\alpha}M_j$
$\Pi_\alpha\Pi_{j\in\alpha}(M_j\times p_j)$
$\Pi_\alpha\Pi_{j\in\alpha}((M_j\times p_j)\times i_j)$
$\Pi_\alpha\Pi_{j\in\alpha}o_j$
$\Pi_j M_j$
$\Pi_j(M_j\times p_j)$
$\Pi_j((M_j\times p_j)\times i_j)$
o
M
$M\times p$
$(M\times p)\times i$
π
π_k
π_1
π_2

Figure 6: The computing model of cooperative multitasking

Part of the operator's work, i.e., interpreting the output, is now responsibility of the operating system, which takes care of automatically dividing the output according to the process which produced it. This is an example of LoO.

Historically, soon after the development of the first multitasking operating systems, processes were allowed to interact. This behaviour requires another level of abstraction, which is the notion of *application*. This is best illustrated by an example: in Unix, printing a document requires to invoke a command, `lpr`, which takes the document as an argument. This program does not really print anything: it just prepares the document for printing and put it on a queue; another process, `lpd`, periodically takes the document on the top of the queue and prints it. This pair of commands forms an application: the Unix printing system.

In our picture, applications are modelled by introducing another pair of LoAs in the previous diagram—as done in Figure 6, which models groups of processes working together as a unit, as if they were a unique program in a standing alone Von Neumann's architecture. In the example above, if we do not introduce process interaction, the output of the process `lpr` (documents prepared for printing) could not be the input of the process `lpd` (printing queue) and hence its output would be null, as nobody had populated the queue, which is not what the user is expected to obtain by his goal, i.e., document printing.

Apart the complex technical appearance, it is worth remarking that the diagrams are obtained as interleaved composition of Von Neumann's architectures. For this reason, we have shown with some details the multitasking example, the dotted arrows meaning the *hidden* relations. The interested reader can do the same with Figure 6, which is again a collection of abstract VNMs working together.

5 Internet and Distributed Applications

There is no doubt that Internet changed the way in which we perceive applications: after the development of its infrastructure, the potential to write distributed applications was at hand. In fact, most Internet structural services are distributed applications, e.g., the Domain Name Server (DNS) system is a distributed database to convert numerical addresses into names and vice versa.

After the deployment of the World Wide Web (WWW) in the early 1990s, whose architecture is a traditional client-server platform, not essentially different from a terminal-mainframe system in the 1960s except for being distributed over the network, Internet became almost a synonymous of the Web, at least in popular culture, a usage reinforced in advertising (Berners-Lee and Fischietti, 2000).

As far as we are concerned, Internet applications are a natural evolution of a concurrent multitasking system, where the background is no more a single computing machinery, but a network of intercommunicating computers. So, in abstract terms, the architecture of the whole Internet can be depicted as in Figure 7. It is the same picture as Figure 6, except that the bottom line[2], the concrete implementing machine, has disappeared. This behaviour is clear: the concrete level of Internet is the set of computers which are interconnected, and they have not to be simulated, as we did in a multitasking environment.

In fact, a single computer can simulate—at least in principle—the entire Internet. This fact is due to the associative and commutative properties of the product, modulo isomorphisms. Again, we see here the power of rigorous abstraction as a driving force to system design.

6 The Era of End-Users

With the advent of window-based system, characterised by many applications interactively working together at the same time, a change in the concurrent multitasking schema take place: the outputs of several applications can be inspected at any time,

[2]Being commutative diagrams, we can safely omit arrows obtained by composition.

$$\begin{tikzcd}
g \arrow[r] \arrow[d] & \Pi_{j\in\alpha} g_j \arrow[r] \arrow[d] \arrow[ddl] & i = \Pi_j i_j \arrow[r] \arrow[d] & \Pi_{j\in\alpha} i_j \arrow[dl] \\
p = \Pi_j p_j \arrow[d] & g_k \arrow[r, "\text{data encoding}"] \arrow[d, "\text{prog}"] & i_k & o_k \arrow[ll, dashed, bend right] \\
\Pi_{j\in\alpha} p_j \arrow[r] & p_k & & \\
M_k & M_k \times p_k \arrow[l] \arrow[u] & (M_k \times p_k) \times i_k \arrow[l] \arrow[uu] \arrow[uur, "\text{exec}"] & \\
\Pi_{j\in\alpha} M_j \arrow[u] & \Pi_{j\in\alpha}(M_j \times p_j) \arrow[l] \arrow[u] \arrow[uul] & \Pi_{j\in\alpha}((M_j \times p_j) \times i_j) \arrow[l] \arrow[u] \arrow[r, "\text{exec}"'] \arrow[uuuur] & \Pi_{j\in\alpha} o_j \arrow[uuu] \arrow[uuuull, dashed] \\
\Pi_\alpha \Pi_{j\in\alpha} M_j \arrow[u] & \Pi_\alpha \Pi_{j\in\alpha}(M_j \times p_j) \arrow[l] \arrow[u] \arrow[uuuul] & \Pi_\alpha \Pi_{j\in\alpha}((M_j \times p_j) \times i_j) \arrow[l] \arrow[u] \arrow[r, "\text{exec}"'] \arrow[uuuuu, bend right] & \Pi_\alpha \Pi_{j\in\alpha} o_j \arrow[u] \arrow[uuuuulll, dashed] \\
\Pi_j M_j \arrow[u] & \Pi_j(M_j \times p_j) \arrow[l] \arrow[u] & \Pi_j((M_j \times p_j) \times i_j) \arrow[l] \arrow[u] \arrow[r, "\text{exec}"] & o \arrow[u] \arrow[uuuuuulll, dashed]
\end{tikzcd}$$

Figure 7: The computing model of network distributed applications

concurrently. This fact introduces another LoA over the previous picture: the whole output of an application is split: the external, visible part and the internal, hidden part.

An example may clarify: if we want to print a document on a Macintosh[3], we choose the 'Print' item in the 'File' menu; the application responds by showing a dialog box asking the printing options we require, and then, after we confirm our choice, it prints the document. As for Unix, the program which interacted with the user do not actually print a document, but it sends it to a printing process which acts like `lpd`. This process has an evident output, the printed document, but it may also show some visible output to the user, like a progress bar. So, the user perceives the output of the application, i.e., the set of processes deputed to treat the document, as coming from a single 'entity'. But some I/O takes places, since one process has to send input, the document, to another process, which prints it. Thus, the output of an application has to be divided into two parts: the *internal* one, the queued document which is of interest to the programmer, and the *external* one, the progress bar and the printed document which is what the user sees and wants.

Which kind of LoA is introduced here? From the point of view of system architecture, there is no difference between typing Unix commands on a shell and point-and-clicking menus and windows so to give the same command. The method of levels of abstractions gives the necessary philosophical and theoretical framework to deal with such cases. In particular, in the case of computing, each LoA is connected with a different level of organisation (LoO), which pertains to the software and hardware architecture, and a level of explanation (LoE), which pertains to the expected use by the end-user vs. the programmer:

> Strictly speaking, the LoEs do not really pertain to the system or its model. They provide a way to distinguish between different epistemic approaches and goals, such as when one analyses an exam question from the students' or the teacher's perspectives, or the description of the functions of a technological artifact from the designer's, the user's, the expert's or the layperson's point of view. A LoE is an important kind of LoA. It is pragmatic and makes no pretence of reflecting an ultimate description of the system. It has been defined with a specific practical view or use in mind. Manuals, pitched at the inexpert user, indicating "how to" with no idea of "why", provide a good example (Floridi, 2011a, 69),

As a LoE has not a correspondent LoO, there is no proper LoA which should be inserted in our model, unless we accept to structure the set of goals g, which is out of the scope of this paper, as it strictly depends on the human-side behaviour, which does not do the actual computation.[4]

Similar considerations can be brought for virtualisation, which is *only* a further stratum of information hiding, which becomes a proper LoE. As already seen previously, multitasking allows information hiding (Figures 3, 4, 5) letting us to model the perceptions by the ideal programmers and application end-users. We can consider cloud

[3]Macintosh is a trademark of Apple Computer Inc.

[4]However, there are graphical representations of the users' behaviours, notable Business Process Model and Notation (BPMN), which try to catch this blurry area of inforgs (Ryan et al, 2009).

computing as a further complex of LoEs. In fact, there is no change of system architecture (LoO) with respect to the standard Internet model: the management of files and applications (where to save? how to backup? when to update?) become invisible to the LoE of the end-user, because the system deals with them. Here, abstraction over the physical and logical devices takes care of hiding them to the users, but, at the same time, they are under control within the system, so to make their resources available on request. This fact stresses once more that abstractions must be constructive, that is, information – e.g., a storage device – must be hidden to provide the correct service, but it must be recoverable on request, to effectively implement the service itself.

Moreover, the gap between the programmer and the end-user becomes deeper: end-users are asked to *trust* the clouding service providers about the 'technicalities', as LoOs and LoAs are often commonly perceived, while programmers and system administrators, the modern operators, are expected to keep the service alive without worrying the end-users themselves. For an extensive application of the method of levels of abstraction to cloud computing, see Wolf et al (2012).

7 Conclusion and further directions of work

What are the minimal LoAs needed to explain computers through history? During the ancient era of computing, all LoAs were in the user's mind, as the machine was only an auxiliary tool: no symbolic interpretation was put into the device. This point was the big change of the modern era, where the universal computing machinery started to *hide* some symbolic interpretation of numbers through abstractions and organisations in parallel, where each LoO is the externalisation of the correspondent LoA. The more computers developed, the more information got hidden and needed reconstruction on demand: to correctly explain this historical process, we proposed here a constructive-based formalism.

The VNM is the first real inforg, as it pertains to machines and human beings together, and hence it forms the first LoA. The main steps in the history of computing may be seen as interleaving of VNMs, so that they form new LoAs: the next two LoAs are the concept of operating system and the notion of application. When the output of the application is split into a visible and an internal part, a fourth LoA should be added providing interactive systems, whose ultimate version is the window-based interface. It is interesting to notice, that this splitting permitted the fifth LoA, i.e., the distribution of the application processes over a network of physical computers.

No more LoAs are needed to explain the evolution of computer systems up to now, but at the same time the information hiding process caused by multitasking and the notion of application gave raise to at least two different *roles* in using computers: programmers and end-users. If it is true that no further LoAs are needed, it is also true that new LoEs are increasingly needed, especially to explain cloud computing.

In this paper, a special attention was given to the foundational part of computing, i.e. the machinery. But computer science is not merely the history of computing machines. Rather, from at least the end of the 1960s an increasing role was given to software (Ceruzzi, 2003), and since 2000 an increasing role is given to end-users, which started to get conscious of their role forming communities of practice.

These two evident limits can be overcome in two possible elements of expansion. First, a model in category theory can be developed to represent the relations between algorithms, programs, and functions—that is, to deal with the LoAs lead by software. In fact, there is a canonical forgetful functor from the category of programs to that of algorithms[5] and, analogously, from the category of computable functions to that of programs, but the functors from algorithms and functions to programs are far from being canonical. This kind of arrows between categories should be studied carefully, importing the results from many branches of Theoretical Computer Science into a unified philosophical framework.

Second, the goals g can be captured, at least in part, if we accept to hide the details behind the applications and try to define the behaviours of ideal programmers and end-users through the software applications they actually use. How to hide it accordingly is something that should be explored conveniently: this direction would develop the LoEs, i.e., practical uses of computers, more than LoAs.

These two directions are left to the future of our research efforts.

References

Allo P (ed) (2011) Putting information first: Luciano Floridi and the Philosophy of Information. Wiley-Blackwell

Berners-Lee T, Fischietti M (2000) Weaving the web. Harper Publishing

Bridges D, Richman F (1987) Varieties of Constructive Mathematics, London Mathematical Society, Lecture Notes Series, vol 97. Cambridge University Press

Ceruzzi P (2003) A history of modern computing. History of computing, MIT Press

Conrad M (1995) The price of programmability. In: Herken R, Herken R (eds) The Universal Turing Machine a Half-Century Survey, Computerkultur, vol 2, Springer Vienna, pp 261–281

Demir H (ed) (2012) Luciano Floridi's Philosophy of Technology: Critical Reflections. Philosophy of Engineering and Technology Book Series, Springer

Donovan JJ (1974) Operating Systems. McGraw-Hill

Floridi L (2008) The method of levels of abstraction. Minds Mach 18:303–329

Floridi L (2010) Information: A Very Short Introduction. Oxford University Press

Floridi L (2011a) A defence of constructionism: Philosophy as conceptual engineering. Metaphilosophy 42(3):282–304

Floridi L (2011b) The Philosophy of Information. Oxford University Press

[5] In a formal sense, this is evidently true, but, from a philosophical point of view, this is questionable: is a precise description of the steps toward a result an algorithm? Shouldn't an algorithm be a clear description of the way to obtain a well-defined goal? What is the goal of a random program? These questions are open issues in the philosophical analysis of algorithms.

Floridi L, Sanders J (2004) Levellism and the method of abstraction. Tech. rep., Information Ethics Group

Goldblatt R (2006) Topoi: the Categorial Analysis of Logic. Dover Books on Mathematics, Dover Publications

Mac Lane S (1998) Categories for the Working Mathematician. Springer

Primiero G (2008) Information and Knowledge: A Constructive Type-theoretical Approach. Logic, Epistemology, and the Unity of Science, Springer

Ryan KLK, Lee SSG, Lee EW (2009) Business process management (bpm) standards: A survey. Business Process Management Journal 15(5):1463–7154

Sambin G, Valentini S (1995) Building up a toolbox for Martin-Löf's type theory: subset theory. In Sambin G and Smith J (eds) (1998), Twenty-five years of constructive type theory, Proceedings of a congress held in Venice, October 1995, Oxford University Press, pp. 221-244.

Sommaruga G (ed) (2009) Formal Theories of Information: From Shannon to Semantic Information Theory and General Concepts of Information. Springer

Turing AM (1950) Computing machinery and intelligence. Mind 59:433–460

Wolf MJ, Grodzinsky FS, Miller KW (2012) Artificial Agents, Cloud Computing, and Quantum Computing: Applying Floridi's Method of Levels of Abstraction. In Demir (2012)