Programs as inscriptions

Simone Martini

Abstract Computer science shares with other disciplines concepts and methods for problem solving. Its distinctive contribution to these common methodologies is the language for doing them. What we (often dismissively) call programming languages are powerful tools for the modeling of reality which scale at several abstraction levels. We will bring some evidence on the role of such "programming" languages as inscriptions, and we will derive from this a simple suggestion for a roadmap for a Computer Science epistemology.

KEYWORDS: Programming languages; Philosophy of computing; Inscriptions; Abstractions; Abstraction mechanisms; Inscriptions.

August 2016

Simone Martini
Dipartimento di Informatica–Scienza e Ingegneria,
Università di Bologna and INRIA, Bologna, Italy.
Mura Anteo Zamboni, 7
40126 Bologna BO
Italy

email: simone.martini@unibo.it
tel: +39 051 2094979; fax: +39 051 2094510

Programs as inscriptions

Computer Science still lacks a fully-fledged epistemology. The borders of the discipline, its principal content, the methodology it uses, what kind of truths it establishes — are questions that have not been thoroughly discussed, and, even less, satisfatorily answered[1]. However, we may take for established that there are concepts that certainly pertain to the discipline: effectiveness and feasibility, information, interaction, abstraction and abstraction hierarchy, and many more. We will argue in this (short) paper that these concepts are intrinsically tied to the linguistic expressions for them, in a way radically deeper than in other sciences—after all, space, symmetry, matter, etc. exist independently from our linguistic expression for them. We will start from the bottom (computation), and we will work up the layers of abstractions levels to programming languages. In the second part, we will bring some evidence on the role of such "programming" languages as inscriptions, and we will derive from this a simple suggestion for a roadmap for a Computer Science epistemology.

1 Languages at the foundations of computing

1.1 Computation is symbol pushing

Turing's analysis (Turing, 1937) reveals the simple combinatorial structure of computation, as manipulation of symbols from a finite *alphabet* by a finite set of finite complexity rules. This alphabetic, mechanical nature of computing could have being missed by some contemporaries—Wittgenstein's dismissive comment ("Turing's 'machines': These machines are humans who calculate," (Wittgenstein, 1980)) seems to miss the point. While the deep introspective analysis of a human process is the basis, and the corroboration, of the definition of a machine, it is the fully abstract, combinatorial mathematical concept it generates the main contribution of Turing's paper. It will be Turing himself, in a kind of revealing parapraxis, to stress this linguistic basis. In his celebrated 1950 paper on "Computing Machinery" (Turing, 1950), at some point he discusses the way a child's mind becomes an adult's one. After a full stop, between parentheses, and before resuming his discussion without any further mention of the subject:

> (Mechanism and writing are from our point of view almost synonymous.)

It will be only in (Lassègue & Longo, 2012) that attention will be drawn to this sentence, argumenting that all systems for computability are "purely formal-alphabetic", and that "they are not" in the physical world. The compelling nature of Turing's analysis, that is the invariance of "the computable" with respect to the for-

[1] Which is not to say that there is no fundamental literature on the subject; e.g., (Turner, 2014; Floridi, 2011) or (Tedre, 2014).

malisms used to define it, is a consequence of this alphabetic nature—a computation in one system is formally ("lexically") coded into a computation of the other.

But if "the computable" is invariant, a specific computation is not. Coding does not preserve much of a computing process. On a Turing machine a computation may touch all the cells of the tape; on a Random Access Machine only a fixed number of registers will ever be used; and a lambda-term computes only a sequential function of its arguments. The abstract machine organizes its possible (alphabetic) computations by using certain linguistic constructs, differing from machine to machine. A computation exists (only) *on its own machine*. And a machine is there only to execute its computations. For a universal machine, this can be restated as "a universal machine exists to interpret (execute) its own language".

1.2 Hierarchies of abstract machines

It is John von Neumann to "abstract away" any concrete notion of machine from the description of a computation. In his *Report on EDVAC* of 1948, machines are *black boxes* for the functions they provide, the language they interpret. Even if we were to "open" such a black box, the only thing we would find is a bunch of lines of code, because universality allows for hierarchies of machines. A machine M_i uses the language of the machine M_{i-1} ("it is written in the language L_{i-1} of machine M_{i-1}") to implement (to express the interpreter for) its own language L_i. Any single construct of language L_i is expressed (interpreted, encoded) with a specific sequence of constructs of language L_{i-1}. Dually, certain patterns occurring in programs written in L_{i-1} are *abstracted* by linguistic constructs of L_i.

The creation of these abstractions in programming languages is a complex process, driven both by experiment and semantics. In (Visser, 2013) we find a discussion of this process of abstraction creation. First, a *programming pattern* is individuated, that is a recipe to solve a re-occurring problem, which the programmer applies manually in any instance (e.g., calling and returning sequences in assembly language, using the return stack). Then, a linguistic *abstraction* is devised and created, that is a construct providing a "black-box" for that pattern (e.g., functions and their parameter passing mechanisms). The essential point is that, once created, a "good" abstraction gets autonomous life, because it captures an important concept of software development.

> Over time the understanding of the abstraction in terms of the original implementation model erodes. New programmers learn to program with procedures without ever learning the underlying implementation scheme (or the mathematical semantics for that matter). The concept is no longer a convenience, but a first-class concept in thinking about software construction. (Visser, 2013)

It is in the shaping of this first-class concept that semantics and theory play a fundamental role. An abstraction introduced and "defined" by a specific implementation, over the time will be defined by other, more abstract means (e.g., denotational, or

axiomatic semantics), thus freeing the abstraction both from the programming pattern it was meant to replace, and from the details of the inferior language(s) in the hierarchies of abstractions. In some cases, semantics and theory will change in important ways our very understanding of the abstraction. A paradigmatic example is the concept of "object". It emerges as the pattern of *activities* (classes), *processes* (object instances) and local variables of processes (fields of an object) in Simula I (Dahl & Nygaard, 1966) and will be given its name as a single abstraction[2] first in Simula 67 (Dahl, Myhrhaug, & Nygaard, 1970) and then in Smalltalk (Goldberg & Kay, 1976). But it will be only (Cardelli, 1984) to single out inheritance as the characteristic feature of object orientation, and to express objects and inheritance as recursively defined records plus subtyping, thus making type systems the supporting skeleton of programming language design.

We have seen this process of abstraction creation all the times in these sixty years of programming languages: abstractions on control (functions, structured programming, exceptions, semaphores, threads, ...), abstractions on data (structured data types, dynamically allocated data, abstract data types, messages, ...), abstractions on control and data (objects, inheritance, modules, ...)—programming languages evolve converting new patterns into abstractions, and giving them autonomous life. Of course, new fields have been conquered by programming languages in this process: name passing models in concurrency (e.g., π-calculus (Milner, 1999)), real-time programming (e.g., Esterel (Berry & Gonthier, 1992)), web services (e.g., BPEL, or Jolie (Montesi, Guidi, & Zavattaro, 2014)). And others will be conquered in the near future: which abstractions will emerge from big data, or cloud and mobile computing, or machine-learning?

1.3 Programming languages

Under the perspective we have taken in the previous section, programming languages are sets of abstractions—good (elegant) languages are collections of *compatible* abstractions. And while we keep calling them "programming" languages, the part of modern languages dedicated to the description of computation is a tiny fraction of the language itself. They are, much more than "programming" languages, powerful tools to organize, make coherent, and model reality. They provide data models, procedural models, interaction models, synchronization models, organization models. Each language, with its own abstractions, provides its specific perspective on the reality it models. What is easy, immediate and terse in a language, could be obscure, derived and cumbersome in another. Certain patterns of reality emerge only in a language, and are irremediably lost in another. There is no language good for all uses. What is common to all these models is that they are discrete, effective, and, in the case of a good language, they will scale at different abstraction levels. These three characteristics make them radically different from other models used in

[2] "The objective of a language for talking about types is to allow the programmer to name those types that correspond to interesting kinds of behavior" (Cardelli & Wegner, 1985).

science, which are usually continuous, effective only in some (special) cases, and set at a specific abstraction level, with no way to move to a different level, either in the same language or by a uniform (compositional) translation into other languages[3].

The common alphabetic ground of these languages explains their raw equivalence, via coding. But of course it does not implies they are equivalent as a mean to model reality. Some abstractions, and the concepts they express, are emergent phenomena at their levels, despite the fact they are translatable (codable) with lower level languages.[4] The availability of flexible typing disciplines allow the natural (that is, at the same level of abstraction) modeling of complex relational structures, which, in a different language, could only be expressed as "coded" into other data structures.

We may say it with an Alan Perlis' aphorism: "A good programming language is a conceptual universe for thinking about programming." And it is too tempting to apply to programming languages what a translation theorist says of natural languages:

> A language fills a niche in the honeycomb of potential perceptions and interpretations. It articulates a construct of values, meanings, suppositions which no other language exactly matches or supersedes. [...] We speak worlds. (Steiner, 1997), p. 99.

Summarizing and concluding this first part, indeed *ipsa forma is substantia*[5]: The way we express a concept (an algorithm, a protocol, a software architecture, ...) is co-essential to that very concept. The essence of computer science lays in the immaterial linguistic expression of computation and interaction. There is never a fully faithful translation between one such expression and another.

2 Programs as inscriptions

Programming languages are not only the "tools of the trade" of the computer professional. They are the common language in which computer science is made, and they form an important part of that science's metalanguage. The design of Algol started with the purpose of the "publication of computing procedures in a concise and widely-understood notation", in such a way that they could be "mechanically translatable into machine programs for a variety of machines" (Backus et al., 1958).

---

[3] The canonical example here is physics and the differential equations it uses as a pervasive model: only in special cases such a model is effective in our technical sense, and sometimes simple problems are not even analytically solvable (e.g., the three-body problem).

[4] Obvious examples are protection mechanisms (for instance as present in object oriented languages with `private` attributes), which simply disappear at lower levels. More "concrete" phenomena, like dynamic method dispatching, can probably be identified in a compiled assembly code as a re-occurring pattern, but with no means to reason on them.

[5] *In substantiis autem intellectualibus, quae non sunt ex materia et forma compositae, [...] sed in eis ipsa forma est substantia subsistens, forma est quod est.* Thomas Aquinas, Summa contra gentiles. Book 2, 54. Intellectual substances are not composed of matter and form; rather, in them the form itself is a subsisting substance; so that form here is that which is.

The availability of such languages made possible the easy communication of new algorithms, and allowed reasoning on those algorithms (and, more generally, on programs). Even before Algol, it was realised that a programming language could be used to describe the semantics of its own interpreter, in a "meta-circular" way[6]. Bruno Latour, with genial insight, explains in this way the relationship between a new science and its language:

> No scientific discipline exists without first inventing a visual and written language which allows it to break with its confusing past. (Latour, 1986)

He writes this with reference to the analysis done by F. Dagognet on chemistry, but he could have been talked of computer science as well. At the end of the forties, once again in the hands of von Neumann, computer science starts to have its own *visual* and written new language, distinct from the language of mathematics and physics: flow-charts first, then formal algorithmic languages, as a distinct concept from the machines executing them.[7] Not later that 1951 Corrado Böhm describes a programming language and its compiler written in that same language (Böhm, 1954)[8]. This is truly the birth-date of computer science as an autonomous discipline, when it evolves from its "obscure" past (made of mathematics, cybernetics, logic, physics, engineering) and consciously presents itself as the science of algorithmic problem solving, for which it develops a new language. We should not make the error of identifying this "founding language" with a specific programming language. It is an early recognition that the contemporaneous presence of *different* specific languages (at various levels, with various purposes, with various targets[9]) is an asset of the discipline, and that no language will work for all uses.

We may be more specific, and see which specific purposes this language achieves. We follow (Latour, 1986), in this enumeration of characteristics. Programs:

- are "*immutable mobiles*."[10] The text of a program may be exchanged on a variety of media, and it does not change its semantics in this process. This should be contrasted with an actual computation (the execution of a program by one of its

---

[6] The first example is in (Böhm, 1954), discussed later; greater impact had the `evalquote` function of Lisp 1.5 (McCarthy et al., 1962); (Reynolds, 1972) classifies such meta-circular interpreters with respect the use of higher-order features.

[7] Also (Turing, 1947) has a clear view of the role of programming languages, and that they could appear at various levels. But the sharp distinction between a language and its machine (a black box) is von Neumann's contribution, as well as the first discussion in print of the visual language of flow-charts.

[8] The 1954 paper contains the material of Böhm's thesis, presented in 1951 at ETH, Zurich.

[9] Here is Turing that could be quoted, for an early recognition of the possibility of high level languages:

> Actually one could communicate with these machines in any language provided it was an exact language, i.e. in principle one should be able to communicate in any symbolic logic, provided that the machine were given instruction tables which would allow it to interpret that logical system. (Turing, 1947)

[10] See also (Goguen, 1992).

interpreters), which is much less mobile and requires a lot of infrastructure to be moved and, especially, to be maintained consistent after the move.

- are *flat*. Written programs are simple two-dimensional Euclidean objects, representing complex time-dependent phenomena, which are left implicit in the text. This absence of the time from a (standard) program is what makes them handy to conceive, understand, and manipulate.
- *their scale may be changed at will*. The availability of programming languages of different granularity and expressibility, and, moreover, the possibility to automatically translate a program from one language to another, allow for the description of a phenomenon at the desired abstraction level, in a such a way that, no matter the complexity, there is a scale (an abstraction level) at which the whole of the phenomenon may be dominated with the eyes and "held by hands", in a "single glance".
- may be *reproduced and communicated at little cost*.
- may be *reshuffled and recombined*. Contrary to a naive idea, this is not an immediate feature. Machine (or assembly) code may be recombined only with difficulty, and with great care. It is, instead, the precise goal of the abstraction mechanisms of higher level languages to allow easier reshuffling and recombination, because the information hiding provided by the abstractions defuses many of the possible interactions among different portions of codes.
- may be *made part of a written text*. That programs are meant to be executed is an evident tautology; what the novice misses is that programs are, first of all, meant to be read—by its author, then by its maintainers, and, more generally, its "clients". This "readability" has inspired important research programs, like Knuth's *Literate programming* (Knuth, 1984) ("let us concentrate (...) on explaining to human beings what we want a computer to do.")
- they *merge with geometry*. This must be explained, since Latour has in mind artefacts like geographical maps, and the requirement to merge with geometry means that one may work (reason) on the written, two-dimensional map, *as if* manipulating and reasoning on the three-dimensional objects. We may risk a reformulation: they are *a faithful model of reality*. Once again, this faithfulness (which is the reason why we use a program—a program of which we could not predict the behaviour from its text would be useless, in general) comes as a result of a complex hierarchy of levels and of interactions. That from counting the instructions of a very high-level program we get a sound asymptotic estimate of its actual running time on a specific architecture, is the result of the subtle simulations occurring between the many abstract machines executing that program, and the electronic phenomena in which it is ultimately translated.

In Latour's terminology (Latour & Woolgar, 1979), (Latour, 1987) programs are thus *inscriptions*. In this sense, programs are to computer science what maps are to geography, or the (visual) language for molecular structure is to chemistry. And like these other inscriptions, programs scale—scientists create *cascades of inscriptions*, where the inscription of level i is obtained as abstraction of one or several inscriptions of level $i-1$. We see, and use, only the the higher level abstractions, the ones that fit in a page, that could be taken in with a single glance.

There is also an evident, and crucial, difference between inscriptions like maps or molecular models, and computer programs. Programs are meant to be executed—they not only *represent* a complex spatio-temporal structure, they *cause* it—programs are *performative inscriptions*. What they cause, however, is again described (or prescribed) by a cascade of other inscriptions—the hierarchy of abstract machines implementing the programming language in which our program is written. It is this (meta-)circular hermeneutical interpretation that makes computer programs apart from other inscriptions. Here we reach a central point of our argument. Because, if we take this seriously, we also have a preliminary roadmap for a computer science epistemology: to study a science, observe first the instrument the scientist uses to produce his/her inscriptions, and then, and crucially, observe what she does with that instrument and *to* that instrument. The study of programming languages, and of their "conceptual" history (how a certain concept entered the field, how its semantics has been modified during the years, which linguistic mechanisms were proposed to "name" that concept in specific languages[11]) would become a blueprint for a more general epistemological investigation.

2.1 Languages for actions

In the *Prospectus* of the *Encyclopédie*, Denis Diderot explains at length the method used to compile its three parts—sciences, liberal arts, and mechanical arts. For the latter, in particular, they faced the problem to collect first, and express then, the knowledge and the competences (e.g., how to blow glass, or weave fabric). The only way was going *a bottega*—visiting the workshops, staying there and collect the information they could.

> We asked the most skilled in Paris and in the kingdom. We even went into their workshops [...] Among a thousand one will be lucky to find a dozen who are capable of explaining the tools or machinery they use, and the things they produce with any clarity.
>
> [D. Diderot, Prospectus à l'Encyclopédie, 141; 1751.]

One could dismiss the problem, and attribute the issue to the lack of education, or to the scarce acquaintance of those artisans with general explanations. Of course, in this way we would instead *miss* the problem. There is a whole array of competences which are expressed in actions and very difficult to express with words.

> Inarticulate does not mean stupid; indeed, what we can say in words may be more limited than what we can do with things. [...] Here is a, perhaps the, fundamental human limit: language is not an adequate mirror-tool for the physical movements of the human body.
>
> (Sennett, 2009), p. 95

For narrowing the gap between "what we can do with things" and "what we can say in words", Sennett acknowledges (and finds in the *Encyclopédie*) the substitution of

[11] See, e.g., (Martini, 2016a, 2016b) for a preliminary investigation of this kind, on the notion of "type" up to the seventies.

"the image for the word." Sequences of images, after "all the junk of an ordinary workshop has been eliminated" (that is, after the essential abstraction is obtained from a lower level) clarify single movements and actions, so that they could be reproduced also by workers unable to use words for this purpose. There are situations, however, where also images are not sufficient. This is especially the case when higher standards and excellence are required. Because the details, the tricks, the experience and the eye of the master, are too complicated, and too hidden, to be condensed in inscriptions. The example Sennett brings forth is Stradivari's liutery workshop, where

> the experience of doing high-quality work was contained in the masters own tacit knowledge, which meant his excellence could not be passed on to the next generation.
>
> (Sennett, 2009), p. 243

Is there a chance that the generality of programming languages, their protean ability to conform to different domains, could make them into instruments for saying "what we can do with things," even when pictures and other inscriptions would not help?

> Programming languages provide a way for us to describe to each other what we know how to do. [...] [They are] intellectual organizing principle[s] for understanding and describing the past, and making sense of the kinds of expertise that flourished and came to maturity.
>
> (Mairson, 2013)

It is more than apt this quotation, coming from a paper which applies programming language design techniques to liutery. It describes *linguistic abstractions* for the concrete production of the geometries of Amati's and Stradivari's violins, revealing regularities and patterns "known by heart" (but we would better say "by hand") by the artisans of the field, but which lacked the language in which they could be "inscribed".

The language of computer science allows for radically new way of saying things (and saying them clearly). It provides that missing language for saying things that were previously inexpressible in several areas of the human experience, in the fruitful plurality of specific programming languages and levels of descriptions.

References

Backus, J. W., Diselets, P. H., Evans, D. C., Goodman, R., Huskey, H., Katz, C., ... Wegstein, J. (1958). *Proposal for a programming language* (Tech. Rep.). ACM Ad Hoc Committee on Languages.

Berry, G., & Gonthier, G. (1992). The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, *19*(2), 87-152.

Böhm, C. (1954). Calculatrices digitales. Du déchiffrage des formules logico-mathématiques par la machine même dans la conception du programme. *Annali di matematica pura e applicata*, *IV-37*(1), 1-51.

Cardelli, L. (1984). A semantics of multiple inheritance. In *Proc. of the international symposium on semantics of data types* (Vol. 173, pp. 51–67). Springer-Verlag. (Extended journal version: Inf.& Comp., 76, 138-164, 1988.)

Cardelli, L., & Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, *17*(4), 471–523. doi: 10.1145/6041.6042

Dahl, O.-J., Myhrhaug, B., & Nygaard, K. (1970). *The SIMULA 67 common base language* (Tech. Rep. No. Publication S-22). Oslo: Norwegian Computing Center.

Dahl, O.-J., & Nygaard, K. (1966). Simula: An ALGOL-based simulation language. *Commun. ACM*, *9*(9), 671–678. doi: 10.1145/365813.365819

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

Goguen, J. A. (1992). The dry and the wet. In *Proceedings of the IFIP TC8/WG8.1 working conference on information system concepts: Improving the understanding.* (Vol. A-4, pp. 1–17). North-Holland.

Goldberg, A., & Kay, A. (1976). Smalltalk-72 instruction manual [Computer software manual].

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, *27*, 97-111.

Lassègue, J., & Longo, G. (2012). What is Turing's comparison between mechanism and writing worth? In S. B. Cooper, A. Dawar, & B. Löwe (Eds.), *Cie* (Vol. 7318, p. 450-461). Springer.

Latour, B. (1986). Visualisation and cognition: Thinking with eyes and hands. In H. Kuklick (Ed.), *Knowledge and society studies in the sociology of culture past and present* (Vol. 6, p. 1-40). Jai Press.

Latour, B. (1987). *Science in action: How to follow scientists and engineers through society*. Harvard University Press.

Latour, B., & Woolgar, S. (1979). *Laboratory life*. Sage Publications.

Mairson, H. G. (2013). Functional geometry and the traité de lutherie. In *Proceedings of the 18th ACM SIGPLAN international conference on functional programming* (pp. 123–132). New York, NY, USA: ACM.

Martini, S. (2016a). Several types of types in programming languages. In F. Gadducci & M. Tavosanis (Eds.), *Hapoc 2015* (p. 216-227). Springer.

Martini, S. (2016b). Types in programming languages, between modelling, abstraction, and correctness. In A. Beckmann, L. Bienvenu, & N. Jonoska (Eds.), *Cie 2016: Pursuit of the universal* (Vol. 9709, p. 164-169). Springer.

McCarthy, J., et al. (1962). LISP 1.5 programmer's manual (MIT Press ed.) [Computer software manual].

Milner, R. (1999). *Communicating and mobile systems: The π-calculus*. Cambridge University Press.

Montesi, F., Guidi, C., & Zavattaro, G. (2014). Service-oriented programming with Jolie. In A. Bouguettaya, Q. Z. Sheng, & F. Daniel (Eds.), *Web services foundations.* Springer.

Reynolds, J. C. (1972). Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th acm national conference* (pp. 717–740). ACM.

Sennett, R. (2009). *The craftsman*. Yale University Press.

Steiner, G. (1997). *Errata*. Yale University Press.

Tedre, M. (2014). *The science of computing: Shaping a discipline*. Chapman and Hall/CRC.

Turing, A. M. (1937). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, *s2-42*, 230–265.

Turing, A. M. (1947). *Lecture to L.M.S. Feb. 20 1947*. Typewritten notes in Turing Archive, AMT/B/1. (In Turing archive, AMT/B/1)

Turing, A. M. (1950). Computing machinery and intelligence. *MIND*, *59*(236), 433–460.

Turner, R. (2014). The philosophy of computer science. In E. N. Zalta (Ed.), *The Stanford encyclopedia of philosophy (Winter 2014 edition)*. http://plato.stanford.edu/archives/win2014/entries/computer-science: Stanford University.

Visser, E. (2013). Understanding software through linguistic abstraction. *Science of Computer Programming*. (To appear) doi: 10.1016/j.scico.2013.12.001

Wittgenstein, L. (1980). *Remarks on the philosophy of psychology* (Vol. 1). Blackwell.

Note: This bibliography was generated from the `IFES.bib` bibliography file, using *bibtex*.