HaPoP-5

Fifth International Symposium on the History and Philosophy of Programming

13 June 2022, Lille, France
European Center for the Humanities and Social Sciences
Co-located with the final conference of the ANR-funded PROGRAMme project

PROGRAM AND SHORT ABSTRACTS

Organised under the auspices of the DHST-DLMPST Commission for the History and Philosophy of Computing - HaPoC

Conference Aims and Scope

In a society where computers have become ubiquitous, it is necessary to develop a deeper understanding of the nature of computer programs, not just from the technical viewpoint, but from a broader historical and philosophical perspective. A historical awareness of the evolution of programming not only helps to clarify the complex structure of computing, but it also provides an insight in what programming was, is and could be in the future. Philosophy, on the other hand, helps to tackle fundamental questions about the nature of programs, programming languages and programming as a discipline.

HaPoP 2022 is the fifth edition of the Symposium on the History and Philosophy of Programming, organised by HaPoC, Commission on the History and Philosophy of Computing. As in the previous editions, we are convinced that an interdisciplinary approach is necessary for understanding programming with its multifaceted nature. As such, we welcome participation by researchers and practitioners coming from a diversity of backgrounds, including historians, philosophers, artists, computer scientists and professional software developers. What is a computer program?

What is a computer program?

This edition of the symposium will be co-located with the final conference of the ANR-funded PROGRAMme project which poses the basic question "What is a computer program?" This seemingly simple question has no simple answer today, but the responses one gives to it affect very real problems: who is responsible if a given piece of software fails; whether a program is correct or not; or whether copyright or patent law applies to programs. The project is anchored in the conviction that a new kind of foundational research is needed. The broad range of scientific and societal problems related to computing cannot be addressed by any single discipline.

The question "What is a program?", is a call for deeper critical thinking about the nature of programs that is both foundational, in the sense that it goes beyond specific problems, but also accessible, in the sense that it should be open to anyone who is willing to make an effort in understanding this basic technique from a broader horizon.

In order to open up the ongoing work on the PROGRAMme project, initiate new collaborations that critically reflect on the nature of programs and engage a broader community with the above issues, HaPoP 2022 is particularly looking for talk proposals that relate to the question "What is a computer program?" and offer a novel reflection from a variety of perspectives, including historical, practice-based, philosophical, logical, etc.

Chairs

Liesbeth De Mol (CNRS - Université de Lille)
Tomas Petricek (Kent University)

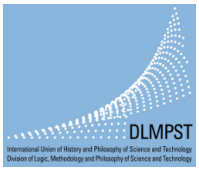
Program committee

Arnaud Bailly, Aleryo
Martin Carlé, Ionean University, Corfu
Liesbeth De Mol (co-chair), CNRS, UMR 8163 STL, Université de Lille
Andrea Magnorsky, Independent
Ursula Martin, University of Oxford
Baptiste Mélès, CNRS, UMR 7117 Archives Henri-Poincaré
Tomas Petricek (co-chair), University of Kent
Mark Priestley, National Museum of Computing, Bletchley
Giuseppe Primiero, University of Milan

Supported by

ANR PROGRAMme project

DHST-DLMPST commission on the History and Philosophy of Computing



European Center for the Humanities and Social Sciences

Université de Lille

UMR 8163 Savoirs, textes, Langage

Schedule

9h50-10h00: Introduction and welcome

10h00-11h10 – Session 1
Chair: Liesbeth De Mol
Warren Sack. *Miniatures, Demos and Artworks: Three Kinds of Computer Program, Their Uses and Abuses*

Shoshana Simons. *Programming practice as a microcosmos of human labor and knowledge relations*

11h10-11h40: Break

11h40-12h50 – Session 2
Chair: Tomas Petricek
Lucas Bang. *Program Size: A Call for Inquiry*
Andre Dickson. *The disturbance of death and debt*

12h50-14h30: Lunch

14h30-15h40 – Session 3
Chair: Mark Priestley
Daniel Kuby. *Towards a linguistic conception of programming*
Nicolas Nalpon and Alice Martin. *Why semantics are not only about expressiveness - The reactive programming case*

15h40-16h10: Break

16h10-17h20 – Session 4
Chair: Martin Carlé
Fabien Ferri. *Diagrammatic writing as writing the non-programmable: a case study from the DRAKON graphical modeling language*
Robin Hill. *Hello World? An Interrogation*

20h00-22h00: Diner in Brasserie de la Paix

Abstracts of Contributed Talks

Program Size: A Call for Inquiry

Lucas Bang
Harvey Mudd College, USA
bang@cs.hmc.edu

Size is a fundamental program characteristic that has deep implications for the phenomenological experience and influence of computer code. Program size has a paradoxically rich yet sparse theoretical history that deserves closer examination. Here, I highlight the importance of program size, using a relatively obscure size theorem of Blum [1] as a beacon, and discuss the relevance of program size on the past, present, and future of programming.

Theory in Praxis. Programmers are typically familiar with several foundational results in theoretical computer science and these results show up with varying frequency in the day-to-day experience of most people who regularly write code. A few examples will help illustrate. Ideas like undecidability, incomputability, and incompleteness [6, 17, 3] are required topics in many computer science programs, but it is not common to take them into practical consideration while performing routine software development tasks. Slightly more consideration-worthy on a regular basis is the notion of complexity classes, notably NP-completness [10]. It is indeed the case that the ability to recognize NP-complete problems can have a real impact on programmers, users, and others affected by solutions to scheduling, planning, routing, and optimization programs. Even more common are the considerations for algorithmic space and time complexity [5]; programming almost always accounts for memory and time efficiency. Finally, program correctness [12] is of utmost concern; by some accounts, programmers spend 35-50% of their time validating code [2]. If you have experience programming, I would encourage you to reflect on how relative frequency with which you have taken each of these facets of theoretical computer science into account while coding.

I claim that the degree to which we consider theoretical areas of computing (e.g. each of the the aforementioned computability, complexity classes, algorithmic complexity, and correctness) in our daily lives is proportional to the potential for phenomenological discomfort caused by not taking them into consideration. I support this claim through the lens of Heidegger's ready-to-hand and present-at-hand framework [8]. Further, I claim that program size is indeed of the utmost importance to the daily activity of programming and to the effect that programs have on society. Abstractions are born, to some degree, in order to manage program size complexity. When an abstraction does not behave as expected, the programmer shifts from a state of ready-at-hand to present-at-hand in the face of large programs. Yet, there does not currently exist a common familiarity with theoretical results related to program size providing a vocabulary that affords reflection or communication about such concerns.

Remember Not To Forget. Among theoreticians, Kolmogorov Complexity, is well-know. For a string s, the Kolmogorov Complexity of s is the size of the smallest program in a universal programming language that outputs s when provided no input [11]. Less well-known, but, I argue more relevant to day-to-day programmer experience is Blum's Size Theorem, which is not as simple to state upon first glance [1].

Blum Size Theorem (1967). *Let ϕ_i be an indexing of recursive functions, g be a recursive function with unbounded range, and f be any recursive function. We can find indices $i, j \in \mathbb{N}$ such that $\phi_i = \phi_{g(i)}$ and $f(|i|) < |g(j)|$, that is, the size of $\phi_{g(i)}$ is arbitrarily larger than the size of ϕ_i though they compute the same function.*

In Robert Harper's *Practical Foundations for Programming Languages* [7], and also in a blog post, "Old Neglected Theorems are Still Theorems" [1], he briefly but clearly spells out the consequences of Blum's Size Theorem: for a programming system in which you are guaranteed to be able to know for certain that any program behaves as desired (e.g. halts), there are specifiable problems for which the smallest solution is an unfathomably large program. As a concrete example of this fact, Harper points to the Euclidean Algorithm for computing the greatest common divisor of two natural numbers (perhaps a few lines of Python). Writing GCD quickly becomes rather unwieldy in the restricted programming system System T where all programs are guaranteed to converge. I can personally attest to having done so and it is indeed rather painful and the program grew quite large.

[1] https://existentialtype.wordpress.com/2014/03/20/old-neglected-theorems-are-still-theorems/

An extensive search has turned up only three meaningful discussions since the year 2000 on the practical relevance of Blum's size theorem to the concerns of programming, only one of them published in an official capacity, and all of them brief. Two of these are the blog and book referenced in the preceding paragraph. Further, Robert Constable at Cornell (Harper was a PhD student of Constable) appears to have taught the Blum Size Theorem within the last several years and discussed its importance to the act of producing programs [2]. Constable and Albert Meyer appear to have been actively exploring program size theory in the 1970s, but not much has happened since then [4, 13].

Modern Relevance. There are perhaps many reasons that the Size Theorem and related results are not more well-known, but one can only speculate. Regardless, I encourage further inquiry into results on program size, how they fit into the spectrum of other well-known theoretical areas of computer science, and their relevance to the daily practical construction of programs.

As practical examples, I examine two cases of coding in restricted, (nearly) total programming systems in attempts to ensure correctness of code, ease of review, and affordance for static analysis. First, as a very specific example, NASA JPL's coding standards for safety critical systems written in C forbid the use of recursion and loops without constant iteration bounds with the source code of the 2011 Curiosity Rover Mars Mission flight software reaching approximately 3 million lines of code [9]. As a second more general example, total programming systems that are susceptible to Blum's Size Theorem are becoming increasingly popular in software development [15, 14].

Praxis in Theory. Wadler [18] and Soare [16] have argued that a critical difference between the work of Turing and Church is that Turing explicitly connected his abstract model of computation to the human experience of calculating, whereas Church does not. Further it was Turing's philosophical argument that finally convinced Gödel that λ-calculus, Turing machines, and the theory of recursive functions are all equivalent and, importantly, satisfactory ways of capturing the notion of 'effective computability'. The philosophical argument that Wadler refers to is Turing's elaboration of the mapping between the tedium of rote calculation by human hand and the operation of the abstract machine.

I take Turing's argument to be making a phenomenological intersubjective case for the relevance of phenomenology in computability theory, or, less abstractly, that foregrounding physical experience, calculating by hand or by instructing a machine, informs the theory of computation. Inspired by Wadler's and Soare's observations that the inclusion of philosophical arguments within Turing's work helped to highlight the relevance of computability to the human experience of calculation, I seek to intervene and inject what I find to be the missing phenomenological intersubjectivity into theoretical results on program size. For, so long as humans write code, we will continue expend more mental and embodied labor to produce larger and larger code bases.

Conclusion. Within these couple of pages I have sketched out an important and overlooked direction of inquiry in the philosophy of programming: phenomenology of program size. Overall my goals in the present work are to (1) draw attention to program size as a fundamental issue in the history theoretical computer science and (2) argue for the relevance of program size in the experience of programming.

References

[1] Manuel Blum. On the size of machines. *Information and Control*, 11(3):257–265, 1967.

[2] Tom Britton, Lisa Jeng, Graham Carver, Tomer Katzenellenbogen, and Paul Cheak. Reversible debugging software "quantify the time and cost saved using reversible debuggers", 11 2020.

[3] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

[4] Robert L. Constable. On the Size of Programs in Subrecursive Formalisms. In *Proceedings of the Second Annual ACM Symposium on Theory of Computing*, pages 1–9. ACM, 1970.

[5] T.H. Cormen, T.H. Cormen, C.E. Leiserson, Inc Books24x7, Massachusetts Institute of Technology, MIT Press, R.L. Rivest, C. Stein, and McGraw-Hill Publishing Company. *Introduction To Algorithms*. Introduction to Algorithms. MIT Press, 2001.

[6] Kurt Godel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover Publications, 1992.

[2] `https://www.cs.cornell.edu/courses/cs6110/2015sp/Lectures/lecture27_6110.pdf`

[7] R. Harper. *Practical Foundations for Programming Languages.* Practical Foundations for Programming Languages. Cambridge University Press, 2013.

[8] M. Heidegger, J. Stambaugh, and P.J. Stambaugh. *Being and Time: A Translation of Sein und Zeit.* SUNY series in contemporary continental philosophy. State University of New York Press, 1996.

[9] G.J. Holzmann. The power of 10: rules for developing safety-critical code. *Computer*, 39(6):95–99, 2006.

[10] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[11] Andrei N. Kolmogorov. On tables of random numbers (reprinted from "sankhya: The indian journal of statistics", series a, vol. 25 part 4, 1963). *Theor. Comput. Sci.*, 207(2):387–395, 1998.

[12] Zohar Manna and Amir Pnueli. Axiomatic approach to total correctness of programs. *Acta Inf.*, 3(3):243–263, sep 1974.

[13] Albert R. Meyer. Program size in restricted programming languages. *Inf. Control.*, 21(4):382–394, 1972.

[14] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*, volume 1 of *Software Foundations.* Electronic textbook, 2021. Version 6.1, `http://softwarefoundations.cis.upenn.edu`.

[15] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at large: A survey of engineering of formally verified software. *Foundations and Trends in Programming Languages*, 5(2-3):102–281, 2019.

[16] Robert I. Soare. Computability and recursion. *Bulletin of Symbolic Logic*, 2:284–321, 1996.

[17] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[18] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, nov 2015.

The disturbance of death and debt

Andre Dickson
Ministry of Trade and Industry, Government of the Republic of Trinidad and Tobago
Republic of Trinidad and Tobago
`accounts@andredickson.com`

In this paper I identify David L. Parnas's "uses" relation in Peter Naur's program death and Ward Cunningham's debt and use the philosophy of Martin Heidegger to argue that even noticing a failure to reuse existing programs requires using a program.

1. "uses" changes While creating good software may often be expensive, all software is easy to ruin. Thus in sustaining the quality of their programs, programmers face resistance in two directions, in creation and maintenance. But what does good mean? In Mythical Man Month Fred Brooks argues that conceptual integrity is 'the most important consideration in system design' and that 'the purpose of a programming system is to make a computer easy to use'[1]. But the idea of conceptual integrity remains difficult to get clear on. As a definition of good and easy, we follow a more pragmatic route offered by Dave L. Parnas in desiring programs that are easy to modify through "uses" relations i.e. by reusing existing data abstractions [9] We call any extension to the behaviour of software that results from the appropriate reuse of existing code written by the programmer as a "uses" change. Next we analyse two accounts of how software can be ruined when one fails to take account of "uses" changes.

2. Program death In *Programming as theory building* Peter Naur states that a 'desired modification may usually be realized in many different ways, all correct' but the 'longer term viability of the program' is still dependent on the 'character' of modifications[7, p.257]. Naur has "uses" changes in mind in his critique of the lacking character of modifications observed in an existing compiler system[7, p. 254]. He argues that software extensions that could have been achieved with existing facilities were performed with 'patches that effectively destroyed [the program's] power and simplicity' (Ibid., p. 254). Naur argues that all changes to an existing program *should* unfold in accordance with a theory of the program, and thus programmers need only do as they already know how to do. For Naur, a failure to exploit "uses" changes signals that a programmer lacks an understanding of the program and marks these events as the 'death of a program' (Ibid., p. 258).

3. Debt and consolidation Ward Cunningham's famous debt metaphor is commonly misconstrued as technical debt and covered up by other categories of quality issues in program text[3, 4]. But debt is a singular phenomena which also hinges on the programmer's failure to exploit "uses" changes[2]. Cunningham argues that the understanding programmers gain in incremental development allow opportunities for improving the existing facilities of a program to show up. However, once programmers introduce new versions of existing functionality they should gradually remove all signs of "deprecated" "uses" changes, a process Naur calls consolidation. For Cunningham, a failure to consolidate 'will make a program unmasterable' and threaten the possibility of future changes (Ibid., p. 30).

4. Useful things and disturbance We now give a philosophical account of use that will make salient issues facing programmers modifying existing programs. In his early works the philosopher Martin Heidegger introduced the category of entities called useful thing which are produced to serve a purpose[5, p.68]. However, a useful thing is holistic in that it always refers to, or *signifies* a reference to, other useful things, and activities. For example a pair of shoes is useful and signifies socks, laces, floors, leather and rubber but primarily it signifies an '*in-order-to*', that is the walking to work, what we are immediately taking care of when responding to everyday situations. Heidegger calls the totality of these signifying references Bedeutsamkeit, which can be translated as meaningfulness, or better significance. Parnas' "uses" relation is such a reference from one useful thing to another, from program text to abstraction that the text 'requires the presence of a correct version of'[9, p.132]. Thus in establishing "uses" relations, "uses" changes setup signifying references between the program text proper and a system of abstractions.

A useful thing can cause a disturbance by not belonging (obstinacy) to this totality of significance. For example, a pair of work boots can be obstinate and disturb if one tries swimming with them. Similarly, Naur's program death disturbs because where an adept programmer expects to see "uses" relations to existing abstractions they encounter "foreign" code. Cunningham's debt disturbs because where a programmer familiar with the text expects to see "uses" relations to new abstractions they encounter "relics" of their previous understanding. But reviewing programs and noticing code as not belonging can only be possible on the basis of a prior understanding of the significance of these "uses" relations in the text [6, p. 188, 243]. Similarly, In making appropriate use of "uses" changes, a programmer must not only bringing forth a potentiality latent in the existing program text, they must also be able to respond correctly to the situation that the demand for change presents[5, p.82]. As Naur points out by way of Thomas Kuhn, program modifications depend not only on knowledge of existing facilities but on an ability to recognize a thematic family of situations in which it would make sense for these facilities to be applied[7, p.255]. As Kuhn argues in his description of paradigms as exemplars, such an ability is acquired not through knowledge of theory but from repeated and guided *use* of theory [8, p.187]. Thus, a programmer can only make appropriate use of "uses" changes in a program on the basis of prior understanding and use of a system of data abstractions.

References

[1] Brooks, Fred P. (1995), The mythical man-month, Anniversary edn. (Reading: Addison-Wesley) pp. 42-43.

[2] Cunningham, Ward (1992), 'The WyCash Portfolio Management System', OOPSLA '92: Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum) 4 (2): pp. 29-30.

[3] Fairbanks, George (2020), Ur-Technical Debt, IEEE Software 37 (4): pp. 95–98.

[4] Fowler, Martin 2009, TechnicalDebtQuadrant, in MartinFowler.com: <https://martinfowler.com/bliki/TechnicalDebtQuadrant.htm

[5] Heidegger, Martin 2010, 'Being and Time', trans. Joan Stambaugh and Dennis J. Schmidt revised edn (Albany: State University of New York Press)

[6] Heidegger, Martin (1992) 'History of the Concept of Time: Prolegomena', trans. Theodore J. Kisiel (Bloomington and Indianapolis: Indiana University Press).

[7] Naur, Peter 1985, 'Programming as theory building', Microprocessing and Microprogramming 15 (5): 253-261

[8] Kuhn, Thomas S. (2012), The structure of scientific revolutions, Chicago: University of Chicago Press.

[9] Parnas, David L. (1979), 'Designing Software for Ease of Extension and Contraction', IEEE Transactions on Software Engineering SE-5(2): p. 128-138.

Diagrammatic writing as writing the non-programmable
A case study from the DRAKON graphical modeling language A parallel glance on the transformations of the computer

Fabien Ferri
Université de Franche-Comté
France
fabien.ferri@univ-fcomte.fr

Abstract. Alan Turing is credited with introducing the notion of the stored program machine, which is at the heart of the theoretical model that underpins computer science [18]. He is also credited with introducing a thought experiment in a paper considered to be the founding paper of artificial intelligence [17]. The issue at stake in this experiment is whether the faculty called intelligence can be separated from the biological support that constitutes living beings. To the question of whether it is possible to reduce biological intelligence to mechanical intelligence, the 1950 article superimposes a second question, concerning the duality of the meaning of determinism, predictive or non-predictive. The challenge of this second question is to understand the relationship between the calculable and the incalculable. If everything that can be written by a program is potentially calculable, what about the nature of what is non-programmable? Where can it be located? Our hypothesis is that the non-programmable is not unwritable insofar as it is diagrammable. Thus, by exploring the content of the notion of the diagram as what is revealed as the complement of the content thought by the notion of the program, we aim to shed light on the nature of what is non-programmable. It is a question of making the non-programmable intelligible in the light of our understanding of what is programmable in the sense of what can be delegated to an effective calculation. We propose to do this by comparing the visual algorithms of the DRAKON graphical modeling language with the choreographic diagrams of the Labanotation [4, 5, 6].

Keywords : diagram, program, DRAKON graphical modeling language.

Although, as some historical studies published in the wake of his centenary have shown, Alan Turing was not the only one to come up with the idea of the stored program machine in the 1930s, since "Zuse had undoubtedly conceived the idea of a digital, binary, program-controlled general-purpose computer by 1936" [7, p. 85], yet it is frequently associated with the introduction of such a notion, which is at the heart of the theoretical model at the foundation of computer science [18]. He is also credited with introducing a thought experiment in a paper considered to be the founding paper of artificial intelligence [17]. The issue at stake in this experiment is whether the faculty called intelligence can be separated from the biological support that constitutes living beings. To the question of whether it is possible to reduce biological intelligence to mechanical intelligence, the 1950 article superimposes a second question, concerning the duality of the meaning of determinism, predictive or non-predictive. The challenge of this second question is to understand the relationship between the calculable and the incalculable. If everything that can be written by a program is potentially calculable, what about the nature of what is non-programmable? Where can it be located?

Our hypothesis is that the non-programmable is not unwritable insofar as it is diagrammable. One argument underlying the thesis that a graphic schematization can be equivalent to a non-programmable and incalculable inscription is the following: the diagram, as soon as it corresponds to the inscription of an operative chain of practical gestures, provides a modelling and simulation medium that offers the reader a degree of freedom and a margin of indeterminacy that a virtual machine running a program does not possess. This is the case, for example, of a choreographic score written in Labanotation [11].

If we reinscribe computer science and artificial intelligence in a long history of technology, they appear as new chapters in the history of writing [12]), as intellectual technologies [16] constituting new ways of thinking and of equipping thought [13, 1]. We can therefore hypothesise that we are on the threshold of a new transformation of reason in its categories of thought [8] insofar as the technical support for inscribing knowledge, as it can currently be analyzed through a class of graphical objects called diagrams, authorizes another way of conceiving artificial intelligence. This other way of conceiving artificial intelligence no longer consists of thinking of it exclusively in terms of the notions of algorithm and calculation. It consists in starting from these semiotic machines that are diagrammatic-type graphic systems, which are both models for informal problem solving and tools that allow us to act better in the world [11].

The diagrammatic approach to writing bases its approach on manipulable discrete units that are already bearers of meaning, constituting the semiotic material of which diagrams are composed [11]. As semiotic machines, diagrams are partially calculating (this is their machine aspect) because they carry a meaning that exceeds their calculatory

dimension (this is their semiotic aspect). The ratio of computation to semiosis (i.e. the production of meaning) in the diagram is inversely proportional: the more computational power one gains, the more semiosis one loses; but the more meaning-bearing semiotic material one gains, the more computability one loses.

Therefore, if knowledge means the ability to carry out an action in order to reach a fixed goal, then a diagrammatic conception of knowledge consists in figuring it through diagrams provided with meaning. This meaning must be the transparency of what these diagrams objectively aim at, in the way that a photograph aims at what it is a photograph of: namely its object. To ask the question of the inscription of knowledge from a diagrammatic conception of writing is therefore to ask the question of the sequencing of knowledge through graphic diagrams provided with meaning. It also means asking the question of the articulation of these diagrams as sequences of operations to be linked in order to carry out a more or less complex task. The representation of this sequence also introduces the question of the grammar that enables the diagrams to be linked to each other to express the meaning of the more or less complex task they represent.

Now, if we restate the problem of schematism from a materialist and diagrammatic perspective by showing that the schema can no longer be considered as a mystery buried in the depths of consciousness [14], then it must be materialized in a medium that allows it to be deployed as a visible schema, the seizure of which operationalizes it in the form of a diagrammatical reading [10]. There is therefore an analogy between the schema/diagram and algorithm/calculation pairs that must be questioned and explored. Indeed, the schema and the algorithm are representations, whereas the diagram and the calculation are temporal realities, i.e. actual executions. What is at stake is the question of invention: reading graphic schemas by executing them diagrammatically, i.e. acting, would it not be analogically actualizing the operative schemas that these diagrams carry?

The interpretation of a diagram does not correspond to the analogue of a computational process, because the diagram, when conceived as the complement of the algorithm, designates a non-computable resolution schema [9]. The diagram is therefore not a temporal figure like the algorithm [2], but a spatial figure, which is not reducible to a geometric figure, insofar as it is a graphic symbol with meaning. If the diagram is neither a temporal figure (since it should not be confused with an algorithm) nor a geometric figure (since it should not be confused with a geometric form), it is because it represents non-calculable operations whose nature must be elucidated. Moreover, a diagram is not a text insofar as, like a map, it allows different levels of information to be superimposed simultaneously [19, 15].

Now, as an algorithm is what drives the execution of a program on this dynamic medium that is the Turing machine, it is a method of resolution that is exploited within the framework of the development of computer science, insofar as this method is executed in a reasonable time interval through a "physics of signs" [3, p. 16]. We are indeed dealing with a mechanics without a substrate, since the computer approach abstracts from the dynamics of matter [2]. Indeed, as a computation is an abstract mechanism (because it is independent of any material or physical substratum) but effective, the notion of mechanical computation makes it possible to found computer science by neutralizing the philosophical problem of temporal infinity. To neutralize such a problem, it considers the notion of temporal infinity given in act as being equivalent to that of a realized temporal interval [2].

By exploring the content of the notion of the diagram as what is revealed as the complement of the content thought by the notion of the program, we aim through this presentation to shed light on the nature of what is non-programmable, in order to make it intelligible in the light of our understanding of what is programmable. We propose to do this using hybrid graphical objects: visual algorithms written in the DRAKON graphical modeling language. This language, invented in 1986 as part of the Russian space programme, was originally intended to govern the artificial intelligence that was to control the Buran spacecraft.

In order to highlight the singular cognitive potential of diagrams, we would like to compare the semiotic grammar of the visual algorithms of the DRAKON language with that of a non-computer diagrammatic notation system such as Labanotation [4, 5, 6]. For a diagrammatic device such as a choreographic score seems to be able to highlight the non-programmable in a non-trivial way, and to make explicit the difference between a visual algorithm and a diagram.

References

[1] Bruno Bachimont. Arts et sciences du numérique : ingénierie des connaissances et critique de la raison computationnelle.

[2] Bruno Bachimont. Herméneutique matérielle et artéfacture : des machines qui pensent aux machines qui donnent à penser. critique du formalisme en intelligence artificielle.

[3] Bruno Bachimont. *Le contrôle dans les systèmes à base de connaissances : contribution à l'épistémologie de l'intelligence artificielle*. Hermès, 2e édition.

[4] Jacqueline Challet-Haas. Grammaire de la notation laban : cinétographie laban.

[5] Jacqueline Challet-Haas. Grammaire de la notation laban : cinétographie laban.

[6] Jacqueline Challet-Haas. Grammaire de la notation laban: cinétographie laban.

[7] B. Jack Copeland and Giovanni Sommaruga. The stored-program universal computer: Did zuse anticipate turing and von neumann? In Giovanni Sommaruga and Thomas Strahm, editors, *Turing's Revolution: The Impact of His Ideas about Computability*, pages 43–101. Springer International Publishing.

[8] Fabien Ferri. Comment et pourquoi le diagrammatique transforme-t-il l'histoire de l'écriture ? 163(4):47–59.

[9] Fabien Ferri. De la pratique mathématique à la philosophie des pratiques. 9(1):97–118.

[10] Fabien Ferri. Matérialiser le schème et dynamiser le schéma : penser et agir par le diagramme. In *Rencontre autour de Bruno Bachimont: la technologie nous fait-elle savoir et penser autrement ?*, number 1009 in Annales littéraires, pages 33–44. Presses universitaires de Franche-Comté, presses universitaires de franche-comté edition.

[11] Fabien Ferri. Science opérative et ingénierie sémiotique : des machines graphiques à la morphogenèse organique.

[12] Antoine Garapon and Jean Lassègue. *Justice digitale : révolution graphique et rupture anthropologique*. PUF.

[13] Jack Goody. *The Domestication of the Savage Mind*. Cambridge University Press.

[14] Immanuel Kant. *Critik der reinen Vernunft*. Hartknoch.

[15] John Kulvicki. Knowing with images: Medium and message. 77(2):295–313.

[16] Pascal Robert. *Mnémotechnologies : une théorie générale critique des technologies intellectuelles*. Communication, médiation et construits sociaux. Hermès science publications Lavoisier.

[17] A. M. Turing. Computing machinery and intelligence. LIX(236):433–460.

[18] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. s2-42(1):230–265.

[19] Marion Vorms. Théories, modes d'emploi : une perspective cognitive sur l'activité théorique dans les sciences empiriques.

Hello World? An Interrogation

Robin Hill
University of Wyoming
USA
hill@uwyo.edu

Abstract. The "Hello World" exercise is interrogated for explanation of its prominence in the teaching of programming and computing. This simple program, which performs no computation, exposes the support environment and sketches a task framework for purposes of pure expedience. The interface that affords the activity of coding works in the same way as the coding of the intended object, and helps to separate the computational from the other aspects of programming.

Introduction A standard pedagogy of programming is the first full source code program "Hello World," one or two lines of output commands that produce the simple message of greeting. Why is *Hello World* so useful? The superficial explanation is clear—to scaffold the experience, to learn the sequence of moves, to gather what is needed and reveal what is to be done. Evidence of its popularity is its Wikipedia page, which offers a romp through 40-odd languages' renditions [2]. The Wikibooks entry yields hundreds [10]. What is the explanation of this popularity?

Possible Explanations

1. The use of *Hello World* is arbitrary, only deployed because others use it.
2. The use of *Hello World* tells us something about the environment of programming.
3. The use of *Hello World* tells us something about programming (the creative part).

To challenge 1, we simply ask how *Hello World* became important enough to merit a Wikipedia page. We ask why it emerged or evolved over the author's years of teaching programming. To support and emphasize 2, we note that the Wikipedia page started around 2001, and has steadily added new languages, demonstrating that HW is indeed a touchstone for programming in general. Even for experts, an unfamiliar IDE is simply a nuisance, an obstacle to running even a correct program. Furthermore, a metric called "Time to Hello World," is used to assess programming interfaces, albeit informally [2]. We continue exploration of possbility 3.

The Setup Adopting a somewhat phenomenological approach, without committing to theories of co-constitution or mediation [5, 4, 1], we examine the first-person experience. In the *Hello World* exercise, we have a student programmer S, facing a keyboard and monitor or equivalent input and output devices. S has an inchoate notion of a program, enough to want to learn how to create one in a language L, from a teacher T. T may be a book, video, or other tutorial material, and L may be a display, scripting, or programming language, reached through an interface such as an Integrated Development Environment (IDE) or just a command line. S may be the same as T. Whereas experienced programmers exploit this exercise to mentally map a new programming environment, we assume a beginner S.

T guides S through this program, which is neither necessary nor sufficient to learning programming, just ready-to-hand. The greeting is arbitrary, "Hello" serving as the entry-level (English language) message in everyday communication. The lesson is required (like "find this reference in the library"), so T coaches S as necessary. While subsequent programming education builds rapidly on it, *Hello World* is not a formal object such as a base case, or a structure of data, or of control, or any kind of semantic object, or any kind of abstraction at all. On the contrary, *Hello World* is utterly concrete, born only of expedience.

The Artifact *Hello, World* is a complete and correct syntactic object that passes through the full code-compile-execute cycle of the development environment presented. For our rank beginner S, it gives rough answers to the questions "What does a program do?" and, at the same time, "How does a programmer do that?" A program runs, that is, action takes place; a result is produced.

The interesting aspect of *Hello World*, however, is that it does no computing. Neither algorithm nor variables take part. What it *does* is to show S a bit about the editor, a bit about program structure, the concept of output, some commands for the invocation of compiler or interpreter, perhaps statement termination. *HelloWorld* delivers nothing for which programs are intended, but rather makes space for programs as intended, in a safe setting.

Similar Exercises Other skills and crafts also have opening activities that run through minimal tasks, free of complication and devoid of useful production—stitching on fabric scraps, proving a simple theorem, bouncing a basketball, playing scales. Some are physical training; some are mental. The results exhibited are how fabric reacts when guided, the springing back of the ball, the notes that follow the fingers; or, more generally— that fabric reacts when guided, that balls bounce, that pressing piano keys sounds musical notes.

All these skills have tools and settings—sewing machine, basketball and court, keyboard—which are introduced in the basic exercises. Note that the tools occupy a separate ontological category from the product. But programming is different in its tools and environment; coding requires an interface built from code. The instrument between the human and the technology is the technology itself. (There may be no other arts, crafts, or professions with such a characteristic since stone tools, except perhaps diamond-cutting.)

Alternatives? Why not something else; why does T pass over other short but engaging candidates for introducing programming? Why not some rudimentary math, like inputting a number to test for a value below zero, or something more hardware-centered, like a display of the system clock time or memory address of an object? All of these make sense, and all may be used in some way to teach introductory programming. But we habitually choose a simple text output, stripped of achievement, because any meaningful result would distract from the framework. (But see [9] for a different view.) The program components presented include neither something that does not quickly end; nor something that needs tending, or interaction; nor a contingency, in code that branches.

The Gains So what gains does the *Hello World* exercise confer? The student S learns not how to code an algorithm that depends on control and data (the ultimate goal), but how to use the tools—and also how to grasp, place, rank, and assemble the tools. And it's a sanity check [2] (a rough measure of correctness), where teacher and student can

meet to verify success. "Hello World" teaches the beginner S that the process is sequential, that a fixed finite set of actions is available, that commands and identifiers must be literally correct, that they must be terminated according to syntax, that any errors mean starting over. *Hello World* hands to S "what you need", roughly, the part of computer programming that's not algorithmic.

When S says, "Oh, I see"—it's a "beginner's gesture of control" [6]. S sees the tools, and S sees how to objectify the tools as tools. In fact, S learns that which constitutes the materiality of code. According to [7], "...the usefulness of the term 'materiality' is that it identifies those constituent features of a technology that are (in theory) available to all users in the same way." In this scenario, that way is the development environment.

Code as a Means and an End As noted above, the strict relation between cause and effect in the coding environment is the same as that relation in the code. The programmer can't make a mistake in a single character in the interface, and can't make a mistake in a single character in the program artifact. The coding of programs is tied up with the running of programs. As Berry puts it, "... software is mediating the relationship with code and the writing and running of it" [1, page 37].

We don't need a bicycle to ride a bicycle and we don't use a garment to sew a gament, and we don't wear a garment as means of sewing. We don't apply musical notes to play scales; we apply our fingers to piano keys. Yet programming is subject to complications imposed by the tools, which are themselves programs. Even the most user-friendly Integrated Development Environment imposes a fixed sequence of steps. References to files and other data structures must follow rules of composition and location. All elements of both (1) the IDE or command line and (2) the object program itself are mechanical, "foreign" to us. The interface that affords coding imposes the same requirements as the coding—a situation rich in interest but free of paradox.

Conclusion *HelloWorld* is a minimal working artifact of coding; it is a complete program in source code. We choose it as an introductory exercise for reasons only of expedience, revealing that even the abstract and rigorous methods and rules of computation are subject to the needs of the here and now. In short, *Hello World* circumscribes the materiality of programming—the interaction with the world that is necessary to accomplish the task, what is necessary to situate the algorithm itself.

HelloWorld surfaces, separates, and subordinates (to background or memorization) the operations of the user interface, while presenting the same mechanical rigor in the interface that will be presented for the algorithmic paradigm. Any answer to the question "What is a program?" inhabits both the tool and the product, making *HelloWorld* not a message to the outer world but a door into the compututational world. "Hello Programmer!"

In short:

- Hello World circumscribes the materiality of programming.
- Hello World reveals that the materiality of coding rests on code.

Further Questions Materiality, in terms of code (and computations), may be revealed by *Hello World*, but it still requires definition. And what about experts? For any level of expertise, *HelloWorld* facilitates a new body of achievement. The experienced student S needs to see how to invoke the compiler, define input and output sources, map the elements of the interface to other expected components, etc., analysis possibly subject to levels of abstraction inherent in the learning process [3]. And a look at exercises of expedience in engineering, where other artifacts that affect the lifeworld are constructed, might shed light on that profession's work [8].

References

[1] David M. Berry. *The Philosophy of Software : Code and Mediation in the Digital Age.* Palgrave Macmillan, 2011.

[2] Wikipedia contributors. "hello, world!" program — Wikipedia, the free encyclopedia", 2022. [Online; accessed 12-February-2022].

[3] Luciano Floridi. *The philosophy of information.* Oxford University Press Oxford, 2011.

[4] Don Ihde. *Postphenomenology and technoscience : the Peking University lectures.* Albany Press, 2009.

[5] Lucas Introna. Phenomenological approaches to ethics and information technology. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, fall 2017 edition, 2017.

[6] B. Lennon. Foo, bar, baz...: The metasyntactic variable and the programming language hierarchy. *Philosophy and Technololgy*, 34:13—-32, 2021.

[7] Paul M. Leonardi. Materiality, sociomateriality, and socio-technical systems: What do these terms mean? how are they different? do we need them? In Paul M. Leonardi, Bonnie A. Nardi, and Jannis Kallinikos, editors, *Materiality and Organizing: Social Interaction in a Technological World*. Oxford University Press, 2012.

[8] Glen Miller and Carl Mitcham. *Designing and Constructing the (Life)World: Phenomenology and Engineering*, pages 175–189. Springer International Publishing, 2020.

[9] Ralph Westfall. Technical opinion: Hello, world considered harmful. *Commun. ACM*, 44(10):129–130, 10 2001.

[10] Wikibooks. Computer programming/hello world — wikibooks, the free textbook project, 2022. [Online; accessed 24-April-2022].

Towards a linguistic conception of programming

Daniel Kuby
Universität Konstanz
Germany
daniel.kuby@uni-konstanz.de

What are programming languages? The standard, ready-made answer found in dictionaries and textbooks can be condensed in the following **machine-centered account of programming languages**: *Programming languages are formal notations for writing algorithms in a machine-readable form.* This description highlights the computational purpose of programming languages in a machine-centered way because it views the notation's principal goal as allowing the specification of a set of computations to be carried out by a machine.

The widespread adoption of this account explains in turn the standard language-as-metaphor view, according to which the term "language" is, at best, a useful metaphor or is, at worst, misguided when used in reference to programming languages [30, 31, 2]. While human languages are a naturally arising phenomenon which attends to the many purposes of human-to-human communication, programming languages are highly artificial constructs designed to aid human-to-machine "communication", which, again, is just a *façon de parler* designating the act of inputting a set of algorithms into a machine. "Writing", "communication", "language"—they are all more or less felicitous linguistic metaphors to conceptualize programming languages in terms of something quite different, namely, human languages [7].

The present paper will argue that, while the machine-centered account of programming languages is useful to understand the programming of *machines*, it is insufficient and indeed flawed when trying to understand the design and the programmer's use of programming *languages*. In particular, the machine-centered account does not provide an adequate understanding of major changes which occurred when code "became language" in the 1950s [33]. It is insufficient and flawed because it does not account for a major shift which the process of linguistification enabled: the invention of programming languages made programming (more) human-centered.

Broadly speaking, the usage of linguistic expressions allows to exploit a medium, namely written language, which is a cultural achievement mastered by humans for millennia [35, 17, 18, 34, 38]. The design of programming languages exploits the cognitive affordances of established languages, both natural and constructed, to aid human-to-human communication in programming. As a famous programming textbook quips: "Programs must be written for people to read, and only incidentally for machines to execute" [1, xxii]. A machine-centered account of programming languages cannot provide an analysis of this development because there is arguably nothing about the introduction of linguistic expressions which aids machines to better "understand" the operations they are supposed to carry out. On the contrary, most choices in the design of programming languages and in the writing of code are human-centered, i.e. their existence can only be understood with reference to the human programmer.

Condensing these very broad considerations, the present paper wants to defend the following **linguistic account of programming languages**: *Programming languages are artificially restricted human languages which exploit the cognitive affordances of both human and formal languages in order to aid human-to-human communication.*

From the vantage point of this linguistic account, I will call into question the language-as-metaphor view and will be advocating taking the term "languages" in "programming languages" seriously in that they are languages in more than a metaphorical sense. In doing so, I will draw from the philosophy of language to contribute to the philosophy of technology and computing [8, 9]. To be sure, I claim that the originality of this account is to be measured with regard

to the field of philosophy only; indeed, it reflects a view in the practice of computing and computer science which is as old as the development of higher-level programming languages themselves [3, 43, 41, 23]. The 'obviousness' of this account from the perspective of practitioners in computing should count as evidence that this account gets something right about the practice.

By building on considerations by Ludwig Wittgenstein [42, §2,§8,§18] on "primitive" languages and Gottlob Frege [16, Vorwort] on formal languages, I will introduce the concept of *restricted languages* to characterize programming languages. Both philosophers contributed—in very different ways—to characterize a specific type of linguistic phenomenon: Some languages are (mildly or severely) restricted with respect to full-fledged natural languages; but they do not, as a consequence of their restriction, participate less in languagehood; and their restriction is driven by their specific function(s). Clearly, Wittgenstein discusses examples occurring in ordinary language and Frege discusses an example in the artificial language context. But the similarity and complementarity of these accounts is striking and will motivates my argument that language restriction mechanisms allow to conceptualize the gap between natural from artificial languages as a continuum. More specifically, I argue that restriction mechanisms arising in natural language activities are repurposed in the creation of artificial languages. (On this account, then, the concept of restricted language does not provide a *differentia specifica* to characterize programming languages, but rather a *genus proximum*, as it were; tentatively, what distinguishes programming languages from other restricted language kinds is their purpose, which in turn drives their specific design.)

I will conclude by previewing some upshots which obtain if the elaboration of my proposal succeeds (and a lot of work remains to be done to this effect!). First, it provides a philosophical framework to capture programming languages in computing practices. Second, it gives a philosophical foundation to approaches in media and cultural studies ("software studies", "critical code studies") which have conceptualized source code as textual artifacts by exploiting a much wider *semiotic* perspective [21, 25, 20, 38, 26, 6, 4, 12, 27, 13]. Third, I will briefly discuss a major obstacle to the linguistic account: the absence of programming languages as an object of investigation in linguistics (isolated but recurring attempts in this direction include [28, 32, 14, 11, 10, 40, 29]). I will assess this state of affairs and point to promising but very early research on programming in cognitive linguistics [15, 5, 19] and empirical linguistics [22, 24, 36, 37, 39] and discuss how the concept of restricted languages could deliver a bridge to research in applied linguistics.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. Electrical Engineering and Computer Science Series. MIT Press [u.a.], Cambridge, Mass., 2. ed edition, 1996.

[2] Paul Adamczyk. On the Language Metaphor. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, pages 121–128, Portland, Oregon, USA, 2011. Association for Computing Machinery.

[3] Ben Allen. Common Language: Business Programming Languages and the Legibility of Programming. *IEEE Annals of the History of Computing*, 40(2):17–31, 2018.

[4] Inke Arns. Code as performative speech act. *Artnodes*, 4:1–10, 2005.

[5] Marina Umaschi Bers. Coding as another language: A pedagogical approach for teaching computer science in early childhood. *Journal of Computers in Education*, 6(4):499–528, 2019.

[6] Kevin Brock. *Rhetorical Code Studies: Discovering Arguments in and around Code*. University of Michigan Press, Ann Arbor, 2019.

[7] Carsten Busch. *Metaphern in Der Informatik: Modellbildung — Formalisierung — Anwendung*. Deutscher Universitätsverlag (DUV), Wiesbaden, 1998.

[8] Mark Coeckelbergh. *Using Words and Things: Language and Philosophy of Technology*. Routledge, New York, 1st edition edition, 2017.

[9] Mark Coeckelbergh. *Using Philosophy of Language in Philosophy of Technology*. Oxford University Press, 2020.

[10] John H. Connolly. Context in the Study of Human Languages and Computer Programming Languages: A Comparison. In Varol Akman, Paolo Bouquet, Richmond Thomason, and Roger Young, editors, *Modeling and Using Context*, Lecture Notes in Computer Science, pages 116–128, Berlin, Heidelberg, 2001. Springer.

[11] John H. Connolly and D. J. Cooke. The pragmatics of programming languages. *Semiotica*, 2004(151):149–161, 2004.

[12] Geoff Cox and Alex McLean. *Speaking Code: Coding as Aesthetic and Political Expression*. The MIT Press, Cambridge, Massachusetts, 2013.

[13] Robert Cummings. Coding with Power: Toward a Rhetoric of Computer Coding and Composition. *Computers and Composition*, 23(4):430–443, 2006.

[14] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. Empirical Language Analysis in Software Linguistics. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 316–326, Berlin, Heidelberg, 2011. Springer.

[15] Evelina Fedorenko, Anna Ivanova, Riva Dhamala, and Marina Umaschi Bers. The Language of Programming: A Cognitive Perspective. *Trends in Cognitive Sciences*, 23(7):525–528, 2019.

[16] Gottlob Frege. *Begriffsschrift, Eine Der Arithmetischen Nachgebildete Formelsprache Des Reinen Denkens.* Verlag von Louis Nebert, Halle, 1879.

[17] Hartmut Günther and Otto Ludwig, editors. *Schrift und Schriftlichkeit / Writing and its Use*, volume 1. Walter de Gruyter, Berlin ; New York, 1994.

[18] Hartmut Günther and Otto Ludwig, editors. *Schrift und Schriftlichkeit / Writing and its Use*, volume 2. Walter de Gruyter, Berlin ; New York, 1996.

[19] Ziva R. Hassenfeld, Madhu Govind, Laura E. De Ruiter, and Marina Umashi Bers. If You Can Program, You Can Write: Learning Introductory Programming Across Literacy Levels. *Journal of Information Technology Education: Research*, 19:65–85, 2020.

[20] N. Katherine Hayles. Print Is Flat, Code Is Deep: The Importance of Media-Specific Analysis. *Poetics Today*, 25(1):67–90, 2004.

[21] Michael Heim. *Electric Language : A Philosophical Study of Word Processing.* Yale University Press, New Haven, 1987.

[22] Anna A Ivanova, Shashank Srikant, Yotaro Sueoka, Hope H Kean, Riva Dhamala, Una-May O'Reilly, Marina U Bers, and Evelina Fedorenko. Comprehension of computer code relies primarily on domain-general executive brain regions. *eLife*, 9:1–24, 2020.

[23] Donald E. Knuth. *Literate Programming.* Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, Calif., 1992.

[24] Yun-Fei Liu, Judy Kim, Colin Wilson, and Marina Bedny. Computer code comprehension shares neural resources with formal logical inference in the fronto-parietal network. *eLife*, 9:e59340, 2020.

[25] Mark C. Marino. Critical Code Studies. *Electronic Book Review*, 2006.

[26] Mark C. Marino. Reading Culture through Code. In Jentery Sayers, editor, *Routledge Companion to Media Studies and Digital Humanities*, pages 472–482. Routledge, New York, 2018.

[27] Yukihiro Matsumoto. Treating Code As An Essay. In Andy Oram and Greg Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*, pages 477–481. O'Reilly, Sebastopol, CA, 2007.

[28] Baptiste Mélès. Approche philologique des langages de programmation. *Techniques et sciences informatiques*, 35(2):237–254, 2016.

[29] Linda D. Misek-Falkoff. The new field of 'Software Linguistics': An early-bird view. *ACM SIGMETRICS Performance Evaluation Review*, 11(2):35–51, 1982.

[30] Peter Naur. Programming Languages, Natural Languages, and Mathematics. *Communications of the ACM*, 18(12):676–683, 1975.

[31] Peter Naur. Programming languages are not languages – why 'programming language' is a misleading designation. In *Computing: A Human Activity*, pages 503–510. ACM Press, USA, 1992.

[32] Fariha Naz and Jacqueline E. Rice. Sociolinguistics and programming. In *2015 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 74–79, Victoria, BC, Canada, 2015. IEEE.

[33] David Nofre, Mark Priestley, and Gerard Alberts. When Technology Became Language: The Origins of the Linguistic Conception of Computer Programming, 1950–1960. *Technology and Culture*, 55(1):40–75, 2014.

[34] Uwe Oestermeier. Von der Erfindung der Schrift bis zu logischen Kalkülen. In Uwe Oestermeier, editor, *Bildliches und logisches Denken: Eine Kritik der Computertheorie des Geistes*, Studien zur Kognitionswissenschaft, pages 19–37. Deutscher Universitätsverlag, Wiesbaden, 1998.

[35] David R. Olson. *The World on Paper: The Conceptual and Cognitive Implications of Writing and Reading.* Cambridge University Press, Cambridge, UK, 1994.

[36] Norman Peitek, Janet Siegmund, Sven Apel, Christian Kastner, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and Andre Brechmann. A Look into Programmers' Heads. *IEEE Transactions on Software Engineering*, 46(4):442–462, 2020.

[37] Chantel S. Prat, Tara M. Madhyastha, Malayka J. Mottarella, and Chu-Hsuan Kuo. Relating Natural Language Aptitude to Individual Differences in Learning Programming Languages. *Scientific Reports*, 10(1):1–10, 2020.

[38] Daniel Punday. *Computing as Writing*. Univ Of Minnesota Press, Minneapolis, 2015.

[39] Janet Siegmund, Norman Peitek, André Brechmann, Chris Parnin, and Sven Apel. Studying Programming in the Neuroage: Just a Crazy Idea? *Communications of the ACM*, 63(6):30–34, 2020.

[40] E. B. Spratt. Programming Linguistics. *Nature*, 219(5161):1399–1399, 1968.

[41] Kumiko Tanaka-Ishii. Semiotics of Computing: Filling the Gap Between Humanity and Mechanical Inhumanity. In Peter Pericles Trifonas, editor, *International Handbook of Semiotics*, pages 981–1002. Springer Netherlands, Dordrecht, 2015.

[42] Ludwig Wittgenstein. *Philosophische Untersuchungen / Philosophical Investigations*. Wiley-Blackwell, Chichester, West Sussex, U.K. ; Malden, MA, rev. 4th ed edition, 2009.

[43] Heinz Zemanek. Semiotics and Programming Languages. *Communications of the ACM*, 9(3):139–143, 1966.

Why semantics are not only about expressiveness: The reactive programming case

Nicolas Nalpon
Equipe informatique interactive, ENAC
France
nicolas.nalpon@gmail.com
Alice Martin
Equipe informatique interactive, ENAC
France
alice.martin@enac.fr

Short abstract

Extended abstract

Context: Juggling with both reactive and non-reactive instructions. There are many reasons to formalize the semantics of programming languages, in order e.g., to explain how a language having complex technical features works (e.g. ProbZelus semantics [3]) or to prove the soundness of the compilation [8]. Programming languages have evolved, increasingly building complex abstractions and thereby easing the programming of new systems. In return, the study of programming languages has become increasingly challenging since we have to formalize richer semantics and use more complex mathematical frameworks. Different semantics use different mathematical frameworks and can be equally expressive but still diverge in terms of *semantic representation*, which also means they diverge in terms of *knowledge representation* [11, 6, 7]. We would like to reflect on that issue and argue that it deserves attention, using the case of *reactive programming*. We show that different semantic representations do not convey the same *epistemic* understanding of a system. This can turn into a dilemma when faced with the choice of a semantic for a *reactive program*, which describes *two types of instructions* that do not easily fit into a single knowledge representation.

We refer to the reactive paradigm [2, 10], abstracting programs as reactions to external events and automatically managing computations dependencies. To program a reactive system, one needs to express both classic *non-reactive instructions* (e.g. "2+2") AND *reactive instructions* (e.g. reacting to a "click"). To show the epistemic difference between two expressively equivalent semantics, we decide to study two pseudo languages: a non-functional reactive language and then a functional one. The former can find its formalized semantics in the bigraphs theory [9] and the second with the λ-calculus [1] — two equivalent semantics in terms of expressiveness (Turing completeness). We compare how each semantics is able to express a reactive and a non-reactive instruction. We show the semantic representations offered by each theory differ in what they make explicit.

The epistemic difference between two semantic representations is not just a trivial observation. We think that in the context of reactive programming, the choice of a semantics for a reactive program makes the question meaningful: a reactive program presenting two types of instructions of different nature (computational and reactive), what choice of semantics to make?

1 Case 1 : Reaction to a "click" input.

Let us express a reactive instruction, using two different frameworks.

$\langle a \rangle ::= ei \mid pi$

(a) Pseudo non-functional reactive language: variables.

$\langle c \rangle ::= \langle a \rangle \rightarrow \langle a \rangle;\ \langle c \rangle \mid \langle a \rangle \rightarrow \langle a \rangle;$

(b) Pseudo non-functional reactive language: propagation operator.

```
e1 -> p1;
p1 -> p2;
```

(c) Reactive program example.

Figure 1: Pseudo non-functional reactive language and reactive progam example.

1.1 Case 1 with Bigraphs representation

The pseudo non-functional reactive language described in Fig. 1 allows us to manipulate interactions between two abstractions called *events* and *processes*. An event, denoted `ei`, represents an external stimulus (e.g mouse click, keypress) and is considered activated when the stimulus is triggered. A process, denoted `pi`, represents a computation which is computed once the process is activated. Processes can be activated only if an event or another process propagates its activation through the operator `->`. Accordingly, the program of Fig. 1c has as semantics: if event `e1` is activated then the process `p1` is computed and then `p2` will be computed.

Bigraphs consist of two relation types over the same set of entities. One relation represents (topological) space in terms of containment through nesting and the other expresses (non-spatial) relationships among entities through hyperedges (non-binary links). Thus, each entity shown as shapes, has a control (type) that determines its arity (number of links). For example in Fig. 2a, control `e1` representing the event `e1` has arity 0 and control `p1` representing the process `p1` has arity 0. Entities may nest other entities, e.g. in Fig. 2a entities `Src` and `Trg` have arity 1 and are nested in entity `p1` representing respectively its input and output port. The operator `->` can be represented through the green hyperlinks as shown by the connection between entities `Src` and `Trg` in Fig. 2a.

A bigraph represents the state of a system at a single point of time. A Bigraphical Reactive System describes how bigraphs evolve over time using reaction rules $r : L \longrightarrow\!\!\!\!\blacktriangleright R$ that specify bigraphs matching bigraph L can be replaced by bigraph R in some larger bigraph. Fig. 2b represents such rules allowing us to define how the activation of an event (or process) is propagated. Thus, in Fig. 2a, if an entity `Act` (representing the activation of an event or a process) is contained by `e1` then rule 1 from Fig. 2b can be applied and propagates the activation to `p1`, and in the same way, rule 2 will propagate the activation to `p2`.

With bigraphs, our reactive program involves only **2 execution steps**: 2 *reactions* are needed to propagate the activation of `e1`. Such semantic representation can be either algebraic or diagrammatic (Fig.2). The semantic representation is explicit about the reaction in both representations: the binding between events is explicitly represented by the name of the edges in the algebraic notation (below).

$$e1.(src\{l1\} \mid trg\{l2\}) \mid p1.(src\{l2\} \mid trg\{l3\}) \mid p2.(src\{l3\} \mid trg\{l4\})$$

$$e1.(src\{l1\} \mid trg\{l2\} \mid act) \mid p1.(src\{l2\} \mid trg\{l3\}) \longrightarrow e1.(src\{l1\} \mid trg\{l2\}) \mid p1.(src\{l2\} \mid trg\{l3\} \mid act)$$

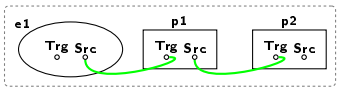
$$p1.(src\{l1\} \mid trg\{l2\} \mid act) \mid p2.(src\{l2\} \mid trg\{l3\}) \longrightarrow p1.(src\{l1\} \mid trg\{l2\}) \mid p2.(src\{l2\} \mid trg\{l3\} \mid act)$$

1.2 Case 1 with a $\lambda-$calculus representation.

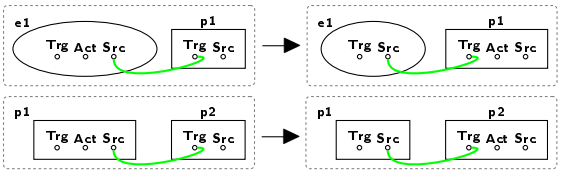
With a functional language, events (respectively processes) of the reactive program in Fig. 1c can be represented as a function $\lambda a.e1a$ (respectively $\lambda a.p1a$). To represent the waiting phase of an event or a process, the function takes an activation a as a parameter. Then, this parameter is given to the function $e1$ (respectively $p1$) which is going to be reduced into another activation $e1a \longrightarrow_{computation} a$ (respectively $p1a \longrightarrow_{computation} a$). The operator $\rightarrow$ used previously is translated as an application of a function. Hence, the program in Fig. 1c yields the following semantic representation:

$$\lambda a.p2(p1(e1a))$$

The following reduction chain happens if $e1$ is triggered :



(a) Bigraphical program representation.



(b) Reactions rules.

Figure 2: Case 1. Diagrammatic representation with bigraphs theory.

1. $\lambda a.p2(p1(e1a))a$ e1 is triggered
2. $\longrightarrow_{\beta} p2(p1(e1a))$
3. $\longrightarrow_{computation} p2(p1a)$
4. $\longrightarrow_{computation} p2a$ $p1$ is computed
5. $\longrightarrow_{computation} a$ $p2$ is computed

Here, there are **4 execution steps**: 4 "β-reductions". In terms of knowledge representation, such an abstraction, even if it allows the properties of a reactive system to be expressed, abstracts from the concept of reaction. The binding supporting the connection between events is left implicit.

2 Case 2 : Function reduction.

When the instruction to express is non-reactive, the results are the opposite. The functional language and its semantic representation within the frame of the lambda calculus express the instruction in a non-verbose manner. On the contrary, the non-functional reactive language struggles to express the instruction and requires way more computation steps.

$\langle M \rangle ::= x \mid MM \mid fun\ x\ \rightarrow M$

(a) Pseudo functional language.

```
(fun x -> x) x
```

(b) Functional program example.

Figure 3: Pseudo functional language and functional progam example.

The pseudo functional language presented in Fig. 3 is identical to the pure λ-calculus and its β-reduction.

2.1 Case 2 with a $\lambda-$calculus representation

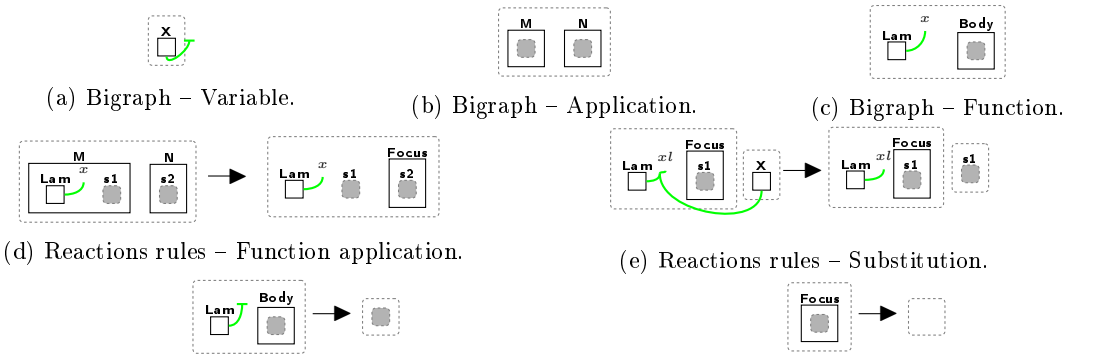
The representation and the execution of Fig. 3b are straightforward using the λ-calculus. Program in Fig. 1c yields the following semantic representation and only **one execution step** is needed to execute it.

$$(\lambda x.x)x \longrightarrow_{\beta} x$$

2.2 Case 2 with Bigraphs representation

This subsection uses special constructs allowing bigraphs to be built (and decomposed) compositionally. Dashed rectangles represent regions, while filled grey rectangles represent sites. Sites contain parts of the bigraph that have been abstracted away, i.e. they contain an unspecified bigraph (including the empty one). To represent the λ-calculus through the bigraphs theory, we associate to each element of the λ-calculus a corresponding bigraph and define rewriting rules to reproduce the behavior of the β-reduction (Fig. 4).

Representing the λ-calculus using bigraphs is a bit tricky because of the merge operator from the bigraphs theory. This operator allows building larger bigraphs horizontally by placing regions side-by-side. It is associative and



(a) Bigraph – Variable.

(b) Bigraph – Application.

(c) Bigraph – Function.

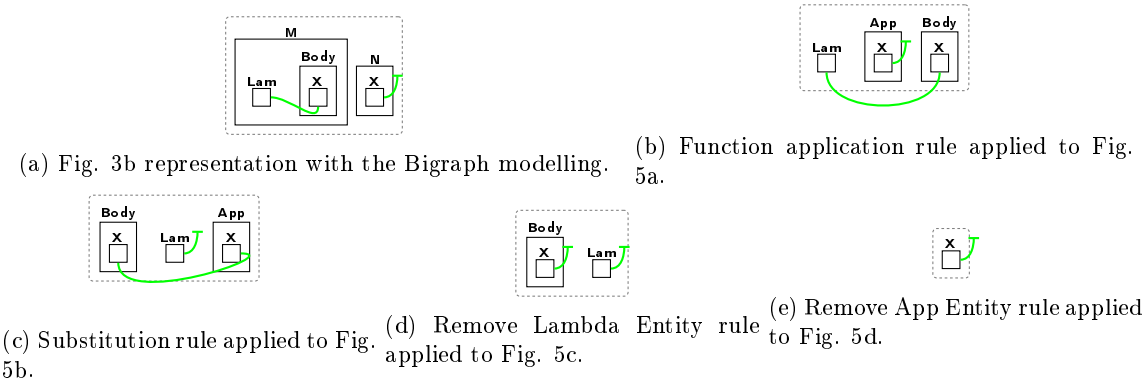
(d) Reactions rules – Function application.

(e) Reactions rules – Substitution.

(f) Reactions rules – Remove Lamda Entity.

(g) Reactions rules – Remove Focus Entity.

Figure 4: λ-calculus via the Bigraphs theory.



(a) Fig. 3b representation with the Bigraph modelling.

(b) Function application rule applied to Fig. 5a.

(c) Substitution rule applied to Fig. 5b.

(d) Remove Lambda Entity rule applied to Fig. 5c.

(e) Remove App Entity rule applied to Fig. 5d.

Figure 5: Execution of the functional program example (Fig. 3b).

commutative. Hence, to represent function or application of function from the λ-calculus, we have to define extra controls indicating the part of a sub-bigraph in a bigger bigraph while, in the λ-calculus, these are encoded by the location of a sub-term in a λ-term. For example in Fig. 4b, we define two controls M and N indicating respectively the left and the right side of an application. Similarly in Fig. 4c, we define a control Lam (as lambda) representing the binder of a variable x and a control $Body$ representing the body of the function. Because of these extra structures, we encode the β-reduction rule using several rewriting rules and therefore increase the number of execution steps.

Rule in Fig. 4d is a preliminary step to the β-reduction. It matches an application of function, removes the extra structures and focuses on the right part of the application, which is the parameter, using the control $Focus$. This step is directly followed by the application of rule in Fig. 4e. This rule matches a function binder linked to a variable and the focused function parameter. It finally substitutes the variable x by the focused bigraph parameter. This rule is applied successively as long the function binder is linked to a variable. Rules in Fig. 4g and Fig. 4f are used to remove the extra structures used to accomplish the function reduction.

Fig. 5 describes the execution of program in Fig. 3b. Here **4 execution-steps** are needed to compute the program.

3 Proposing an epistemic criterion

Representing a computation AND a reaction does not find an easy expression within a single semantics (dedicated to the description of step-by-step computation like the lambda-calculus, or dedicated to the description of reacting events and processes like bigraphs). We do not want to draw attention to the trivial observation that one necessarily loses something by describing one mathematical language in another. More precisely, we want to draw attention to the dilemma that arises here when the goal is to propose semantics for a reactive program (Fig. 1c). The difficulty arises from the fact that p1 or p2 triggered respectively in reaction to e1 and p1, can themselves be computations.

And this intermingling of computational and reactive aspects is inevitable. How can these two aspects be reconciled in the same semantics and knowledge representation? Expressing reactive AND non-reactive instructions is more or less tedious given a chosen semantic representation. It looks like a single candidate can hardly express both. The fact that a semantic representation can more or less understandably describe a system is an epistemic criterion. It has nothing to do with standard criteria of expressiveness, such as Turing completeness. Our concern is similar to Krivine's [1]. That does not mean that it is a mere qualitative criterion. A way to capture it, we suggest, could be evaluated in terms of execution steps. There is more at stake than a conceptual debate. Stating that a reactive language does not find a single unified semantics satisfying both the representation of reactive and non-reactive instructions makes *hybrid* semantics look worth considering. Therefore, we hypothesize that reflection on semantic representations could in return inspire more understandable semantics and easier sound proof writing.

If we look at what exists, our criterion can identify two types of frameworks and languages for reactive systems. We can indeed identify those that choose to explicitly distinguish between the reactive and non-reactive parts (e.g., React_native [4]) and those that do not (JavaFx/Swing). In the same spirit, some reactive languages choose to choose between the use of imperative state machines or dataflow, and some choose to mix them. Which choice favors the programmer's understanding of the system while facilitating verification of the semantics? One could hypothesise that a hybrid semantics by minimising the number of steps (using a computational framework for a semantics dedicated to computational instructions and an interaction theory to ground a semantics dedicated to reactive instructions) would make the program more understandable (for the programmer) and modular (for the convenience of the proof maker). The advantages and disadvantages of hybrid semantics are discussed [5]. We would at least like to point out that this is a growing challenge for programming complex reactive systems today. We hypothesise that this situation and debate are indicative of an important criterion for programming languages and the knowledge representations they presuppose: an epistemic criterion, which cannot be reduced to expressiveness.

References

[1] Roberto M. Amadio and Pierre-Louis Curien. *Domains and Lambda-Calculi*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

[2] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. 45(4):1–34.

[3] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. Reactive Probabilistic Programming. In *International Conference on Programming Language Design and Implementation (PLDI)*, London, United Kingdom, June 15-20 2020. ACM.

[4] Adam Boduch and Roy Derks. *React and React Native*. 2020.

[5] Jean Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In *Proceedings of the 5th ACM International Conference on Embedded Software, EMSOFT 2005*, 2005.

[6] Paul Humphreys. *Extending Ourselves: Computational Science, Empiricism, and Scientific Method*. 2006.

[7] Jean Krivine. From Molecules to Systems: the problem of knowledge representation in molecular biology, 2019.

[8] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. page 14.

[9] Robin Milner. *The space and motion of communicating agents*. 2009.

[10] Guido Salvaneschi and Mira Mezini. Debugging for reactive programming. In *Proceedings - International Conference on Software Engineering*, 2016.

[11] David Waszek. Informational Equivalence but Computational Differences? Herbert Simon on Representations in Scientific Practice. In *Proceedings - 6th International Conference on the History and Philosophy of Computing*, Zurich, 2021.

Miniatures, Demos and Artworks: Three Kinds of Computer Program, Their Uses and Abuses

Warren Sack
University of California, Santa Cruz USA
wsack@ucsc.edu

[1] Krivine compares the semantic representation of formal languages applied to molecular interaction networks [7]

Abstract. Miniatures, demos and artworks (MDAs) are three kinds of computer program produced in universities, research laboratories, and the art world. Primarily they are created for teaching, arguing, selling, and sparking the imagination to show a larger public what is possible for the future of software. The most well-known miniature is Minix developed by Andrew Tanenbaum (Vrije Universiteit Amsterdam) to teach students about the workings of the Unix. "The mother of all demos" was produced in 1968 by Douglas Engelbart (SRI) introducing the computer mouse, resizable windows, teleconferencing, hypertext, word processing, and collaborative editing – and basically everything we have today in personal computing. The best-known software artwork is probably Spacewar!, a computer game created by Steve Russell in collaboration with several others at MIT in 1962. In this talk I will describe how MDAs are similar and different from one another, have been used and abused in imagining the future of software. Examples include those from the Yale AI Project of the 1980s (where the design of miniatures was a vital concern); demos from the MIT Media Laboratory of the 1990s (an agonistic environment of "demo or die"); and contemporary artworks from the Whitney Museum of American Art, NYC.

Miniatures, demos and artworks (MDAs) are three kinds of computer program produced in universities, research laboratories, and the art world. Primarily they are created for teaching, arguing, selling, and sparking the imagination to show a larger public what is possible for the future of software. The most well-known miniature is Minix developed by Andrew Tanenbaum (Vrije Universiteit Amsterdam) to teach students about the workings of the Unix. "The mother of all demos" was produced in 1968 by Douglas Engelbart (SRI) introducing the computer mouse, resizable windows, teleconferencing, hypertext, word processing, and collaborative editing — and basically everything we have today in personal computing. The best-known software artwork is probably Spacewar!, a computer game created by Steve Russell in collaboration with several others at MIT in 1962. In this talk I will describe how MDAs are similar and different from one another, have been used and abused in imagining the future of software. Examples include those from the Yale AI Project of the 1980s (where the design of miniatures was a vital concern); demos from the MIT Media Laboratory of the 1990s (an agonistic environment of "demo or die"); and contemporary artworks from the Whitney Museum of American Art, NYC.

MDAs have overlapping attributes and characteristics that distinguish them from each other and from the computer programs used by governments, business, and consumers as reliable, everyday tools and platforms. They can be elaborated, via prototyping, into a “version zero” from whence the theories and methods of software engineering and human-computer interaction can be employed to develop specifications, begin user-needs studies, to model a system, and then to organize a team or teams of programmers and HCI professionals to create a product, open-source or otherwise. While MDAs may be an iterated step of an iterative development cycle, they are usually just the first step, the research step, of a research and development process. MDAs are almost always created by just one or a small group of people over an appreciable period of time and so, in terms of design, they are prior to the step of "ideation" when larger groups come together to share ideas in real time. MDAs are philosophical statements inscribed as computer programs.

Miniatures

At the Yale Artificial Intelligence Project of the 1970s and 1980s, after completing the dissertation, doctoral students had one more gauntlet to run before they were done: to write a miniature version of their dissertation program. Programs written for the dissertations were typically thousands or even tens of thousands of lines of code. To rewrite a system in miniature entailed rearticulating the central mechanisms of the full-sized version in just hundreds of lines of code. To shrink the code down two orders of magnitude in size, the doctoral student sometimes collaborated with Christopher Riesbeck, who was then a research scientist at Yale. These miniatures were then published in a series of books, the first of which was Inside Computer Understanding: Five Programs Plus Miniatures (Lawrence Erlbaum, 1981).

As Roger Schank and Christopher Riesbeck [3] explain in their preface, "We have written this book for those who want to comprehend how a large natural language-understanding program works. ... As a part of the curriculum we designed for them, we created what we called 'micro-programs'. These micro-programs were an attempt to give students the flavor of using a large AI system without all the difficulty normally associated with learning a complex system."

Some of these miniatures, or "micro-programs" still influence the systems developed today. For example, James Meehan's 1976 dissertation program, Tale-Spin, a story generator. Richard Evans, the AI designer for The Sims 3 game has explained that the techniques developed for Tale-Spin were used, decades later, to drive the characters in The Sims.

Demos

It is useful to understand all contemporary software demos – even though beyond the confines of big data – as both abductive and rhetorical. Demos are created as arguments, usually arguments for more resources or more time. They are built to see what can be possible to do in the future if more time and more money are devoted to this project. Douglas Engelbart gave his mother of all demos in 1968, but a commercially viable product incorporating those technologies – like the Apple Macintosh – did not arrive until almost 15 years later. Engelbart's demo worked but required years of design and engineering work to make it into a product. Demos – even those like Engelbart's that were not data based – are arguments about the future and so belong to the art of rhetoric. They are abductive because the audience is convinced by the performance and not through an audit of the technology: the audience must fill in the missing pieces to be convinced.

Artworks

As Sol LeWitt wrote in his "Paragraphs on Conceptual Art," Artforum (Summer 1967), "When an artist uses a conceptual form of art, it means that all of the planning and decisions are made beforehand and the execution is a perfunctory affair. The idea becomes a machine that makes the art." Making machines as a form of making art has become an important approach to software art as illustrated by artist Casey Reas's Software Structures artwork for the Whitney Museum of American Art (`https://artport.whitney.org/commissions/softwarestructures/map.html`). Reas, Jared Tarbell, Robert Hodgin, and William Ngan recoded/reinterpreted several of LeWitt's wall drawings as twenty-six different JavaScript programs.

Conclusion

Miniatures are pedagogical, demos persuasive, and artworks pleasurable. They are all relatively small but potentially complex creations of the lab, the studio, and the university. They can all be easily distinguished from deployed, working systems, but can exert tremendous influence over their design. As such, making MDAs is a practice of writing computer programs to imagine and influence the future of software.

References

[1] Christian Paul (2015), Digital Art, Third Edition, Thames & Hudson.

[2] Warren Sack (2019), The Software Arts, MIT Press: Cambridge, MA.

[3] Roger Schank and Christopher Riesbeck (1981), Inside Computer Understanding: Five Programs Plus Miniatures, Lawrence Erlbaum.

Programming practice as a microcosmos of human labor and knowledge relations

Shoshana Simons
University of California, Berkeley USA
`shoshana_simons@berkeley.edu`

This presentation argues that the practice of computer programming has a co-constitutive relationship with broader human labor and knowledge relations and considers this relationship's implications for computer science pedagogy. From understanding a program's semantics to measuring its runtime, the "planetary network" of labor, humans, and materials enabling its functioning is consistently invisibilized to the programmer (Crawford et al 2018). The program is fetishized as an apple is at the grocery store (Chun 2008). Writing a computer program involves trust, division of labor, and blackboxing, like handing off tax documents to an accountant (Shapin 1994). Word, in particular written word, mediates the programmer's relationship to materiality (Galloway 2012). The programmer the commander of word (think "command line"), the machine the slave, secretary, and soldier of embodied labor (Benjamin 2019). **Drawing on course materials from computer science institutions across the United States, this presentation establishes a few ways in which the practice of programming embeds (i.e. structurally**

maps) into broader human labor and knowledge relations as evidence for the co-constitutive or co-productive (Jasanoff 2004) **relationship between them.** More specifically, this embedding suggests that computer programming is the programmer's **microcosmic training ground** for working within broader knowledge and labor relations under capitalism. Along the way, the presentation offers an **understanding of ethics** as fundamentally inseparable from technical practice; **computer programmers learn broad moral habits from their technical practice in that they enact it metaphorically** (Hofstadter 2001). Understanding the ethics of programming is not done through blackboxing it (Latour 1987) but rather examining it in practice (Daston et al 2007). Finally, the presentation explores the potential of **re-configuring programming practices for reconfiguring broader economic and epistemic human relations**. In particular, the presentation puts forward a few pedagogical suggestions for computer science, from incorporating a more materialist vision (Murphy 2017) into teaching semantics and runtime, to integrating critique into the practice of programming.

References[1]

Benjamin, Ruha. 2019. Race After Technology: Abolitionist Tools for the New Jim Code. Polity.

Chun, Wendy Hui Kyong. 2008. On "Sourcery," or Code as Fetish. Configurations. Vol 16. No 3.

Crawford, Kate. Joler, Vladan. 2018. Anatomy of an AI System. SHARE Lab.

Daston, Lorraine, Galison, Peter. Objectivity. Princeton University Press. 2007.

Galloway, Alexander. 2012. The Interface Effect. Polity.

Hofstadter, Douglas. 2001. Analogy as the Core of Cognition. MIT Press.

Jasanoff, Sheila. 2004. States of Knowledge: The Co-production of Science and the Social Order. Routledge.

Latour, Bruno. 1987. Science in Action. Harvard University Press.

Murphy, Michelle. 2017. Alterlife and Decolonial Chemical Relations. Cultural Anthropology. 32, no. 4: 494-503.

Shapin, Steve. 1994. A Social History of Truth. University of Chicago Press.

[1] A reference does not imply that the idea was drawn from the author but rather a resonance between what is said here and what they have written.

Practical information

Location The symposium takes place in *Espace Baietto* at *MESHS*,

MESHS
2 Rue des Canonniers
59000 Lille

The MESHS is right across the two main stations of Lille.

Lunch The lunch is not included in the registration. There are a lot of places around MESHS where one can grab a sandwich or have a small lunch.