Programs = data = first-class citizens in a computational world

BY NEIL D. JONES AND JAKOB GRUE SIMONSEN*

Department of Computer Science, University of Copenhagen (DIKU) Njalsgade 126, 2300 Copenhagen S, Denmark

From a programming perspective, Alan Turing’s epochal 1936 paper on computable functions introduced several new concepts, including what is today known as self-interpreters and programs as data, and invented a great many now-common programming techniques. We begin by reviewing Turing’s contribution from a programming perspective; and then systematize and mention some of the many ways that later developments in models of computation (MOCs) have interacted with computability theory and programming language research. Next, we describe the ‘blob’ MOC: a recent stored-program computational model without pointers. In the blob model, programs are truly first-class citizens, capable of being automatically compiled, or interpreted, or executed directly. Further, the blob model appears closer to being physically realizable than earlier computation models. In part, this is due to strong finiteness owing to early binding in the program; and a strong adjacency property: the active instruction is always adjacent to the piece of data on which it operates. The model is Turing complete in a strong sense: a universal interpretation algorithm exists that is able to run any program in a natural way and without arcane data encodings. Next, some of the best known among the numerous existing MOCs are described, and we develop a list of traits an ‘ideal’ MOC should possess from our perspective. We make no attempt to consider all models put forth since Turing’s 1936 paper, and the selection of models covered concerns only models with discrete, atomic computation steps. The next step is to classify the selected models by qualitative rather than quantitative features. Finally, we describe how the blob model differs from an ‘ideal’ MOC, and identify some natural next steps to achieve such a model.

Keywords: programming; recursion theory; models of computation

1. Turing’s paper from a programming perspective

First and foremost, Turing [1] made a convincing analysis of the concept of ‘effective process’ or ‘computable function’. This led, by small and logically well-founded steps, to a computation model now widely known as the Turing machine.

Author for correspondence (simonsen@diku.dk).

One contribution of 18 to a Theme Issue ‘The foundations of computation, physics and mentality: the Turing legacy’.
Turing’s machine had a concept of control state, called a ‘configuration’. It also had a concept of data state. Turing’s data state was an infinite tape, almost everywhere blank, containing symbols from a finite alphabet. Turing’s program was a finite set of quintuples \((p, a, b, m, q)\). Each quintuple defines a single computation step: if the machine is in control state \(p\) and is scanning tape symbol \(a\), the next step is to replace \(a\) by \(b\), move the scanning head as directed by \(m\) (right one position, left one, or not at all) and change control state to \(q\).

The control state is a priori bounded for any one program; but the data state is unbounded. The paper showed several non-trivial examples of programming, including a by-hand version of the technique later known as macro expansion.

It also introduced the concept of program as data: an encoding of a program as a sequence of tape symbols. This concept made it possible to state and prove a significant result: the undecidability of the halting problem. The concept of ‘program as data’ led further, directly to an explicit construction of a universal machine: a single Turing machine able to simulate any other Turing machine (or even itself). In programming languages, this is now known as a ‘self-interpreter’.

There were some loose ends in the original paper: a few programming bugs were found in the universal machine (perhaps, the first programming bugs ever discovered and publicly corrected). A few features were only touched lightly upon: the size of the tape alphabet, and the division among input values, work space and output values. Furthermore, although programs can be data, these data are just encodings, and very far from being directly executable.

The paper established the Turing machine both as a means of describing the act of computation (abstractly, but with an eye to physical computation by a human computer) and as a vehicle for general problem solving by programming.

2. Programming and models of computation

Structure of this paper. Our goal is to obtain an ‘ideal’ model of computation that allows for general problem solving, is physically realizable, and allows the same theoretical developments as the Turing machine without the need for awkward encodings. This goal has been partially achieved by later developments, both by trying to knot the loose ends above and by refining the fundamental definitions of machine model. Nonetheless, it appears that, 75 years later, the ideal model is still elusive. In §2, we review some developments occurring after Turning [1] to see how they relate to this goal. In §3, we briefly overview our recently developed ‘blob’ model. In §§4 and 5 (also appendix A), we study more systematically how models of computation (MOCs) may be compared and evaluated.

Two meanings of the word ‘model’. This word is widely used, but with two very different meanings. A first analytical meaning comes from the natural sciences. From this view, a ‘model’ is a description of an already-existing reality. The model is good or bad inasmuch as it gives a faithful description of reality, e.g. so it can be used to predict the outcomes of not-yet-performed experiments.

A second synthetic meaning comes from fields such as computer science or engineering, in particular, in ‘model checking’. These fields centre on constructing an artefact (e.g. a computer program or a hardware device) that is faithful to a problem specification. The constructed artefact is a good or a bad model insofar as it satisfies the specification. If so, the program or hardware device may be used in the secure knowledge that it will behave as required.
Programs = data = first-class citizens

Turing’s [1] paper was widely interpreted from the first, analytical viewpoint. It seemed that he had captured a significant part of a natural phenomenon: a ‘computably solvable problem’. This was strengthened by the fact that several other researchers, with widely different and independent starting points, had all converged on equivalent formulations of computability. This confluence of ideas has been reported vividly by Gandy [2].

On the other hand, program construction is itself a synthetic activity: one has in mind in advance what a program should accomplish (e.g. it should satisfy a specification; for example, it should compute a mathematically defined function). This viewpoint may be seen in the world’s first programming manual [3].

Impact of Turing’s paper. We now list some key insights and later activities, in roughly chronological order. (This list is very far from being exhaustive.) Beyond being an historical overview, the following topics encompass the very essence of computer science as known today, more than 75 years later.

The development in the 1940s and 1950s both in the field of computability theory, hand in hand with the description of computation by binding names to values, was already in its early stages at the time Turing did his work. Artefacts of this work, such as self-reproducing programs, were studied at length, both theoretically (as a consequence of Kleene’s second recursion theorem) and operationally (via cellular automata). Study of resource-constrained computation in the 1960s led to the definition and development of time- and space-bounded computation, e.g. the complexity class \( \text{PTIME} \) [4,5]. The advent of physical computing devices led to the development of a great many machine architectures and hardware implementations of many of these machine architectures. A great many programming languages and their compilers were developed from the 1950s (a development still occurring today). Since the 1980s, there has been a surge of interest in automatic program transformations such as partial evaluation [6].

Work on programming led to a refinement of algorithmic techniques, such as subroutines, recursive programs and many others. Researchers developed the second aspect of the term ‘model’ further, leading to program synthesis from logical specifications, program verification and later model checking.

From this cornucopia of results, we pick two broad aspects where Turing [1] was pioneering: The development of programming as a general problem-solving technique, requiring programming languages, and the development of MOCs as vehicles for theoretical reasoning and for physically implementable constructions.

(a) Programming languages

Turing’s major step forward by devising the universal machine was to show that the algorithm (for solving a problem) is primary, and that the hardware on which it was executed was secondary. This major advance shifted attention from what one can build (as a computing device) to how one can program a single device, as long as it is sufficiently efficient to simulate (or otherwise realize) the universal Turing machine.

This viewpoint is central to our work: we regard the possibility of programming, to satisfy a given problem specification, as being just as important to computability as is the hardware architecture of a computation model. Without programmability, a computing model is difficult to use in practice, and far from a ‘universal machine’ as envisioned in 1936. Unfortunately, many later
MOCs have downplayed the programmability aspect, resulting in machines with considerable computational power, but that are very hard to use for general-purpose computing.

In 1936, programming languages were not yet thought of: programs were just ‘codes’, to be provided as fodder for (some version of) the universal machine. Nonetheless, programming languages were soon on the march. Early steps can be seen in Church’s λ calculus [7]; later, McCarthy devised the programming language LISP closely based on it [8]. Furthermore, even the recursion schemes of Kleene, Gödel and others could be seen as programming languages. More practical work proceeded bottom-up, starting with low-level languages such as Autocode and Fortran [9].

More sophisticated languages, beginning with Algol V. 60 [10], embodied concepts for algorithm development and organization principles, e.g. bottom-up, top-down and recursive program development, modularity, types, higher order functions, automatic memory management and much more. Several generations of programming languages developed ancillary concepts used in diverse application settings, for instance, implicit memory management and concurrent computations via threads, channels, etc.

(b) Computability theory

Turing’s paper [1] on Hilbert’s Entscheidungsproblem contained a sketch proof that the set of partial functions \( \mathbb{N} \rightarrow \mathbb{N} \) computable by the class of Turing machines was exactly the same as the class of functions computable by the lambda calculus that Church (independently and a bit previously) used to show the unsolvability of the Entscheidungsproblem [11]. Turing [12] provided a full proof of this equivalence as well as the equivalence of the Turing machine with Kleene’s recursive function theory [13].

These equivalence results are remarkable not only in their conclusions, but also their proof methods. All proofs proceed by simulating one model of computation in another, and carefully avoid non-constructive methods such as the principle of the excluded middle. In modern terminology, the proofs are roadmaps to construct compilers between programs in the models.

Given the equivalences between such models, subsequent results proved in recursive function theory turned out to hold, mutatis mutandis, in every reasonable model \( M \) of computation Gandy [2]. For reference works on computability theory, see Rogers [14] and Jones [15].

We very briefly state some fundamental assumptions and theorems of computability theory. We write \( \mathbb{D} \) for data, and assume that programs are data values (in classical computability theory, \( \mathbb{D} \) is the set of natural numbers). Let \( p \) be a program in a given programming language \( M \) (e.g. a set of quintuples defining a Turing machine). Then, \( JpK(d) \) represents the result of \( p \)'s computation (possibly undefined, if \( p \) fails to terminate).

**Existence of Turing’s universal machine**

\[
\exists U \in \text{programs} \ \forall p \in \text{programs} \ \forall d \in \mathbb{D} : JpK(d) = JUK(p, d).
\]

**The s-m-n theorem, for the case \( m = n = 1 \)**

\[
\exists s_1^* \in \text{programs} \ \forall p \in \text{programs} \ \forall d_1, d_2 \in \mathbb{D} : JpK(d_1, d_2) = JJs_1^*K(p, d_1)K(d_2).
\]
Kleene’s second recursion theorem

\[ \forall p \in \text{programs } \exists e \in \text{programs } \forall d \in \mathbb{D} : \text{JpK}(e, d) = \text{JeK}(d). \]

Programming language researchers have computer implemented all these ideas: specifically, the universal machine is a self-interpreter [6], the s-m-n theorem is the theoretical basis for partial evaluation (cf. [6]) and computer implementations of the recursion theorem have been performed [16,17].

These main results from recursive function theory are present in all Turing-complete MOCs [14], but occasionally require some re-encoding.

(c) A bestiary: some models of computation

Since Turing’s pathbreaking work (and that of his colleagues from the mid-1930s), a great number of diverse MOCs have appeared—so many and so diverse that it is difficult to gain an overall view of their features, motivations or contributions. The list below contains some well-known MOCs up to 1975 that are still in (academic) use, as well as a few later models. Space constraints prevent us from describing each model, and from giving an exhaustive list of the multitude of extant models. We list the model names only, along with the earliest reasonable reference.

Finite automata (FA). First systematic treatment by Rabin & Scott [18].

von Neumann architecture. Draft circulated by von Neumann [19].

Lambda calculus (\(\lambda\)). Untyped \(\lambda\) calculus by Church [11].

Counter machine (CM). Informal in 1950s; Lambeck [20] and Minsky [21].

Text register machine (1\#). Moss [16]. A more programmable basis for computability theory.

Random access machine (RAM). Register machine with indirect addressing by Wang [22]. Final definition by Cook & Reckhow [23].

Random access stored program (RASP). Elgot & Robinson [24].

Cellular automaton (CA). von Neumann et al. [25]; von Neumann [26].

LIFE. Conway, published in popular science article by Gardner [27].

Abstract state machine (ASM). Gurevich [28], Börger [29] and Dexter et al. [30].

Reaction–diffusion machine (RDM). Adamatzky et al. [31]; Bandini et al. [32]; Adamatzky [33]. Turing’s [34] paper and the chemical abstract machine [35].

Blob. Hartmann et al. [36,37]. Yet another model, described next.

3. A biologically motivated model of computation

The blob model\(^1\) introduced by Hartmann et al. [36,38] is a biologically inspired model of computation specifically tailored to afford Turing completeness, programmability in a physically plausible setting that satisfies strong finiteness (explained below).

The leitmotif is to keep the program control point and the current data inspection site always close to a focus point where all actions occur (see figure 1). This can be done by continually shifting the program or the data, to keep the control cursor (PC) and the data cursor (DC) always adjacent to the focus.

\(^1\)Our blob model [36,38] is not to be confused with another MOC of the same name, independently developed by Gruau et al. [39,40].
Before reading on, we urge the reader to consult the movie http://dk.diku.blob.blobvis.s3.amazonaws.com/largedata-play.avi to see the actual execution of a blob program. (PC is the dark green blob and DC is the dark red blob.)

A run-time state consists of an assembly of ‘blobs’. These may be thought of as abstract versions of molecules or cells, each bound to at most four adjacent neighbours\(^2\) and floating in a not-further-specified ‘biological soup’.

A bond connects exactly two blobs: it is a two-way link between two bond sites. There is no fan-in or fan-out: any bond site is unbound or bound to exactly one blob.

A blob program \(p\) is (by definition) a connected assembly of blobs. A data value \(d\) is (also) by definition, a connected assembly of blobs. At any moment during execution, i.e. during computation of \(J_p K(d)\), we have

- one blob in \(p\) is active, known as the PC;
- one blob in \(d\) is active, known as the DC; and
- a bond *, between the PC and the DC, is linked at a designated bond site (bond site 0 of each).

A blob has several bond sites and a few bits of local storage limited to fixed, finite domains. Specifically, our model will have four bond sites, identified by numbers 0, 1, 2, 3. At any instant during execution, each bond site can hold a bond—that is, a link to a (different) blob; or a bond site can hold \(\perp\), indicating unbound.

In addition, each blob has eight cargo bits of local storage containing Boolean values, and also identified by numerical positions: 0, 1, 2, …, 7. When used as program, the cargo bits contain an instruction (described below) plus an activation bit, set to 1. When used as data, the activation bit must be 0, but the remaining seven bits may be used as the user wishes.

For program execution, one of the eight cargo bits is an ‘activation bit’; if 1, it marks the instruction currently being executed. The remaining seven cargo bits are interpreted as a 7 bit instruction so there are \(2^7 = 128\) possible instructions in all.

Instructions and semantics. The blob instructions correspond roughly to ‘four-address code’ for a von Neumann-style computer. The format is

\[
\text{operation code parameters (bond0, bond1, bond2, bond3).}
\]

\(^2\)While this may seem at first sight to resemble the global state of a CA, there are essential differences: (i) the blobs are not arranged in a two-dimensional grid, (ii) there is no universal clock to cause all cells to perform their state transitions synchronously, and (iii) blobs are uniformly programmable in the sense that there is a single set of transition rules for all programs.
The operation code and parameters together occupy seven cargo bits; the rest of the instruction is the four bonds. An essential difference from von Neumann-style computers is that a bond is a two-way link between two blobs and is not an address at all. It is not a pointer; there exists no address space as in a conventional computer. The four bond sites contain links to other instructions, or to data via the PC–DC bond.

What happens at the program-to-data bond? An instruction can do the following.

— **Move.** Move the data cursor along bond 1 (or bond 2 or 3).
— **Set.** Set one of the data cursor’s cargo bits \( i \) to 1 (or 0) \( (i = 1, 2, \ldots, 7) \).
— **Branch.** Test whether the data cursor’s cargo bit \( i = 1 \) or 0? \( (i = 1, 2, \ldots, 7) \).
— **Branch.** Test whether the data cursor’s bond 1 is empty or not? (or 2 or 3).
— **Insert.** Add a new blob at bond 1 (or 2 or 3).
— **Swap.** Interchange some bonds at distance 0, 1 or 2 from the DC.
— **Fan-in.** Merge control from two predecessor instructions.

**Why exactly four bond sites and eight cargo bits?** One bond site is reserved for the PC–DC bond. Two more bond sites are needed to create an instruction list (its predecessor bond to the preceding instruction, and a successor bond); and a conditional instruction has both a true predecessor and a false predecessor. At least three bond sites per blob seem to be needed for Turing completeness. The choice of eight cargo bits is only a technical convenience, so that one instruction can be stored in a single blob.

**Strong finiteness.** There is one, fixed set of blob instructions (not counting the bonds). A program is a set of blobs, and program evaluation happens according to this, fixed, set of instructions. Different programs do not have different instruction sets (and this is not necessary for Turing completeness, as shown in Hartmann *et al.* [36] in the appendix). The bond connections, however, may be different in different programs.

The formal semantics of instruction execution are specified precisely by means of a set of 128 biochemical reaction rules in the style of Danos & Laneve [41].

The blob formalism has some similarity to *LISP* or *SCHEME*; but, there are no variables, there is no recursion and bonds have a ‘fan-in’ restriction.

**What can be done in the blob world?** A self-interpreter has been constructed, and the usual programming tasks (appending two lists, copying, etc.) can be solved straightforwardly, albeit not very elegantly because of the low level of blob code. Hartmann *et al.* [36] show how to generate blob code from a Turing machine, thus establishing Turing completeness.

**Programs as data.** One crucial difference between the blob world and most other paradigms is that programs are literally data. For the universal Turing machine, Alan Turing devised a standard representation of programs as data to be placed on the input tape. But for our model, there is no difference between a blob belonging to the ‘data’ part and a blob belonging to the ‘program’ part of an ensemble of blobs. Indeed, such a distinction would be artificial.
4. A wish list for an ideal model of computation and playing Linnaeus

We see (from perhaps a biased viewpoint) the following criteria as necessary and desirable properties for an ‘ideal’ MOC. The most important are listed first.

Existence of programs. There exist programs in the MOC. This cannot be taken for granted. Even though constructing a state transition function for a finite automaton or von Neumann-style CA may be thought of as programming, there do exist contexts where it is hard to see any clearly identified program. Examples include LIFE (with one a priori fixed transition function for all programs) and various biological computation models, including the reaction–diffusion model [31,34].

General problem solving. There should be a natural development path from an informal algorithm to an MOC program, and this path should not be excessively long or difficult. It is also desirable that the MOC be Turing complete: it is computable, and any Turing-solvable problem should be MOC solvable.

Physical realisability. Program execution should be possible without action at a distance. Examples of action at a distance in extant MOCs include data pointers (or addresses or indirect addressing), as found in the von Neumann architecture, or the RAM or RASP, and quantum entanglement (an enchanting idea, but still only a dream on the level of programming). Turing’s model did not involve action at a distance.

Uniformity. It is enough to have one set of hardware, so one does not need to construct new hardware to solve new problems.

Applying this thought to programs, it implies strong finiteness: there should only be an a priori fixed number of possible instructions. This implies there can be no unbounded program building blocks of any sort, e.g. instruction labels, register numbers or variable names, or even constants appearing in instructions.

Programs as data objects. Readability of programs is a prerequisite for existence of a universal machine. A next step is that programs be writeable, e.g. as in the s-m-n theorem, or in a compiler, or in a partial evaluator.

A step still further is that program activation is possible: once a program is generated, it may be set into action. Only ‘stored program’ models have this property. There are a few examples in the literature, e.g. the von Neumann architecture [19] and the RASP model [24]. While not now explicitly present in the blob model of Hartmann et al. [36], this ability should be very easy to add, since a program is a collection of blobs.

Concurrency. Many programs may be active, i.e. executing, at the same time. (These could have been given initially, or generated dynamically by one master program, analogous to biological reproduction.)

Other desirables. Programs should have plausible running times, e.g. not significantly worse than or uncomputably better than the runtime classes known from complexity theory, e.g. PTIME, NPTIME, PSPACE (or their parallel counterparts) should be stable classes.

(a) Playing Linneaus

For two of the properties on the wish list, it is quite clear how to identify whether a given MOC has them: uniformity and programs as data objects. For others, for example, physical realizability, there may be several ‘grades of truth’.
Is a CA physically realizable, even if we have to keep a copy of its transition function in each cell? Is it less realizable than a finite automaton? Is it more realizable than a RAM? To brazenly claim that certain models completely satisfy one of our wish-list properties is likely too contentious. Instead, it is more fruitful to try comparing MOCs along dimensions that are less subject to interpretation, and subsequently attempt to relate these dimensions to the wish list.

A matrix (Table 1) can be used to compare the MOCs of §2. Row labels are the MOCs listed previously. The column labels are the dimensions detailed in appendix A. For example, Pcur and Dcur describe the volatility and number of program and data cursors; Prog describes how programmable the model is; UM describes whether a universal machine exists; Pgen refers to program generation; SelfR refers to self-reproduction and Tcpl refers to Turing completeness.

We hope the comparisons cast more light than heat. The row ‘Cblob’ is an envisioned extension of our blob model; it is currently under development.

(b) Relating the bestiary to the wish list

Comparing the various MOCs reveals that none seems to satisfy all requirements for an ideal model of computation. For example, the rather stringent restriction of strong finiteness is not satisfied by FA or by Turing machines, owing to their unbounded control points (e.g. the \(p, q\) in a state transition \((p, a, q)\) or \((p, a, b, m, q)\)). One can, without loss of generality, assume that the tape alphabet size is uniformly finite, e.g. binary, but one cannot assume the existence of only a fixed finite number of many control states. Strong finiteness also rules out the \(\lambda\) calculus, and the CM, \(1\#\), RAM and RASP models, as well the von Neumann/Burks CA, again owing to the unbounded number of control states.

On the other hand, this requirement is satisfied by both LIFE [27, 42] and by our blob computation models [36, 37] (though in quite different ways). LIFE has only one state transition rule and that applies to all computations. A blob implementation of an arbitrary Turing machine can be made strongly finite, by the use of a fan-in tree to handle control points (details are given in the appendix of Hartmann et al. [36]).
5. Towards a better models of computation

(a) What we have and what we would like to have

The blob model of §3 and Hartman et al. [36,37] stands fairly high on the wish list of desirable qualities: there is only a fixed finite number of instructions, so it is strongly finite; it appears more physically realizable than many other models owing to the absence of pointers or other action at a distance; a universal machine has been constructed and programs are the same as data and inhabit the same computational world, so programs can, in principle, be generated, and stored along with the data.

There are, however, some missing desirable features. For example, blobs as yet have no mechanism for activating a generated program. Furthermore, in the current model, there can be only one program cursor and one data cursor active at any instant. In other words, the model is not concurrent.

The last MOC in the matrix, called Cblob, is speculative: it has not yet been formalized, but is envisioned to be essentially identical to the blob model, but with concurrency and program activation added. At the time of writing, this seems straightforward to do, but formal definitions and computer experiments have yet to be made and carried out.

(b) Conclusions

Alan Turing’s work opened new ways of thinking that led to modern computers, programming languages and a variety of MOCs of nearly botanical complexity. The analyses and classifications presented in this paper are a first step towards organizing and understanding some facets of this complexity. Our end goal remains a programmable, physically realizable model in which the standard theoretical results such as the recursion theorem and self-reproduction straightforwardly hold.

Appendix A. Some dimensions in models of computation

We pay special attention to two factors of prime importance to programmability and physical realizability: finiteness (and with respect to what) and binding times (of what to what at which point in a computation’s time). More specifically, we assign a number among 0,1,... for each MOC and each dimension. The finiteness factor is whether a dimension is absolutely finite, finite for any one program or infinite. The binding time factor is at which point in a computation’s time a value is fixed, i.e. bound.

(a) Explanation of the dimensions

Advice to the reader: consult the details below ‘by need’, rather than reading through them systematically.

ISet = instruction sets
1 = Fixed for all programs.
2 = Different for different programs.
Reasoning: assign 2 as a measure if different programs can have different unbounded building blocks, e.g. next states or register numbers. This does not count instruction addresses, or bond values in a blob program.

**Prep = program representation**
1 = None, e.g. FA.
2 = Codable, e.g. universal Turing machine.
3 = Executable, e.g. von Neumann or RASP or blob.

Meaning: can a program be represented so that it can be an input data value to other programs?

**Dsize = data space size**
1 = Fixed finite, e.g. von Neumann or FA.
2 = Input-dependent.
3 = Finite but expandible in any one run, e.g. a Turing machine with an extensible tape.
4 = Truly infinite, although only a finite ‘active zone’ exists at any moment, e.g. CA.

**Pcur = program control cursors**
1 = One control cursor, e.g. as in Turing machine/FA/RAM/…
2 = A finite statically determined number of program control points (program dependent), as in shared-memory concurrently executing programs.
3 = Finitely many control cursors at any one time, but expandible in any one run, as in multi-program threads.
4 = Unboundedly many control cursors kept in memory, e.g. in a stack or a heap, e.g. in programs with first-class continuations.
5 = There may be infinitely many control cursors in principle, although there is only a finite ‘active zone’ at any moment, as in CA.

**Dcur = data cursors**
1 = One, e.g. as in Turing machine/FA/RAM/….
2 = Many, but the number of data cursors is fixed by the program, e.g. CM or 1#.
3 = Expandible at run time.

**Prog = programmability**
0 = None, e.g. as in hardware.
1 = A little, e.g. as in field programmable gate array (FPGA).
2 = Implicit, as in LIFE (you can only set the start configuration).
3 = Explicit, as in von Neumann’s CA (the transition function may be chosen).
4 = Explicit, as in von Neumann computer architecture or the Turing machine.

**PhR = physical realizability**
1 = Actor and actee are physically adjacent, and interactions occur locally, without global synchronization.
2 = Data interactions occur locally, but with global synchronization.
3 = Not physically realizable, since it requires action at a distance, e.g. data pointers/addresses/indirect addressing; or quantum entanglement (an enchanting idea, but still only a dream on the level of programming).

**Daccess = data access**
1 = Read-only.
2 = Read-write.

**Bindings = (bindings of variables to values)**
1 = None, e.g. TM or blob model.
2 = Fixed for each program.
3 = Unbounded but there is only one active program-dependent environment, e.g. stacked bindings.
4 = Unbounded and many bindings can be active, e.g. multi-threaded computations with local memories.

**Pgen = program generation:** this requires the Prep property. Theoretical basis: the \( s-m-n \) theorem. Practical aspects include compiling and partial evaluation.
1 = Programs must be constructed by hand.
2 = Program code can be generated (but a loader is needed to make a program runnable).
3 = Load and go, i.e. code in memory can be executed immediately.

**UM = universal machine:** (now often called a ‘self-interpreter’). This requires the Prep property. The theoretical basis is computability of a \( \text{universal function} \). Practical aspects include program simulation. Relevant dimensions are
1 = The MOC does not have a universal function.
2 = The MOC does have a universal function.

**SelfR = self-reproduction**
1 = Kleene and Rogers results, from recursive function theory.
2 = CA, first von Neumann, then \( \text{LIFE} \).
3 = One can create one program, then activate it (same as Pgen point 3).
4 = Akin to biological self-reproduction (a program can create and activate many programs, e.g. clones).

**Tcpl = Turing complete**
1 = No.
2 = yes.

*Special entry forms in the matrix column UM for universal machines*

- Entry [2]: a universal machine can be built but is of limited generality, owing to the finite word size and address space of a von Neumann computer.
- The entries (2) indicate that while universal machines can be constructed they are exceedingly indirect, since programs must be encoded via Gödel numbers.
- Entries 1?: the authors have not seen a universal CA, and it is not clear how one could be devised. (Note. von Neumann showed something different: that a CA could simulate a universal Turing machine.)
Programs = data = first-class citizens

— Life seems unlikely to have a natural universal machine, since it lacks program generation or activation. Further, a sensible way to compare the abilities of Life and, say, PTIME, is not immediately evident.

References


Phil. Trans. R. Soc. A (2012)


von Neumann, J. 1945 First draft of a report on the EDVAC. University of Pennsylvania Moore School of Electrical Engineering.


