HISTORY AND PHILOSOPHY OF LOGIC, 2015
Vol. 36, No. 3, 278–301, http://dx.doi.org/10.1080/01445340.2015.1059992

Hardware, Software, Humans: Truth, Fiction and Abstraction

GRAHAM WHITE
Queen Mary University of London, London
g.graham.white@gmail.com

Received 14 January 2015 Accepted 22 March 2015

We start with a example of assembler programming, and show how even at this low level the structure of the programming language does not directly mirror the structure of the hardware, but that it is also decisively influenced by the human practices surrounding computer use, and that assembly language gives a view of the hardware which is accommodated to human interests and capabilities. We give several historical examples and illustrate the changing pattern of mutual accommodation between human practices and computer technology, and argue for a more explicitly dialectical and critical approach to the history and philosophy of programming.

1. Prelude: the history of programming

There is a dominant preconception in the study of computation, which holds that the essentials of the design of computer hardware are specified as something like Turing machines, but that, in contrast, the design of computer software (from programming languages on up) allows for much more variation and choice: it is almost as if hardware is part of nature, whereas software is produced by culture. We will argue that this preconception is misleading: that both sides of the divide are equally cultural. Our main argument will be historical: we will aim to show that the evolution of both sides is far more intricate than people generally suppose, and that, correspondingly, the interactions between the two sides are far more complex than one would imagine.

First, though, we should start with something on the history of programming.[1]

1.1. The first programming languages

In the early days of computers, programming was done either by manipulating plugs and wires, or by directly writing machine instructions in binary: both of these were difficult and error-prone. Programming languages were invented to allow programmers to write in a more comprehensible form: even in the 1940s, languages were developed which allowed programmers to use abbreviated names for commands rather than binary machine code (*Wheeler 1992, Haigh et al. 2014b*). Such languages are now known as *assembler languages*, and they represent the least abstract level of programming language design.

Fortran dates from 1954, and made it possible to write programs in a notation very like that of standard mathematics (*Backus 1981*). Although the Fortran designers paid very little attention to theory—Backus, the leader of the project, says 'we simply made up the language as we went along' (*Backus 1981*, p. 30)—both syntax and semantics soon became important factors in the design of programming languages. More abstract languages followed: Algol was designed over the period 1958–1960 (*Naur 1981*, *Perlis 1981*), Lisp from 1958 to 1962 (*McCarthy 1981*), and many of the difficulties of developing these languages were due to two factors: it was difficult to define the syntax of a language at

[1] More detail on this side of things, especially a description of the philosophy of programming languages, is given in *White 2004*.

all precisely, and the semantics of these languages seemed utterly mysterious. The latter was an extremely serious problem: without some sort of semantics, it was hard to say what counted as a correct implementation of these languages. Although Lisp eventually achieved a precise semantics, it was designed by starting from the implementation and then attempting to find mathematical structure in the resulting language: several of the Lisp primitives were called after hardware features of the machine that it was originally implemented on (*McCarthy 1981*, p. 175), whereas its original semantics was 'ramshackle' (*Landin 2000*). Nevertheless, it was also possible to see that the rewards for a precise syntax—or, more ambitiously, a precise semantics—were extremely high: Algol 'proved to be an object of stunning beauty' (*Perlis 1981*, p. 88).

What, then, do these programming languages look like? We will describe a generic language, quite similar to Algol; since Algol has had an enormous influence on language design its features can be found in many others. These languages have some similarity to formal languages like first-order logic: like these languages, they have variables, to which values can be assigned, and they have both predicates and functions. And many of the basic operations of programming can be viewed as the assignment of values to variables, so one might think that these operations, too, could be viewed in this way.

There are, however, differences, of which the most important one is that programming languages allow programs to perform actions that *change* the values of variables, or which have other irreversible effects (e.g. input or output); we say that these actions have *side effects*.

1.2. The importance of semantics

Possibly the most important fact about the language—hardware relation is that it is subject to norms: programming languages, of course, are normed simply because they have syntax, but, on the other hand, hardware too should be built according to a design—what is sometimes called an 'architecture'—and the language should be designed to fit the architecture, rather than merely to work on a particular physical implementation of the hardware architecture. Otherwise, one risks having one's software depend on accidental, unintended properties of the hardware: Brooks describes a historical case during the development of IBM hardware:

> In an unpoliced system all kinds of side effects may occur, and these may have been used by programmers. When we undertook to emulate the IBM1401 on System/360, for example, it turned out that there were 30 different 'curios' – side effects of supposedly invalid operations – that had come into widespread use and had to be considered as part of the definition [of the IBM1401 hardware].[2]

This two-sided normativity makes the hardware–software relation much more complex than is generally supposed.

2. Three surprises

We start our investigation of the hardware–software relation by introducing the conceptual issues. The author came to appreciate them as a consequence of teaching assembly language in an introductory computer architecture course: this section could, then, be described, by those with a tolerance for such terminology, as autoethnography.

We are told—in fact, if, like me, 'we' teach computer architecture, we *tell* people—that computer languages can be either high level or low level, and that the low level ones,

[2] *Brooks 1995*, Chapter 6.

specifically the various assembly languages, all reflect more or less directly the configuration of the hardware. Furthermore, you might expect the hardware to implement something very like a von Neumann architecture, or maybe even a Turing machine (apart from the obvious concessions to finiteness), so that you would also expect that the fundamental architecture of a computer would not have changed much since the early days: thus, for example, Ceruzzi (*2000*) writes

> The basics of [the UNIVAC] design remained remarkably stable during the evolution of computing from 1945 to 1995. Only toward the end of this period do we encounter significant deviations from it, in the form of 'massively parallel' processor or 'non-von Neumann' architectures.

So it comes as a bit of a shock when, in such teaching, our students get to know a particular assembly language—I am thinking specifically of MIPS32 (*Farquhar and Bruce 1994*)—and write programs which are sequences of instructions, and they are tempted to believe that the hardware will execute the instructions in the order that we write them in, but no: if there is a branch instruction, which tells the computer, depending on the equality or otherwise of two numbers, to resume execution from some other instruction it will not execute the branch *right then* but only when the instruction after the branch has been executed (*Hennessy and Patterson 2014*, A-59) (This is because a branch takes a comparatively long time to execute, so the computer might as well have something to get on with while it does so.).

So that, then, is surprise 1. Now students, in such a situation, generally learn assembler by working on a simulator, rather than by actually executing the code on the appropriate hardware: and surprise 2 is that, with the simulator,[3] you have a choice of whether it gives you this rather unintuitive behaviour or not. So you can choose between a simulator which simulates the hard-to-learn behaviour of the real hardware, or the somewhat easier behaviour of fictitious hardware. Clearly, the question of 'what is the real hardware', or 'what is the real low level', is not as straightforward to answer as we, or our students, might like to think.

We shall be arguing in this paper that this phenomenon is quite typical: at the low level of assembly programming, you might expect that human factors are rather minimally in play, if they are in play at all, and that the programming language somewhat directly mirrors what the hardware does. But not so: humans who work with computers at this level are presented with a carefully achieved view of what the hardware is doing, and it would be an oversimplification to imagine that this view was a direct presentation of the hardware. To some extent this is inevitable: modern computer hardware, and especially central processing unit (CPU) chips, is extraordinarily complex, most of it is designed by computer programs, and nobody can have any sort of overview of what the CPU actually does. Furthermore, the behaviour of modern computers is nondeterministic and chaotic, so that we cannot predict their behaviour, at least in the short term. If you deal with this nondeterminism in its own terms (measuring the statistics of hard disc access, caching and the like), then you are an electronic engineer and not a computer scientist: *Hennessy and Patterson 2012* will give you a good idea of what this entails.

But there is also a role for computer scientists, namely the people who design and implement algorithms, design higher level programming languages, database systems, and the like. For this, one needs a rather more abstract view than the one given by electronic engineering. As it happens, the programming models that computer scientists use (students,

[3] Specifically, this one http://spimsimulator.sourceforge.net/

obviously, but also professionals) present computers to them as something vaguely like Turing machines: however, in reality, the computers themselves are not Turing machines, but something much stranger.

The ability to negotiate this translation between the hardware and the programmer's idealisation of it is one of the things that makes computer science a useful profession. For a current example of this, we can consider multicore processors: these are genuinely concurrent machines, and the standard problems of concurrency arise (avoiding deadlock, making sure that the processors' views of data are consistent with each other). Most programmers find genuine concurrent programming extraordinarily difficult, so the ideal would be to design a programming language in which one could program concurrent machines as if they were standard, sequential machines. This is probably unachievable: these are genuinely hard issues. However, dealing with them—by finding programming tools with which normal programmers can program concurrent machines—is probably the key to having conceptual models of modern computers that humans can, in some way, program.

Which brings us to surprise 3. We might like to think that the sophisticated intellectual background behind modern programming languages took a long time to arrive, just as it took a long time to get from small, old, slow hardware to modern, fast, capable hardware. Furthermore, we think, the initial state was one of great simplicity, and was firmly rooted in hardware which in turn implemented the fundamental ideas derived from Turing's work. Now a great deal of the early work on computers was certainly very hardware-focussed—for example, Mauchly's description of the EDVAC was basically a description of the machinery, with some indication of the functional roles of its parts (*Bergin 2014*)—but, as we find from *Priestley 2008*, *Haigh 2015*, *Daylight n.d.* and *Nofre et al. 2014*, a great deal of that intellectual infrastructure was in place very early, by the 1950s or early 1960s at least, Turing was in some ways peripheral to the early developments, and the initial state of computation was hardly simple (although as *De Mol and Bullynck 2012* document, the *aspiration to* simplicity was part of the intellectual milieu in which computer science was developed).

What we have, then, is a system with three components: the human designers and programmers, the hardware, and the software and assembly languages that present a view of the hardware to the programmers. It would be tempting to think of the relationship between these components as rather simple (programmers write programs, software translates programs into machine code, and the hardware runs the code), and it would also be tempting to think of these relationships as fixed and somehow given a priori (presumably they sprang fully formed from the head of Turing). It is, on the contrary, a system that is constantly under strain, mostly due to ongoing rapid progress in technology coupled with the rather stable nature of programming language design. Consequently, the picture that emerges is of continuous frantic improvisation in order to present the same (or nearly the same) view to programmers while the underlying hardware is constantly in flux.

I shall be arguing for an egalitarian picture of these relationships: that is, that none of the components is completely constrained by the others. For example, Galloway (*2011*, p. 64) writes that 'digital software-hardware environments, including robust and successful ones, are … likely to be underdetermined with respect to their potential use'; there are always 'tacit usage skills' in a community of users, which makes it desirable to conduct '[e]thnomethodological observation of expert users and their systematic "misuses" or adaptations of programs or environments' (*Galloway 2011*, p. 64). The difficulty becomes more acute when the context of use lies in the past, when 'there is almost never a perfect meshing between the environment in which a digital object is rendered today and as it was created in another environment' (*Galloway 2011*, p. 64).

Innovation in such mixed systems does not pass only in the direction from designers to users: for example, Tin (*2011*, pp. 75f) describes how, in Taiwan in the 1980s,

> users tinkered with newly introduced microcomputer technology and participated in shaping the physical make up and social meaning of microcomputers.

Similarly, Ensmenger (*2009*, p. 88) describes software maintenance as 'as much a social as a technological endeavour. Usually what needs to be "fixed" is the ongoing negotiation between the expectations of users, the larger context of use and operation, and the features of the software system in question'. Interestingly, much of this maintenance is a '[response] to changes in the business environment'; so the problems that maintenance deals with are not problems solely in the hardware or software components of the system, but can also originate with changes in the human environment. Furthermore, the innovation required can often be a change in existing social or cultural systems. Russell, for example, describes how open standards, such as the Open Systems Interconnection model, were introduced in conscious critique of the existing closed, proprietary culture of development and implementation: 'ISO's identity and structure embodied a critique of the existing order in which market actors were anything but equal partners' (*Russell 2012*, p. 78).

Conversely, the improvisation and adaptation can present a new view to programmers on existing hardware: Haigh et al. (*2014a*) describe how Adele Goldstine, von Neumann and others reconfigured the hardware of the ENIAC—a machine which was built before more 'modern' machines such as EDVAC or the Manchester Baby—so that it 'emulated (to use an anachronistic term) the von Neumann architecture' (*Haigh et al. 2014a*, p. 50). The term 'emulate' is certainly anachronistic, and the agents of this reconfiguration would doubtless have viewed it as merely a piece of opportunistic improvisation: but nevertheless it opened up a distinction between the hardware *an sich* and how it was presented to programmers,[4] a distinction which was to become ever more important as computing developed.

A fortiori, I shall definitely not be viewing the universal Turing machine as an idea which dropped from heaven, and which was then triumphantly implemented in both hardware and the design of programming languages. The prevalence of Turing machines in textbooks of computability is precisely that: it is the permanence of an abstract model of computation with a well-developed metatheory. Furthermore, Turing had more ideas than that, those other ideas were also influential, and ideas other than Turing's were influential. The connection between computers and universal Turing machines was not appreciated by many of Turing's contemporaries (*Priestley 2008*, *Haigh 2015*). And, although the idea of the Turing machine has had a considerable influence on, for example, language design, modern computers are not in any simple way Turing machines, as we will come to see later.

Furthermore, this history is very important: none of these phenomena can escape history. For all of his well-deserved influence, modern programming languages and programming practice deviate quite markedly from Turing's own ideas: for example, Turing was extraordinarily attached to the idea of self-modifying code (*Priestley 2008*, Section 3.4), an idea of the attractiveness of which is hard to appreciate from the current perspective, and which was, in any case, radically circumscribed in von Neumann's design (*Haigh et al. 2014b*).

3. Methodology

There will be a historical component to this argument, and my own picture of this history has been much influenced by the work of Mark Priestley and his colleagues: see, for

[4] Gobbo and Benini (*2013*) have a rather attractive semi-formal account of this distinction.

example, *Priestley 2008*, *Haigh 2015*, *Daylight n.d.*, *Nofre et al. 2014* and *De Mol and Bullynck 2012*. My own methodology, however, will deviate from Priestley's: I shall be somewhat influenced by Garfinkel's ethnomethodology (*Garfinkel 1967*), and, in particular, his picture of social action as being continually constituted by the active intervention of its participants.

What Haigh et al. (*2014a*, p. 42) describe as 'user-driven innovation' can be usefully regarded as such active intervention, although the ethnomethodological perspective is somewhat wider: user-driven innovation is rooted in the design community, whereas ethnomethodologists would regard these processes of constitution as being pervasive in human interaction, whether the activity was one of design or not. However, I will not be directly using ethnomethodological methods: for one thing, many of my examples are historical and not open to direct research, and, for another, my underlying model, with the three components of people, software and hardware—or, to be pedantic about agency, of computer users and programmers, of software designers, and of electronic engineers—is different from, and somewhat more complex than, Garfinkel's. (In particular, the role in this improvisatory process of one of these groups—namely the electronic engineers—is not solely driven by the constitution of their own viewpoint but by enabling the computer programmers and users to constitute theirs, that is, to arrange matters so that users and programmers can continue as before.) However, the underlying intuition is very much the same: namely that the coherence of concepts and practices can only be sustained by the continuous active intervention of the actors in the process.

I will, to a great extent, be telling this story from the point of view of programmers, rather than electronic engineers, and so I will be treating the hardware in a somewhat unanalysed fashion (in particular, I will neglect the extent to which the electronic engineers' view of hardware is itself an abstraction). This neglect is for a variety of reasons: life is finite, for one, I understand the software side much better, for another, and, finally, if you want information on how the design perspective looks from the hardware side you can get a good idea from *Hennessy and Patterson* (*2012, 2014*), which are wonderful resources. I shall also cite *Stallings 2013* to cross-check what Hennessy and Patterson say.

Some of the explanation—of, for example, the semantics of variables—might seem to be elementary from the point of view of practising computer scientists, but these explanations should be read as phenomenology, not as science: these concepts may, from a professional viewpoint, seem elementary, but they are neither necessary nor obvious. They have been constructed historically, and they are not the only conceptual structure that one could use to found a discipline of electronic computation.

For information on community practice—the community being that of programmers and designers of hardware and software systems—I shall rely to some extent on textbooks for an account of the community view of particular technologies: in particular, I shall rely on *Hennessy and Patterson 2014* for an account of the current state of hardware and of good practice in assembly programming. I have found Wikipedia useful for information on the community view of these matters (the relevant articles in Wikipedia are quite often written by members of the community), although, because of the conventions of journal articles, Wikipedia is very difficult to cite.

In the following sections, the predominant argument will be to develop examples which undermine the following constellation of ideas: that there is a unidirectional set of influences from hardware to software to user practice, that low-level languages directly mirror hardware, that the constraints that hardware design has to accommodate come solely from physics and engineering, and that there are no normative concepts at work in low-level descriptions of computation.

There will be a number of pedantic footnotes, which the reader may safely skip.

4. The semantics of numbers

I shall take it that at least some programs are written in order to implement algorithms, and that, correspondingly, they have a semantics: numerical variables and constants in a program will correspond to numbers in the algorithm, and, depending on the algorithm, arithmetical operations can be applied to them, they can be tested for equality, and so on. Elucidating this semantics is part of the more general task of defining a mathematical semantics of programs (see *Priestley 2008*, Section 4.9; *White 2004*); this semantics is mathematically complex, but I shall mostly be concerned here with an informal concept along the lines of 'what the programmer intends when they are writing the program' (an informal concept which could, in principle, be elucidated by asking the programmer what, for example, a particular variable represents, or what a particular command does to a particular variable).

We should notice, however, that the 'numbers' which are the semantic values of variables may not actually have the standard mathematical properties which we expect. For example, the IEEE standard for floating point numbers (*IEEE Computer Society 2008*) defines the following entities:

(1) a finite set of (representations of) floating point numbers: these correspond one to one with a finite set of mathematical real numbers, except that $+0$ is not the same as -0,
(2) $+\infty$ and $-\infty$ (not equal to each other),
(3) NaN, which stands for 'not a number'.

We expect there to be only a finite set, because, after all, computers are finite. The non-equality of $+0$ and -0 is quite strange, and breaks some of the fundamental properties of numbers as we know them.[5] The two infinities are included because we want these numbers to be closed under arithmetic operations, performed under a computer: consequently, $\frac{1}{0}$ should evaluate to $+\infty$, and $-\frac{1}{0}$ should evaluate to $-\infty$. Similarly, $\frac{0}{0}$ should evaluate to NaN.

This seems a little odd, especially because this representation contains something which describes itself as 'not a number'. Furthermore, integers are generally represented in computers *without* the difference between $+0$ and -0. Why the difference? Number representations are never uniquely determined: in practice, choosing the correct one will depend on choosing which representation optimises factors such as speed and memory use.[6] Nowadays, in comparison with earlier periods, memory is very cheap,[7] so, as far as number

[5] For example, that if $x + y = x + z$, then $y = z$: taking here $x = 1$, $y = +0$, and $z = -0$, we have $x + y = x + z = 1$, but $y \neq z$.

[6] One should note that the situation for the designer deciding on number representations is somewhat complex, and more or less as follows. Modern hardware implements unsigned binary integers, with the standard operations. Given this, there are basically three choices for the representation of signed integers – sign and magnitude, ones' complement, and twos' complement – and, of these, twos' complement is, by general consensus, optimal and is the one that is used. Floating point arithmetic is generally performed in special-purpose hardware: there are many choices which could be made, but there is a standard (*IEEE Computer Society 2008*) which is widely implemented in that special-purpose hardware, which is taken for granted by large software libraries, and which consequently is solidly entrenched. It is not universally followed, however: the programming language Java had rather non-standard floating point implementations, and there has been over a decade of wrangling in the Java community about what to do about it (see *Java Numerics 2013*).

[7] Especially in comparison with the very early days, when, as Haigh et al. *2014b*, p. 6 write,

> Memory technologies were, as these comments remind us, the dominant challenge facing computer builders in the late 1940s. Discussions of drums, delay lines, Selectrons, cathode ray tubes, wire recorders, and phosphor discs occupied a central place in the first computing texts and conferences.

representations go,[8] optimising memory use is not a decisive criterion. Speed, however, is. Now any particular choice of number representation will speed up some calculations and slow down others. So, in practice, optimising number representations depends on finding out which calculations are performed most (*Hennessy and Patterson 2014*, p. 11): it is, in part, an empirical matter. And (with commonly used programs as they are usually written) the most frequent operations with integers are not the same as the most frequent operations with floating point numbers. To be specific, one of the most common operations with floating point numbers is evaluating inequalities, that is, finding whether one number is less than another or not.[9] So floating point representations are chosen so that, if we regard the two floating point numbers as simply sequences of zeroes and ones, and if we regard those sequences as (unsigned) integers, then the two floating point numbers will have the same inequality relations as the 'underlying' integers, and evaluating the integer inequality will be faster (*Hennessy and Patterson 2014*, p. 199). But this representation has, as a side-effect, the inequality of the representations of $+0$ and -0; this has side effects elsewhere, of course, but the price is generally regarded as worth paying.

The strange item `NaN` is easy to explain if we realise that it is the result that you get if you attempt an operation on numbers which does not produce a numerical output (dividing zero by zero, adding $+\infty$ to $-\infty$, and the like). That is, it stands for things that ought to be numbers, but which are not: it is thus a *normative* concept.

4.1. Accommodations

So, we have two surprising results here: one is the occurrence of normative concepts among integer representations, and the other is the dependence of the design decisions on the human activity of writing programs (in particular, on the question of what floating point operations are executed the most by the programs that humans generally write).

We can see, here, all three components of the system at work. The hardware gives us implementations of basic operations: (which may or may not include floating point). Software uses these operations, and software is written in response to human needs. The process of optimising these systems feeds back from software to hardware: if your floating point is implemented in hardware, then that hardware will be designed according to constraints which come, in part, from human needs, and also, in part, from the characteristics of programs that humans write. Similarly, the `NaN` items are traces of human normative concepts (we would like our calculations to give answers) implemented in hardware.

5. Variables and memory addresses

One of the key ideas in programming is the idea of a variable: there are basically two stories to tell about it, one being about the idea of a variable *in general*, the other being the idea of a variable in assembly language programming. We will tell the first story first, because these ideas we will be more familiar to the modern audience: nevertheless, there are certain distinct features of variables in assembler, which will also describe.

5.1. In general

In modern terms, a *variable*[10] is an object which has a name and which can contain a value (*Watt 1990*, pp. 37f). Variables can receive values, or have their values changed, in

[8] Of course, memory use can be a factor when we are dealing with large data structures distributed in memory: see *Chilimbi 2001*.

[9] By contrast, we want to test whether integers are equal to zero, rather than testing whether one is less than another. The difference arises because the most frequent computations made are the ones that we use to test whether a loop has terminated or not: with integer calculations, these tests are done by evaluating equalities, whereas with floating point calculations we do it with inequalities, because floating point calculations are susceptible to rounding errors, and, consequently, equalities between floating point numbers will very rarely hold: we test, then, for *approximate* equalities, which comes down to testing inequalities.

[10] To be precise, a variable in a declarative programming language such as C or Java.

assignment statements, which are usually written with an equality sign, thus:

```
x = x + 1.
```

These variables have the following properties, which distinguish them from mathematical variables as generally understood.

(1) Variables have *scope*: they are declared at a particular point in the program, which will in general be inside a programming construct (a loop body, a method body, and so on). Variables remain in existence so long as they remain in scope.

So, we can make the following distinction. Consider a single location *m* in memory at two different points in the program: there are two cases:

(a) The location holds the value of a variable *x*, which is in scope at both points.

(b) There is no variable which is in scope at both points whose value is stored at the memory location.

(2) Variables can change their values: for example, the *assignment statement* `x = x+1` increases the value of the variable *x* by 1.

(3) Different variables can stand for the same value. Note that, if we have two variables with the same value, we can meaningfully ask whether that value is stored in the same area of memory: we can just change the value of one of the variables and see whether the value of the other changes.[11]

Correspondingly, Java (for example) has two ways of testing the equality of variables. `x.equals(y)` will test whether the values are the same: `x == y` will test whether the values are the same and are stored in the same place in memory. So Java has an abstract concept of memory locations, and can test for their equality: C, which is rather more low level, allows access to addresses in memory, which Java does not do. These properties motivate the common description of a variable as 'an area of memory with a name attached': of course, one can give more or less abstract semantics to the idea of an area of memory, so it is not as low level a description as it might appear to be.

We should contrast these properties with two other ways of calculating with values: Turing machines considered in themselves (what we may call *bare Turing machines*), and variables in mathematics.

Bare Turing machines do not have these properties. Turing machines certainly have tapes, and locations in the tape store values, which can be read or written. But locations of the tape do not have *names* (in fact, although the locations are arranged in order along a tape with a beginning but no end, and so implicitly can be numbered, even these numbers are not available to the machine, which only knows about tape movements relative to its current position). Furthermore, although a Turing machine can read and write to the tape, there is no concept of variable scope, and neither is there a concept of updating *the same entity* which we have when we update a variable. If a Turing machine writes to a tape location, this could correspond to updating a value in the algorithm which the Turing machine program implements, but it could equally well be the replacement of a temporary value with another unrelated temporary value. And there is no way that we could simply look at the hardware and find out which of these possibilities had occurred.

[11] We are here ignoring the difference between values on the stack and values on the heap (in Java terms, between primitive types and reference types).

Of course, we could implement, on a Turing machine, a programming language which could do all this: but the implementation would be able to do these things because it was an implementation of a programming language with the appropriate semantics.[12]

Variables in programming languages are also not the same as variables in mathematics, since they can be updated, and mathematical entities, whatever they are, do not change. These equality signs do not stand for mathematical equalities: they are not symmetric, for example (the assignment statement `x + 1 = x` does not make sense, because `x + 1` is not a variable, although `x` is).

5.2. The history of the concept of variables

How did we get here? This is quite a complex body of theory and practice, and its history has not really been written. However, we can say a few things.

One possible source for the idea of updating variables is the practice of numerical analysis, that is, the systematic calculation of solutions to mathematical equations. Much of numerical analysis is concerned with progressive approximation to, for example, the value of an integral, or to an irrational number: one would start with a rough approximation and then progressively improve it, using something like Newton's method. Although numbers are mathematical entities, and thus cannot change, there is a very good sense in which one can talk about changing, or updating, an *approximation* to a number; one can see this at work in old textbooks of numerical analysis, which frequently have quite detailed instructions about how to lay out such calculations on a page (*Hartree 1952*, *Southwell 1940*).[13] Locations on paper would here be the precursors of areas of memory; in the case of relaxation methods, the values in a table row would be the values successively held by a memory location.

Now early electronic computers were widely used for carrying out numerical calculations of this sort (*Priestley 2008*, *Nofre et al. 2014*)—indeed the early computation groups consulted established numerical analysts such as Hartree, who himself was quick to recognise the value of electronic computers (*Medwick and Mahoney 1988*)—and the practice of numerical analysis was, in fact, referred to by early computing researchers: Howard Aiken and Grace Hopper write, describing the design of an early computer, that

> The development of numerical analysis ... [has] reduced, in effect, the processes of mathematical analysis to selected sequences of the five fundamental operations of arithmetic: addition, subtraction, multiplication, division, and reference to tables of previously computed results. The automatic sequence controlled calculator was designed to carry out any selected sequence of these operations under completely automatic control. (*Aiken and Hopper 1946*, p. 386, cited in *Priestley 2008*, p. 92)

and so it is not inconceivable that the idea of a variable was at least partly influenced by the practice of numerical analysis.

[12] Namely those described in *O'Hearn and Tennent 1997a,b*.

[13] For example, see *Hartree 1952*, p. 9

> Numerical work should not be done on odd scraps of rough paper, but laid out systematically and in such a way as to show how the intermediate and final results were obtained: and the numbers entered on the work sheet should be written neatly and legibly. Use of ruled paper is a help in keeping the layout of the work neat and clear.

And *Southwell (1940*, p. 10, Table 2) has a table showing a number of numerical values which are to be calculated, and, for each of them, the sequence of arithmetic operations which yield progressive approximations to that value.

We should also notice that numerical analysts would (then, and probably subsequently) talk of numbers *changing*, by which they meant not the way that a time series could represent the changing value of a physical quantity, but the way that successive approximations to a *single* numerical value would change. Thus, Hartree writes

> a change in the estimate of $\delta^2 y'_0$ by ε makes a change $\frac{1}{12}(\delta x)^2$ in y_1,

and Wilkes similarly writes

> it will be assumed that the quantities y_0 and y_1 are held in the store of the machine in 'storage locations' numbered 100 and 101 respectively, and that storage location 102 contains a number η. η will change as the calculation proceeds, and will finally become equal to y_2.[14]

Thus, we have the following set of ideas: 'numbers' are what memory locations contain, and these entities can change.

And, in fact, early computers had quite a lot of support for performing computations of this sort. This is hardly surprising, since, as we have seen, one of their main uses was for numerical analysis. In particular,[15]

(1) In 1943, Mauchly outlined a solution of the ballistic equations on the ENIAC, in which each variable had a fixed assignment to a particular register.
(2) Many early machines had addressable storage locations, and (in 1947 or so) programmers such as von Neumann and Goldstine used to describe the situation relationally: they would say that the value of a particular mathematical expression was stored in a particular location. This was hard and confusing to do.
(3) By 1948 this relational description had fallen out of use, and in 1949 Turing explicitly equates variables and storage locations (*Turing 1989*). He also has a dashed notation for distinguishing the values of a variable before and after an operation.

So, in some form or another, these ideas emerged very early, possibly guided by the practice of numerical analysis. Algol, which dates from 1960, is generally recognised as the first programming language which clearly and perspicuously applies these principles. Even so, it took some time after that for a good formal semantics for these languages to emerge; *O'Hearn and Tennent 1997a,b* has the details and many of the papers in the evolution of the formal semantics, from the definition of Algol 60 onwards.

We should also remark, rather parenthetically, that early computer researchers also used the mathematical idea of variables: for example, Turing, in 1936, described the implementation of algorithms in Turing machines by means of a notation which he called 'machine tables', or, for short, *m*-tables. These tables would describe the behaviour of the machine when it was presented with a particular symbol on the tape and a particular internal configuration: the behaviour was represented by means of mathematical functions, and these functions could contain free variables which would hold the value of the currently scanned symbol. So the idea of the variable—in the *mathematical* sense—certainly enters here.

5.3. *Memory addresses in assembly language*

Computers typically hold numbers in memory, and memory locations have addresses, which are numbers (in this respect, computers differ from Turing machines). But computers also hold instructions in memory, so these instructions also have addresses. At any rate since EDVAC, instructions and numbers have been held in memory in the same way (*Haigh et al. 2014b*).

[14] Both citations are from *Campbell-Kelly* (*1992*, p. 21).
[15] I am indebted here to Mark Priestley, who, in a private communication, provided most of the following facts.

These two sorts of addresses are used in the following way:

(1) If we want to perform an operation, such as addition, on data, then we will need the addresses of the arguments and the address where the result is to be stored.

(2) If we want to execute an instruction, then the CPU must know the address of that instruction, so that it can fetch it and then execute it. Normally, computers execute instructions in a sequential order, simply moving from one instruction to the next in sequence, and this can be done by simply incrementing the address that one fetches instructions from. However, there are circumstances in which one wants to execute instructions out of order: for this, one needs to know explicitly the address of the instruction that has to be executed next.

This set of ideas—that is, that storage locations are persistent, and that they have addresses—is very early: for example, Williams describes how Newman explained to him and Kilburn that 'numbers could live in houses with addresses' (*Copeland 2011*, p. 22).

Instruction addresses can be problematic: the main problem is that we may not know, when writing code, what addresses its instructions are going to have. This may happen if we are writing a subroutine, designed for repeated execution in conjunction with other code. Problems then arise with the various forms of out-of-order execution.

Out-of-order execution can happen in two ways. The first is when we simply continue execution from a particular instruction: this is called a *jump*. The other is when we have a *subroutine*, which is a sequence of instructions with a beginning and end: the behaviour that we want is that we want to be able to *call* the subroutine using a particular instruction (the calling instruction), that the subroutine should then execute, and that, when it gets to the end, execution should resume from the instruction after the calling instruction (the address of this instruction is called the *return address*.

This piece of bureaucracy with addresses was recognised very early on. Turing discusses how to deal with return addresses for subroutines by putting the return address of the subroutine into memory:

> When we wish to start on a subsidiary operation we need only make a note of where we left off the major operation and then apply the first instruction of the subsidiary. When the subsidiary is over we look up the note and continue with the major operation. (*Turing 1946*, cited in *Priestley 2008*, Section 4.5, p. 104)

We should note that, although the precise details of 'mak[ing] a note of' have varied, this is still the same way that the return addresses of subroutines are handled.

There still remains the problem of jumps. Turing, too, worked on this problem: instructions were to be written in what Turing called a 'popular' format, and they were to be written, one instruction per card, in 'groups' (in practice, things like subroutines and the like: they should be sequences of instructions which could be guaranteed to end up in contiguous places in memory). Each instruction would have associated with it the name of the group and its 'detail figure', or place within the group. Programs would be constructed by taking all the relevant cards (for the main routine, subroutines and so on), collating them, and then assigning memory locations: from this one could translate from group name and detail figure to a memory location. Then one would have to replace all the memory addresses in instructions (which would, of course, have been written in the popular format with group names and detail figures) with the actual memory addresses. Turing seems to have envisaged this being done by hand (assisted by punched card manipulation machines, which were common technology at the time), but he did recognise that it would be something which could be done 'within the machine' (*Priestley* 2008, Section 4.5, p. 104).

The task of showing how such address translation could be done within the machine was begun by Goldstine and von Neumann (*1948*); they proposed first loading the program and its subroutines, and then running a 'preparatory routine' which would perform the required address modification (*Priestley* 2008, Section 4.5, p. 105). Williams, in 1948, worked out the details of this, and wrote, for the EDSAC, a sequence of instructions called the *initial orders*, which would perform all this address modification.

5.4. Accommodations

We have seen here the problems that were raised by introducing this set of ideas, and these practices, into computer programming: there is no direct support for these things in the Turing machine, and there was very rudimentary support, in the form of memory and instruction addresses, in the early hardware. There were already human practices—namely those developed for numerical analysis—which computers needed to support, and the solution was to develop coding languages and practices which were not too distant from what people were already doing by hand, and to fill the gap between human coding and machine execution by a process of translation. This process of translation—from human-written code to machine code—is one of the first appearances of our three-component system: hardware, human intellectual practices, and software mediating between them.

One of the important practices in this development was what is now called *code reuse*: the ability to write a single piece of code—a subroutine, or the like—and to be able to use it in a variety of contexts. From this comes the problem that we may not know, ahead of time, what the addresses of the code might be: so, consequently, we need to be able to manipulate code addresses in a systematic way. Turing has a very concrete description of the process, but the requirement is quite abstract: we know that our subroutine will be given some particular area of memory to work in, but we will not know ahead of time (and should not) precisely which area of memory it will be. This can be formulated mathematically, although the formulation is not trivial: *O'Hearn and Tennent 1997c*, p. 115 has a formulation in terms of possible world semantics (the possible worlds are given by collections of 'store shapes'). Despite the mathematical complexity of this concept, it does emerge, fairly naturally, from a requirement generated by the practice of programming, the requirement of code reusability.

Thus, though the Turing machine *was* invented as a model of computation, it was invented at a time when the only computation was human computation: there is nothing in the mere idea of a Turing machine which would by itself give rise to the elaborated human practice of programming which we describe in this article. Turing, in fact, was very interested in the practical problem of automating computation, and so he did a lot of work on this elaborated human practice, as we have seen in Section 5.1; but in another possible world he could have simply invented the Turing machine, written *Turing 1937*, and left it at that.

6. Modern hardware

In this section, we will look at how modern hardware typically works. One of the main problems here is that of speed disparity: the CPU of a modern computer executes a basic instruction in less than a nanosecond (complex instructions, like multiplication, might of course take longer), whereas hard disk access will take tens of milliseconds. RAM access is intermediate between the two. The ratio between CPU speed and hard disk speed is somewhat over a million. If the CPU had to wait milliseconds every time it needed data for an instruction, then computers would be much slower than they actually are. There are several technologies that we can use to deal with this. None of them is directly under the control of applications or systems programmers: they are designed by chip designers, they

(hopefully) make the hardware run faster, and they are *transparent*: that is, programmers simply write the instructions they would have written anyway, but the computer executes them faster than it would have done without caching, pipelining, and so on.

6.1. Caching

This is the strategy of fetching data in relatively large units, and storing it in fast memory near where it is to be used. For example, modern CPUs have caches where they store data that they get from RAM: they fetch data in large contiguous blocks, store it in their cache, and, when they need more data from RAM, they check the cache first (*Hennessy and Patterson 2014*, Section 5.3, pp. 383ff). This generally works: with modern hardware, CPUs will find the data in the cache about 90% of the time (*Hennessy and Patterson 2012*, *Stallings 2013*).

The reason why caching works, when it works, is what is called *data locality* (*Hennessy and Patterson 2014*, *Stallings 2013*). This is the idea that data which are relevant for a particular calculation is usually held contiguously: if we have large amounts of data, then it will generally be in an array or some large data structure of that sort, and that data structure will (hopefully) be held contiguously. So if we are iterating along an array, and if we cache the array in contiguous blocks, then most data will be found in the cache: the only reason to go to RAM for data is when we have already iterated through the data in the cache.

Similarly, instructions are stored in RAM in the order that they occur in the program, and they are *generally* executed in that order too. So caching tends to win here also.

However, data locality may fail in both of these scenarios. Two-dimensional arrays (i.e. arrays with two indices) can be thought of as big matrices, and there are two ways of iterating over them: in the first, you select column 0, iterate over that, then select column 1, iterate over that, and so on. For the other way of iterating, you do the same thing but with rows rather than columns. Now if you write array code in Java, it turns out that one of these ways gives you data locality but the other way does not,[16] so there is a large performance penalty for doing it one way rather than another.

Consequently, although there are large gains to be made by caching, they are not automatic: both for instructions and for data the gains depend on the statistical character of the code or data concerned.

6.2. Pipelining

Instruction execution in a CPU is generally performed in several stages: first the instruction is fetched from RAM, then the CPU decides what sort of instruction it is (which affects whether the instruction needs data, etc.), then the instruction is executed, and finally the result is written back to RAM. These stages are generally performed by different parts of the CPU, but they each depend on the previous stage having been performed. But there is nothing wrong with executing stage 1 for a particular instruction while the CPU is executing Stage 2 for the previous instruction: this is called *pipelining* (*Hennessy and Patterson 2014*, Section 4.5, pp. 272ff, *Stallings 2013*, pp. 517ff).

So, if we have a sequence of instructions like this

```
i1;
i2;
i3;
 ⋮
```

[16] If you have an array `a[i] [j]`, then you should iterate over `i` in the inner loop and over `j` in the outer loop, rather than vice versa.

then we could have a pipeline which, at successive times t_1, t_2, t_3, was doing the following:

t_1	stage 1 of `i1`		
t_2	stage 1 of `i2`	stage 2 of `i1`	
t_3	stage 1 of `i3`	stage 2 of `i2`	stage 3 of `i1`
		⋮	

What can go wrong? Suppose, on the other hand, we have a loop:

```
for(i=0;i<4;i++) {
        i1;
        i2;
        i3;
}
i4;
```

so that we will execute instructions in the following order:

```
i1 i2 i3 i1 i2 i3 i1 i2 i3 i1 i2 i3 i4.
```

The pipeline will fill up with, successively, `i1`, `i2`, `i3`, and then (because it is the next instruction in the listing) `i4`. However (unless this is the last time round the loop), the next instruction to be executed after `i3` is `i1`: but we cannot in general tell whether we are going round the loop again until the last instruction of this loop has been executed. And so, whenever we get to the end of the loop, we have to wait until the pipeline empties itself and then we have to start refilling the pipeline from `i1`

Because of this, modern CPUs generally do *branch prediction*: that is, at places where a branch might happen (end of a `for` loop, beginning of a `while` loop, and so on) they try to guess what branch might be taken, and keep filling the pipeline accordingly. For example, it is generally a good idea to guess that, when you get to the end of a `for` loop, you will continue execution from the beginning of it: this is because most loops get executed more than once, and often much more than once (there is no point in having them otherwise), so that you will generally win by guessing that way.

Branch prediction, then, is like data locality: it is a statistical matter, and depends on code being written in a certain way. It would certainly be possible to write code in such a way that it would break pipelining by forcing branch prediction to misbehave, but it would probably be quite hard to do, and it would need some specialised knowledge of what hardware it was to be run on (how many stages in the pipeline, some details of its branch prediction, and so on).

6.3. Registers

It has always been recognised that it was advantageous to have fast memory locations inside the CPU, and that these could be used for storing frequently used data. These locations are called *registers*, or *accumulators*, and have been part of computer design since the early days: e.g. the ENIAC—which started running in 1945 or so—had 20 accumulators (*Priestley 2008*, p. 61).

Registers can often achieve a considerable speedup. For example, if we have a loop like this:

```
for(i=0;i<100;i++) {
        i1;
        i2;
        i3;
        ...
}
```

then we could, theoretically, read the loop variable `i` from the hard disk every time it was to be used, and write it to hard disk every time it was to be updated, but that would be very wasteful: the sensible thing to do is to keep `i` in a register, initialise it to zero when we start executing the loop, and to release the register after the end of the loop: it need never even be stored in RAM. Assembly code allows access to registers: in fact, many assemblers (MIPS, for example) only allow arithmetic operations between the contents of registers, and only allow the result to end up in a register. There are other instructions for reading data from RAM to a register (*loading* the data) and for writing data from a register to RAM (*storing* the data). This is the so-called *load-store paradigm*.

Now because they involve RAM access, `load` and `store` instructions are comparatively slow, so there is a great deal to be gained by eliminating them as much as possible. For example, a `store` of data from a register to a location, followed by a `load` of the same data to the same register from the same location, can generally be eliminated; the same situation, only with different registers, can generally be replaced by simply moving data from one register to another, and so on. But a given CPU will only have a certain number of registers, so one will inevitably encounter situations where one has to store data in a particular register in order to make room for new data. This is the problem of *register allocation*: how to decide in what registers old data should be replaced by new data.

Now compilers, from languages such as C or Fortran, would generally emit assembly code for the relevant architecture, which would then be assembled and run. It used to be the case that human programmers could generally write better and more efficient assembler than compilers could produce (better register allocation, and so on): this is no longer the case, because of progress with register allocation algorithms. Thus, programmers' contact with modern computers generally does not reach inside the CPU, and, in particular, it generally does not deal with registers.

6.4. Hard disks

Hard disks are, as is well known, composed of a stack of rotating magnetic platters together with a set of read/write heads which are mounted on an arm that can move the heads in or out over the platters (*Hennessy and Patterson 2014*, *Stallings 2013*). The surfaces of each platter are divided into concentric *tracks*: that is, if the heads are at a fixed position, they will read or write from a particular track on each surface as the disks go round. The set of all of the tracks which are under the heads with the arm at a fixed position is called a *cylinder*. Each track is divided into *sectors*.

Thus, to access the data in a particular sector, the disk must do the following:

(1) move the heads to the appropriate cylinder,
(2) wait until the appropriate sector rotates under the head, and
(3) read or write the data from that sector.

The average time for (1) is between 3 and 13 ms, and the average time for (2) is about 5 ms: these are comparatively long times in computer terms. It is obviously advantageous

to take advantage of contiguity: that is, if we successively read data, it would be best if it came from the same cylinder.

However, this is very difficult to achieve. Modern hard disks have hardware in them called *disk controllers*, which do several things. One thing they do is to cache data as they read it from the disk or before they write it to the disk: because they do this, they can re-order reads and writes in order to minimise head movement. They also do error checking (they record extra check bits with the data in order to detect errors if they occur), they monitor errors, and they can move data off a sector and onto another if they think a sector is deteriorating. Because of this, it is very hard to tell whether data is located contiguously or not: blocks of data could start off close to each other, but end up distant because of data movement. None of this is under the control of programmers, and so optimisations that used to be possible are now no longer possible. On the other hand, disk are now genuinely faster and more reliable, due to more intelligent disk controllers, so this is probably a net gain.

6.5. Accommodations

We have seen that the programmer sees a particular view of the workings of a computer: the programmer can load and store data from the RAM to the CPU, can work on it in the CPU, and gets a generally sequential view of the execution of instructions. The programmer can also access hard disks, and these hard disks are organised in cylinders and tracks and sectors. None of this is strictly speaking true: data are cached between RAM and the CPU, and in the disk controller. Instructions are not strictly executed one at a time, but they overlap due to pipelining. And the hard disk geometry as seen by the programmer is an idealisation: data movement makes it very difficult to optimise the location of data on a hard drive.

So, again, we have a rather complex system. The programmer has a rather simplified view of the hardware: it is one where the speed disparities are not a problem, in which instructions are executed sequentially and one at a time, and in which hard disk access generally works unproblematically. That programmers can get away with this depends on a great deal of carefully optimised hardware: caches, pipelines, disk controllers, and the like. So programmers write programs, these programs are executed on a machine which does all sorts of clever things to maintain the rather fictitious view of itself that it presents to programmers, and they are, on the whole, executed quite fast. So this is what the machine does for its human users.

But, also, the machine depends in a certain way on its human users. Most of these optimisations are not guaranteed to work—that is, they are not guaranteed to run programs faster—but only work given programs and tasks with a particular statistical distribution. Optimisation is generally done to make the common case (i.e. what programmers mostly want done) fast. So human behaviour has an effect on this: the things that people want to do are usually what is optimised for (computer games are a case in point). Given all this, it is not surprising that computer benchmarks are a contentious, and ultimately political, subject (*Hennessy and Patterson 2014*, Section 1.9, pp. 46ff).

7. Semantics

Consider what we have seen of the development of programming languages, and particularly what we have seen of the idea of a variable. This was, we argue, introduced into the world of computers because it was implicit in the practice of numerical analysis: it was introduced in a rather ad hoc form, but gradually took on shape, first with the design of languages (such as Algol) which supported it perspicuously, and then with the development of semantics for those languages (and, following on from that, with the development of

program checking tools and other such infrastructure). These developments can be considered as the attempt to start with a partially theorised practice and then attaining reflective closure: developing a theory which rationalises and supports the practice. We should notice that reflective closure can be very difficult to achieve: for example, nobody seems to have realised that programs would take a good deal of debugging until they actually tried writing and running programs on real computers, even though, before this, many people had quite a good understanding of a good deal of the theory of computation (*Campbell-Kelly 1992*, p. 22): learning this took bitter experience. If, in addition, we want to formulate our insights mathematically (perhaps in order to do formal verification), then there might be a significant gap between having the conceptual insight and being able to develop the mathematics to express it: we see this, for example, with the formulation, which we have described above, of the requirements for code reusability and their eventual reformulation in terms of possible world semantics.

7.1. Compositionality

Since there have been programming languages, there has been work on their semantics: this is not surprising, firstly because many of the people who developed programming languages were logicians and thus naturally thought in terms of semantics, and secondly because computer scientists wanted to do things like prove that programs ran correctly, which is a question which could conceivably be answered by investigating the semantics of programs.

This proved to be a difficult problem. As Priestley remarks, 'semantics for logic are typically compositional' (*Priestley 2008*, p. 113): that is, logical formulae are generally built up from smaller components, and we can get the semantics of the larger pieces (i.e. the mathematical objects they stand for) by composing the semantics for smaller pieces. It would be natural to try to define a semantics for programming language in the same sort of way. However, it is not straightforward: consider, for example, one of the simplest ways of combining commands in programming languages, namely concatenating them:

```
command1
command2
```

which says that command 2 should be executed after command 1. What mathematical objects correspond to these commands? How do we combine them?

It was some time before good answers to these problems emerged: decisive breakthroughs were made by Strachey and his school in Oxford in the 1970s (*White 2004*, *Stoy 1977*). These results did not merely allow a mathematical analysis, but they led towards an understanding of the space of possible programming languages. In particular, they put into perspective the so-called *functional* languages: these are languages in which variables have fixed values, that is, their values can only be defined and read, but not updated. Variables in these languages, then, are much more like mathematical variables. Consequently, these languages have good mathematical properties, which made their semantics quite perspicuous (*White 2004*, Section 3.2). In particular, Abramsky (*1997*) has produced a semantically based taxonomy of a large collection of languages, based on functional programming but with added features; the semantics is based on the game-theoretic approach to logic pioneered in *Lorenzen 1984*.

7.2. Accommodations: the Kepler problem

We should notice that the benefits of this reflective closure – the development of an abstract formal semantics for programming languages – depend on having programming

languages which are amenable to such analysis. Here is an example of the use of such formal methods, together with the problems that might arise.

There is a problem, traditionally called the Kepler problem, which has to do with packing spheres in three-space: the problem is to discover what the arrangement is that leads to the greatest density (or, in practice, to show that the obvious arrangement leads to the greatest density). The problem was posed by Kepler in 1609: in 1998, Tom Hales solved the problem by producing a proof which reduced the problem to checking that all the members of a finite, but large, set of geometrical configurations possessed a certain property, and by writing software which checked the geometrical configurations. He submitted his work to the *Annals of Mathematics*, a journal which has a policy of checking, in detail, the proofs in submitted papers. They checked his proof, but then had to deal with his computer code, which was written in C++: the question was whether the code did what he claimed. The *Annals* appointed a committee of logicians to try to formally verify his code: by 2003, such verification had still not been achieved, and the *Annals* decided to publish the proof and have the computer code published elsewhere (*MacPherson 2005*, *Thomas 2004*).

Faced with this impasse, a group of mathematicians and computer scientists launched Project Flyspeck (*The Flyspeck Project 2014a*) with the aim of recomputing Hale's original calculations in such a way that they could be formally verified. They used, instead of C++, a functional language called Objective CAML, and they used a proof assistant called HOL Light which was very well integrated with CAML. They also checked the workings of HOL Light by starting with a very small and perspicuous core, which they then extended in stages, checking the correctness of the extensions by using the core.[17] To quote from the Flyspeck documentation,

> The system is built on a small kernel, and the type checking insures that no fatal bugs can occur outside the kernel. That is, if you leave a bug in your code, the worst that can happen is that you will fail to prove a theorem you hoped to prove. You can never put a bug in your code that yields a false statement (*The Flyspeck Project 2014b*).

The calculations were then recomputed, and the proof verified, in 2014 (*The Flyspeck Project 2014a*).

So why did Project Flyspeck succeed, when the previous attempt to verify Hales' calculation had failed? There are several differences, one of which is that the calculations were performed in a language which was designed, *inter alia*, for ease of verification: by contrast, C++ is a language which performs well but which seems to be hard to understand in any formal sense.

Why should C++ be different from CAML? One of the main differences is that C++ is imperative, whereas CAML is functional.

> In an imperative programming language, a term C is said to *interfere* with a term E if executing (or, as appropriate, assigning to or calling) C can affect the outcome of E. For example, command $x := a$ interferes with expression $x + 1$, but not vice versa. ...
>
> J.C. Reynolds has provided a more refined analysis. He argues that conventional procedural languages are problematical precisely because they permit *covert* interference, that is, interference that is not syntactically obvious. (*O'Hearn et al. 1997*)

[17] Note that this bootstrapping procedure could, together with the philosophical orientation called reliabilism, form the basis of a response to the DeMillo–Perlis attack on the very idea of formal verification described in *MacKenzie 2001*, Chapter 6.

So C++ suffers from interference (and, in practice, covert interference), and this makes it a hard language to verify because of such interference: in verifying the effect of a command, one has to consider not just its overt effect, but also any covert effects it might have on variables that interfere with the ones that it is acting on. CAML does not suffer from this, which makes it easier to verify.

What are the consequences of this for our model? We recall that this work on formal verification can be regarded as reflective closure, that is, as the rational analysis of a pre-existing practice. Now reflective closure works both ways: it can, of course, simply tell us something about our practice, but it can also prompt us to change our practice in such a way that it becomes more reliable or more perspicuous. And this is what seems to have happened here.

8. Epilogue: parallelism

Over the last several decades, there has been a remarkable trend: computers have become faster and more powerful, while becoming, if anything, cheaper (*Hennessy and Patterson 2014*, *Stallings 2013*). This now seems to be coming to an end: CPU clock speeds are not increasing any more. Rather, hardware manufacturers are turning to *multicore* hardware,[18] that is, hardware designs which have more than one processor on a chip (*Hennessy and Patterson 2014*, *Stallings 2013*). This change to multicore hardware means a change from serial to parallel processing, that is, processing in which many computations can go on at one time. This is a fundamental change, the emotional impact of which is perhaps best captured in James Mickens' essay 'The Slow Winter' (*Mickens 2013*).

Why might this be so fundamental? Many early computers, such as Turing's ACE, were capable of some sort of parallelism. This may indeed have seemed natural: Grier (*2011*) argues that the early culture of programming was strongly influenced by the planning of industrial operations in factories: similar ideas were certainly developed slightly later by Simon (*Heyck 2008a,b*) and by the cybernetics community (*Sjoblom 2011*). However, there seems to have been a general pressure towards strictly serial computers: programmers should be able to write a series of instructions and be assured that the machine would execute them in that order. Early computers quite often had some capacity for parallel programming but programmers—and competent programmers at that—generally found this very difficult to cope with: as Eckert writes of programming the ENIAC,

> In thinking out the various operations of this machine, if they can be thought out in a purely serial fashion, it is not necessary to worry about any irrelevant timing between the two steps. ... The human brain does not think in several parallel channels at the same time: it usually thinks these things out step by step. Therefore, in all ways, it is found exceedingly desirable to build the machine so that only single steps are performed at any time. The ENIAC is usually used in this way.[19]

Note 'usually': it was obviously possible to use the ENIAC in a non-serial way.

At that period, this issue was expressed in a discussion of the merits of 'centralised' versus 'decentralised' architectures: 'centralised' architectures are those 'in which the results of all calculation pass through a single accumulator' (Good, quoted in

[18] This whole historical development is related in some way to Moore's law (*Schaller 1997*), that is, the observation that the number of transistors per chip has been doubling every eighteen months since 1970 or so. The change to multicore hardware means that this increase can continue, although, since each chip now has several cores (i.e. processors), the number of transistors per core has levelled off. Similarly, the clock speed of CPUs has levelled off at about 2–4 GHz.

[19] J. Presper Eckert, in a lecture on 'A Preview of a Digital Computing Machine' (1946), cited in *Priestley 2008*, p. 94; see also *Haigh et al. 2014b*, pp. 22ff.

Copeland 2011, p. 23). By contrast, decentralised machines are those—such as Turing's ACE – in which calculations could be performed along a variety of hardware paths.[20]

For one reason or another, centralised designs won out, and programming has generally remained quite serial ever since; there are niches in which parallel programming is used (mostly out of necessity), but, for example, the average undergraduate computer science curriculum has very little parallel programming in it.

8.1. Accommodations

It would be easy to read this story as simply driven by technological necessity, but there is also a more subtle backstory. It is still certainly true that parallel programming is difficult: Brooks (*1997*, p. 332) writes 'It is well known that parallel programs can be hard to reason about, because of the potential for undesirable interference between commands running in parallel'. Nevertheless, a good deal of theoretical work has been done on parallel computing, some of it quite successful. Functional languages seem to be well adapted for parallel computation, precisely because they are not susceptible to interference. So an obvious solution might be to change our practice and start using functional programming: and, to an extent, this is happening. Universities are starting to teach functional programming, or are resuming the teaching of it (my university, for example, has recently started teaching functional programming again after about a decade in which it was not taught).

However, functional languages are not well adapted to all problems, which leads one to ask whether we can change our practice in a more nuanced way than adapting functional languages across the board. In fact we can: for example, there is a framework called Hadoop,[21] which is used for the parallel processing of big data. Hadoop is written in the imperative language Java, but is based on a functional programming paradigm called map reduce (*Lin and Dyer 2010*). And it turns out that, provided that people program with Hadoop in a fairly disciplined way—that is, if they do not use imperative constructs in sensitive contexts—they can get the advantages of functional programming without having to learn to cope with purely functional languages.

So this constitutes a successful accommodation: it is not the puritan functional programming paradise, in which everyone would program in beautiful functional programming languages. It is also a rather niche application, and it does not tell us much about how to do parallel programming in other contexts. But it is an example of the sort of accommodations that people make in order to deal with the technology that they have available, and the sort of human systems that they have available, based on the insights that the available theory can give them. And, as well as the purely technological issues, some of the issues at play here are how the humans make sense of the technology that they have, and how they adapt the technology so that they can make sense of it. This, of course, raises further problems, but arguably this evolution (from imperative programming to functional programming to the use of functional idioms in imperative languages) leaves the technological—social system slightly further along than it was previously.

So the expository arc of this paper has shown how particular programming phenomena—those having to do with the idea of a variable, and with memory addresses—started with a practice rooted in pre-electronic computation, which then got imported into

[20] Turing's design was not only decentralised, but also intensional: instructions did not access memory directly, but only via a 'system of interconnections and gates', which Turing called a tree. *Brooks 2012* describes the ACE as 'a beast to program'. Kilburn, responsible for much of the design of the Manchester Baby, was absolutely certain that 'my computer was not going to look like that', and he used, rather, Newman's simpler idea of 'numbers being identified by the address of the house in which they were placed'.

[21] See http://hadoop.apache.org/.

computer science and gradually rationalised (first with suitable languages, then with mathematical semantics). The rationalisation seemed to suggest a change to the practice (namely the adoption of functional languages instead of imperative languages): this change proved very unpopular on the brute human level. And, finally, there are suggestions that appropriate compromises may be emerging, allowing programmers to use functional idioms in languages that they feel comfortable with.

This raises a methodological problem. We started with the observation that the communities who program and design computers are constantly engaged in a process of adjusting matters so as to constitute a coherent view of the world; this naturally led into a rather ethnomethodological view of the subject. We have seen, however, that this situation is not static: these are practices which are evolving, in many ways as a result of conscious and reasoned decisions taken in order to remedy perceived defects of the then current state of affairs. These changes are changes not solely in actual practices and artefacts but also in norms: for example, the evolution of programming languages was partly a matter of becoming clear about what makes a *good* programming language, and what we should aim for when we design them. Some of these norms, as we have seen, may end up inscribed in hardware. So, our methodology maybe ought to reflect this evolutionary aspect of the history. Something like the critical theory of *Honneth 2014*, with a systematic account of the connection between social institutions and the implementation of norms, may well be appropriate: the historical data that we have discussed should be a rich source for the development of such a critical theory of computer technology and, more generally, of cognitive science as a whole.

Disclosure statement

No potential conflict of interest was reported by the authors.

References

Abramsky, S. 1997. 'Games in the semantics of programming languages', in M. S. P. Dekker and Y. Venema, *Proceedings of the 11th Amsterdam Colloquium*, Amsterdam, 1–6.

Aiken, H. H., and Hopper, G. M. 1946. 'The automatic sequence controlled calculator', *Electrical Engineering*, **65**, 384–91. 449–54, 522–28.

Backus, J. 1981. 'The history of Fortran I, II and III', in *Wexelblat 1981*, pp. 25–45. From the *ACM SIGPLAN History of Programming Languages Conference*, New York, NY, June 1–3, 1978.

Bergin, T. J. T. 2014. 'What is a computer? An answer from the past', *Annals of the History of Computing*, **36** (3), 80–4.

Brooks, F. P. 1995. *The Mythical Man-Month*, Boston: Addison-Wesley.

Brooks, S. 1997. 'The essence of parallel Algol', in *O'Hearn and Tennent 1997b*, pp. 331–48.

Brooks, F. P. 2012. Turing's Pilot ACE: Why not important? *Alan Turing Centenary Conference*, Manchester. http://videolectures.net/turing100_brooks_pilot_ace/.

Campbell-Kelly, M. 1992. 'The Airy tape: An early chapter in the history of debugging', *Annals of the History of Computing*, **14** (4), 16–26.

Ceruzzi, P. E. 2000. *A History of Modern Computing*, Cambridge, MA: MIT Press.

Chilimbi, T. M. 2001. 'Efficient representations and abstractions for quantifying and exploiting data reference locality', in M. Burke and M. L. Soffa, *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, June 20–22, ACM, 191–202. http://doi.acm.org/10.1145/378795.378840.

Copeland, B. 2011. 'The Manchester computer: A revised history part 2: The Baby computer', *Annals of the History of Computing*, **33** (1), 22–37.

Daylight, E. G. n.d., Towards a historical notion of 'Turing – the father of computer science'. To appear in *History and Philosophy of Logic*; http://www.dijkstrascry.com/TuringPaper.

De Mol, L., and Bullynck, M. 2012. *A short history of small machines. Given at Computability in Europe 2012.* http://logica.ugent.be/centrum/preprints/CIE12.pdf.

Ensmenger, N. 2009. 'Software as history embodied', *Annals of the History of Computing*, **31** (1), 88–91.

Farquhar, E., and Bruce, P. 1994. *The MIPS Programmer's Handbook*, Los Altos, CA: Morgan Kaufmann Publishers.

Galloway, P. 2011. 'Personal computers, microhistory, and shared authority: Documenting the inventor-early adopter dialectic', *Annals of the History of Computing*, **33** (2), 60–74.

Garfinkel, H. 1967. *Studies in Ethnomethodology*, Englewood Cliffs, NJ: Prentice-Hall.
Gobbo, F., and Benini, M. 2013. 'From ancient to modern computing: A history of information hiding', *Annals of the History of Computing*, **35** (3), 33–9.
Goldstine, H. H., and von Neumann, J. 1948. *Planning and coding problems for an electronic computing instrument, part ii, volume 3*, Technical report, Institute for Advanced Study, Princeton.
Grier, D. A. 2011. 'Programming and planning', *Annals of the History of Computing*, **33** (1), 86–8.
Haigh, T. 2015. 'Actually, Turing did not invent the computer', *Communications of the ACM*, **57** (1): 36–41.
Haigh, T., Priestley, M., and Rope, C. 2014a. 'Engineering "the miracle of the ENIAC": Implementing the modern code paradigm', *Annals of the History of Computing*, **36**, 41–59.
Haigh, T., Priestley, M., and Rope, C. 2014b. 'Reconsidering the stored program concept', *Annals of the History of Computing*, **36** (1), 4–17.
Hartree, D. R. 1952. *Numerical Analysis*, Oxford: Clarendon.
Hennessy, J. L., and Patterson, D. A. 2012. *Computer Architecture: A Quantitative Approach* (5th edn). Los Altos, CA: Morgan Kaufmann.
Hennessy, J. L., and Patterson, D. A. 2014. *Computer Organisation and Design: The Hardware-Software Interface* (5th edn). Los Altos, CA: Morgan Kaufmann.
Heyck, H. 2008a. 'Defining the computing: Herbert Simon and the bureaucratic mind – part 1', *Annals of the History of Computing*, **30** (2), 42–51.
Heyck, H. 2008b. 'Defining the computing: Herbert Simon and the bureaucratic mind – part 2', *Annals of the History of Computing*, **30** (2), 52–63.
Honneth, A. 2014. *Freedom's Right*, Cambridge, MA: Polity.
IEEE Computer Society. 2008, *IEEE standard for floating-point arithmetic*. IEEE Std 754-2008. doi:10.1109/IEEESTD.2008.4610935.
Java Numerics. 2013. Java Grande Forum Numerics Working Group. http://math.nist.gov/javanumerics/.
Landin, P. J. 2000. 'My years with Strachey', *Higher Order and Symbolic Computation*, **13**, 75–6.
Lin, J., and Dyer, C. 2010. *Data-Intensive Text Processing with MapReduce*, San Rafael, CA: Morgan and Claypool.
Lorenzen, P. 1984. *Normative Logic and Ethics*, Mannheim: Bibliographisches Institut.
MacKenzie, D. 2001. *Mechanizing Proof: Computing, Risk and Trust*, Cambridge, MA: MIT.
MacPherson, R. D. 2005. 'Machine computation and proof', in A. Bundy, M. Atiyah, A. Macintyre, and D. MacKenzie, *The Nature of Mathematical Proof* , number 1835 of *Philosophical Transactions of the Royal Society*, Vol. 363, London: The Royal Society, p. 2461.
McCarthy, J. 1981. 'History of Lisp', in *Wexelblat 1981*, pp. 173–85. From the *ACM SIGPLAN History of Programming Languages Conference*, New York, NY, June 1–3, 1978.
Medwick, P. A., and Mahoney, M. S. 1988. 'Douglas Hartree and early computations in quantum mechanics: The history of computing in the history of technology', *Annals of the History of Computing*, **10** (2), 105–11.
Mickens, J. 2013. The slow winter *;login: logout*, pp. 14–7.
Naur, P. 1981. 'The European side of the last phase of the development of Algol', in *Wexelblat 1981*, pp. 92–139. From the *ACM SIGPLAN History of Programming Languages Conference*, New York, NY, June 1–3, 1978.
Nofre, D., Priestley, M., and Alberts, G. 2014. 'When technology became language: The origins of the linguistic conception of computer programming, 1950–1960', *Technology and Culture*, **55** (1), 40–75.
O'Hearn, P. W., Power, A. J., Takeyama, M., and Tennent, R. D. 1997. 'Syntactic control of interference revisited', in *O'Hearn and Tennent 1997b*, pp. 189–225.
O'Hearn, P., and Tennent, R. (eds). 1997a. *Algol-Like Languages, Vol. 1*, Boston, MA: Birkhäuser.
O'Hearn, P., and Tennent, R. (eds). 1997b. *Algol-Like Languages, Vol. 2*, Boston, MA: Birkhäuser.
O'Hearn, P. W., and Tennent, R. D. 1997c. 'Parametricity and local variables', in *O'Hearn and Tennent 1997b*, pp. 109–163.
Perlis, A. J. 1981. 'The American side of the development of Algol', in *Wexelblat 1981*, pp. 75–91. From the *ACM SIGPLAN History of Programming Languages Conference*, June 1–3, 1978.
Priestley, P. M. 2008. *Logic and the Development of Programming Languages, 1930–1075*, PhD thesis, University College London.
Russell, A. 2012. 'Standards, networks and critique', *Annals of the History of Computing*, **34** (3), 80.
Schaller, R. 1997. 'Moore's law: Past, present and future', *IEEE Spectrum*, **34** (6), 52–9.
Sjoblom, G. 2011. 'Control in the history of computing: Making an ambiguous concept useful', *Annals of the History of Computing*, **33** (3), 88–97.
Southwell, R. 1940. *Relaxation Methods in Engineering Science: A Treatise on Approximate Computation*, Oxford: OUP.
Stallings, W. 2013. *Computer Organisation and Architecture* (9th edn). Cambridge, MA: Pearson.
Stoy, J. E. 1977. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, Cambridge, MA: MIT Press.
The Flyspeck Project: 2014a. Accessed 15 September 2014. https://code.google.com/p/flyspeck/.
The Flyspeck Project: 2014b. Accessed 15 September 2014. https://code.google.com/p/flyspeck/wiki/FlyspeckFactSheet.
Thomas, R. 2004. *Welcome to the maths lab*. http://plus.maths.org/content/welcome-maths-lab.

Tinn, H. 2011. 'From DIY computers to illegal copies: The controversy over tinkering with microcomputers in Taiwan, 1980–1984', *Annals of the History of Computing*, **33** (2), 75–88.
Turing, A. 1937. 'On computable numbers, with an application to the Entscheidungsproblem', *Proceedings of the London Mathematical Society*, **42**, 230–65.
Turing, A. M. 1946. *Proposal for development in the mathematics department of an automatic computing engine (ACE)*, Technical report, National Physical Laboratory, Teddington.
Turing, A. 1989. 'Checking a large routine', in M. Campbell-Kelly, *The Early British Computer Conferences*, Cambridge, MA: MIT Press. pp. 70–2. http://dl.acm.org/citation.cfm?id = 94938.94952.
Watt, D. A. 1990. *Programming Language Concepts and Paradigms*, Upper Saddle River, NJ: Prentice Hall.
Wexelblat, R. L. (ed). 1981. *History of Programming Languages*, New York: Academic Press. From the *ACM SIGPLAN History of Programming Languages Conference*, June 1–3, 1978.
Wheeler, J. M. 1992. 'Applications of the EDSAC', *Annals of the History of Computing*, **14** (4), 27–33.
White, G. G. 2004. 'The philosophy of programming languages, in L. Floridi, *The Blackwell Guide to the Philosophy of Computing and Information*, Oxford: Blackwell, pp. 237–47.