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, DK-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 systematise 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*. Novelties: 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 realisable than earlier computation models. In part this owes to *strong finiteness* due 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, 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 1936 paper from a programming perspective

First and foremost, Turing made in (40) 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.

Turing's machine had a concept of *control state*, there 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

[†] Email of corresponding author: `simonsen@diku.dk`

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 nontrivial 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. Further, 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 realisable, 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 Section 2, we review some developments occurring after (40) to see how they relate to this goal. In Section 3, we briefly overview our recently developed "blob" model. In Sections 4, 5, 6 we study more systematically how models of computation may be compared and evaluated.

Two meanings of the word "model". This word is widely used, but with two very different meanings. A first analytic 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 center on constructing an artifact (e.g., a computer program or a hardware device) that is faithful to a *problem specification.* The constructed artifact is a good or bad model insofar as it *satisfies* the specification. If so, the program or hardware device may be used in secure knowledge that it will behave as required.

Turing's 1936 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 in (16).

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, Turing (42).

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 of the field of *computability theory*, hand in hand with the description of computation by *binding names to values*, were both already in their early stages at the time Turing did his work. Artifacts 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 PTIME (11; 21). The advent of physical computing devices led to he 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 (25). 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's 1936 was pioneering: The development of programming as a general problem solving technique, requiring *programming languages*; and the development of *models of computation* 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 is 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 realise) 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 models of computation have downplayed the programmability aspect, resulting in machines with considerable computational power, but 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 lambda-calculus (9); later, McCarthy devised the programming language LISP closely based on it (31). Further, 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 (28).

More sophisticated languages, beginning with ALGOL 60 (35), embodied concepts for algorithm development and organisation 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 (40) on Hilbert's *Entscheidungsproblem* contained a sketch proof that the set of partial functions $\mathbb{N} \rightharpoonup \mathbb{N}$ 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 (10). Turing's 1937 paper (41) provided a full proof this equivalence as well as the equivalence of the Turing machine with Kleene's *recursive function theory* (26).

Another accomplishment: 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 constructing *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* (16). For reference works on computability theory, see Rogers (38) and Jones (24).

We very briefly state some fundamental assumptions and theorems of computability theory. Notation: write $\mathbb{D}$ for the set of all data values, and Programs for the set of all programs, where we assume that Programs $\subseteq \mathbb{D}$ (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 $[\![p]\!](d)$ represents the result of p's computation (possibly undefined, if p fails to terminate).

Existence of *Turing's Universal Machine*:

$$\exists U \in \text{Programs}\ \forall p \in \text{Programs}\ \forall d \in \mathbb{D} : [\![p]\!](d) = [\![U]\!](p, d)$$

The *s-m-n Theorem*, for the case $m = n = 1$:

$$\exists s_1^1 \in \text{Programs } \forall p \in \text{Programs } \forall d_1, d_2 \in \mathbb{D} : [\![p]\!](d_1, d_2) = [\![[\![s_1^1]\!](p, d_1)]\!](d_2)$$

Kleene's Second recursion Theorem:

$$\forall p \in \text{Programs } \exists e \in \text{Programs } \forall d \in \mathbb{D} : [\![p]\!](e, d) = [\![e]\!](d)$$

Programming language researchers have computer-implemented all these ideas: specifically the Universal Machine is a *self-interpreter* (25), the *s-m-n* theorem is the theoretical basis for *partial evaluation*, cf. (25), and computer implementations of the Recursion Theorem have been performed, see (34; 20).

These main results from recursive function theory are present in *all* Turing-complete models of computation (38), 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 models of computation (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 and Scott (37) 1959.

von Neumann architecture Draft circulated by von Neumann (44) 1945.

Lambda Calculus (λ) Untyped lambda calculus by Church (10) 1936.

Counter Machine (CM) Informal in 1950s; Lambek (29) and Minsky (33) 1961.

Text Register Machine (1#) Moss (34) 2006. A more programmable basis for computability theory.

Random Access Machine (RAM) Register machine with indirect addressing by Wang 1957 (46). Final definition by Cook and Reckhow (12) 1973.

Random Access Stored Program (RASP) Elgot and Robinson (15) 1964.

Cellular automaton (CA) Von Neumann *et al* in 1940s; (45) 1951.

LIFE Conway, published in popular science article by Gardner (17) 1970.

Abstract state machine (ASM) Gurevich, Börger, et al (14).

Reaction-diffusion machine (RDM) Adamatzky et al. (3; 5; 1) 2004/2005. Related : Turing's 1952 paper (43) and the chemical abstract machine (6) (1992).

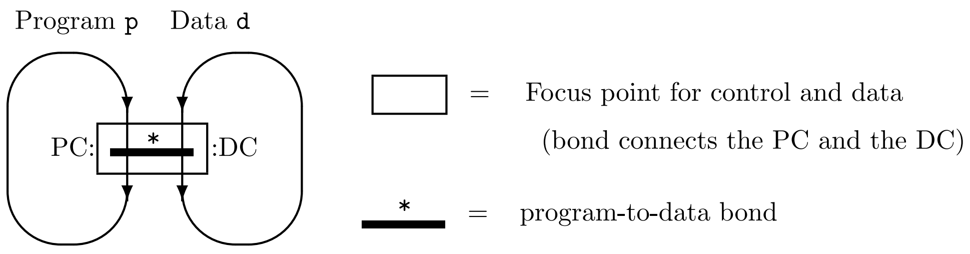
Blob Hartmann, Jones, Simonsen (22; 23) 2010. Yet another model, described next.

3. A biologically motivated model of computation

The *Blob model*† introduced by Hartmann *et al.* (22; 4) is a biologically-inspired model of computation specifically tailored to afford Turing completeness and 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. 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.

Running program `p`, i.e., computing $[\![\mathtt{p}]\!](\mathtt{d})$



Program p
Data d
PC:
*
:DC
= Focus point for control and data
(bond connects the PC and the DC)
*
= program-to-data bond

Before reading on, we urge the reader to consult the movie included as electronic supplementary material at the journal web site; the following material will be much easier to understand after viewing the movie at `http://dk.diku.blob.blobvis.s3.amazonaws.com/largedata-play.avi`.
to see the actual execution of a blob program. (PC = the dark green blob, and DC = 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 4 adjacent neighbors†, 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 $[\![p]\!](d)$ we have:

- *One* blob in p is active, known as the *program cursor* or PC.
- *One* blob in d is active, known as the *data cursor* or DC.
- A bond `*`, between the PC and the DC, is linked at a designated bond site (bond site 0 of each).

† Our blob model (22; 4) is not to be confused with another MOC of the same name, independently developed by Gruau and others (19; 18).

† While this may seem at first sight to resemble the global state of a cellular automaton, there are essential differences: a) the blobs are not arranged in a 2-dimensional grid; b) there is no universal clock to cause all cells to perform their state transitions synchronously; and c) blobs are *uniformly programmable* in the sense that there is a single set of transition rules for all programs.

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 8 *cargo bits* of local storage containing Boolean values, and also identified by numerical positions: $0, 1, 2, \ldots, 7$. 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 7 bits may be used as the user wishes.

For program execution, one of the 8 cargo bits is an "activation bit"; if 1, it marks the instruction currently being executed. The remaining 7 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:

```
operationcode parameters (bond0, bond1, bond2, bond3)
```

The operation code and parameters together occupy 7 cargo bits; the rest of the instruction is the 4 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. Further, a bond is not coded as an address (i.e., a bit string). The 4 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

- *Move*: move the data cursor along bond 1 (or bond 2 or 3)
- *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 4 bond sites and 8 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 3 bond sites per blob seem to be needed for Turing completeness. The choice of 8 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 (22) 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 (13).

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. (22) shows 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: 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 MOC; and playing Linnaeus

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

Existence of programs : There should exist programs in the MOC. This cannot be taken for granted. There do exist contexts where it is hard to see any clearly identified program, even though constructing a state transition function for a finite automaton or von Neumann-style cellular automaton may be thought of as programming. Examples without evident programs include LIFE (with one *a priori fixed* transition function for all programs); and various biological computation models including the reaction-diffusion model (43; 3).

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 should be enough to have *one* set of hardware, so one doesn’t 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 : Mechanical *readability* of programs is a prerequisite for existence of a universal machine. A next step is that programs be mechanically *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 (44), and the RASP model (15). While not now explicitly present in the blob model of (22) this ability should be straightforward to add, since a program *is* a collection of blobs.

Concurrency : Would imply that 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. In particular PTIME, NPTIME, PSPACE (or their parallel counterparts) should be stable classes.

(*a*) *Playing Linneaus*

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

A matrix can be used to compare the MOCs of Section 2. Row labels are the MOCs listed previously. The column labels are the dimensions detailed in Appendix 6. For example, Pcur and Dcur describe the volatility and number of program and data cursors; Prog describes how programmable the model is; UM, whether a universal machine exists; Pgen refers to program generation; SelfR to self-reproduction; and Tcpl 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.

MOC	Iset	Prep	Dsize	Pcur	Dcur	Prog	PhR	Dacc	Bind	UM	Pgen	SelfR	Tcpl
FA	2	1	1	1	1	4	1	1	1	1	1	0	no
vN arch	1	3	1	1	3	4	3	2	3	[2]	3	3	no
TM	2	2	3	1	1	4	1	2	1	2	2	1	yes
λ	1	2	3	1	2	4	3	1	3	2	2	1	yes
CM	2	(2)	3	1	2	4	1	2	2	(2)	1	1	yes
`1#`	2	2	3	1	2	4	1	2	2	2	2	1	yes
RAM	2	2	3	1	1	4	3	2	2	2	2	2	yes
RASP	2	3	3	1	1	4	3	2	2	2	3	2	yes
CA	2	1	4	4	0	3	2	1	1	1?	1	2	yes
LIFE	1	1	4	4	0	2	2	1	1	1?	1	2	yes
ASM	2	2	3	3	2	4	3	2	2	2	2	3	yes
RDM	2	1	4	4	0	3	1	1	1	1	1	1	yes
Blob	1	3	3	1	1	4	1	2	1	2	2	1	yes
Cblob	1	3	3	3	3	4	1	2	1	2	3	4	yes

(*b*) *Relating the bestiary to the wishlist*

Comparing the various MOCs reveals that none seem to satisfy all requirements for an ideal model of computation. For example, the rather stringent restriction of strong finiteness is not satisfied by finite automata or by Turing machines, due 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 the tape alphabet size is uniformly finite, e.g., binary; but one cannot assume existence of only a fixed, finite bound on the number of control states. Strong finiteness also rules out the λ-calculus, and the CM, `1#`, RAM and RASP models, and as well the von Neumann/Burks cellular automata, again due to the unbounded numbers of control states.

On the other hand, this requirement *is satisfied* by both LIFE (17; 2) and by our blob computation models (22; 23) (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 (22)).

5. Towards a better MOC

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

The Blob model of Section 3 and papers (22; 23) stands fairly high on the wishlist of desirable qualities: there is only a fixed finite number of instructions so it is strongly finite; it appears more physically realisable than many other models due 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. Further, 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 models of computation 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 realisable* model in which the standard theoretical results such as the recursion theorem and self-reproduction *straightforwardly hold.*

6. Appendix: some dimensions in models of computation

We pay special attention to two factors of prime importance to programmability and physical realisability: *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 identifies whether a dimension 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.

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 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., finite automata.
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, e.g. a pushdown automaton
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., cellular automaton

Pcur = Program control cursors:
1 = One control cursor, e.g. as in TM/FA/PDA/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 multiprogram threads.
4 = Unboundedly many control cursors kept in memory, e.g., in a stack or the 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 cellular automata.

Dcur = Data cursors:
1 = One, e.g. as in TM/FA/PDA/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 FPGA (field programmable gate array).
2 = Implicit, as in LIFE (you can only set the start configuration).
3 = Explicit, as in von Neumann's cellular automata (the transition function may be chosen).
4 = Explicit, as in von Neumann computer arhitecture, or the Turing machine.

PhR = Physical Realisability:
1 = Actor and actee are physically adjacent, and interactions occur locally, without global synchronisation
2 = Data interactions occur locally, but with global synchronisation
3 = Not physically realisable, 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., multithreaded 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; or
2 = Program code can be generated (but a loader is needed to make a program runnable); or
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 *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 = Cellular automata, first von Neumann, then 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 cases in the matrix column for universal machines:

- Entry [2]: a universal machine can be built but is of limited generality, due to the finite word size and address space of a von Neumann computer
- The entry (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 cellular automaton, 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 (45).)
- 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, complexity class PTIME, is not immediately evident.

REFERENCES

[1] A. Adamatzky. Physarum machine: Implementation of a Kolmogorov-Uspensky machine on a biological substrate. *Parallel Processing Letters*, 17(4):455–467, 2007.

[2] A. Adamatzky. *Game of Life Cellular Automata.* Springer-Verlag, 2010.

[3] A. Adamatzky, B. de Lacy Costello, and T. Asai. *Reaction-diffusion computers.* Elsevier, 2005.

[4] G. Agha, O. Danvy, and J. Meseguer, editors. *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, volume 7000 of *Lecture Notes in Computer Science.* Springer, 2011.

[5] S. Bandini, G. Mauri, G. Pavesi, and C. Simone. Computing with a distributed reaction-diffusion model. In Margenstern (30), pages 93–103.

[6] G. Berry and G. Boudol. The chemical abstract machine. *Theor. Comput. Sci.*, 96(1):217–248, 1992.

[7] S. J. Brams, K. Pruhs, and G. J. Woeginger, editors. *Fair Division, 24.06. - 29.06.2007*, volume 07261 of *Dagstuhl Seminar Proceedings.* Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[8] C. Calude, M. J. Dinneen, and F. Peper, editors. *Unconventional Models of Computation, Third International Conference, UMC 2002, Kobe, Japan, October 15-19, 2002, Proceedings*, volume 2509 of *Lecture Notes in Computer Science.* Springer, 2002.

[9] A. Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936.

[10] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[11] A. Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the 1964 International Congress on Logic, Methodology, and Philosophy of Science*, pages 24–30. North Holland, 1965.

[12] S. A. Cook and R. A. Reckhow. Time-bounded random access machines. *Journal of Computer and Systems Sciences*, 7(4):354–375, 1973.

[13] V. Danos and C. Laneve. Formal molecular biology. *Theor. Comp. Science*, 325:69 – 110, 2004.

[14] S. Dexter, P. Doyle, and Y. Gurevich. Gurevich abstract state machines and Schoenhage storage modification machines. *J. UCS*, 3(4):279–303, 1997.

[15] C. Elgot and A. Robinson. Random-access stored-program machines, an approach to programming languages. *Journal of the Association for Computing Machinery*, 11(4):365–399, 1964.

[16] R. Gandy. The confluence of ideas in 1936. In R. Herken, editor, *The Universal Turing Machine–A Half-Century Survey*, pages 51–102. Springer Verlag, 1995.

[17] M. Gardner. Mathematical games—the fantastic combinations of John Conway's new solitaire game LIFE. *Scientific American*, 223:120–123, 1970.

[18] F. Gruau and C. Eisenbeis. Programming self developing blob machines for spatial computing. In Brams et al. (7).

[19] F. Gruau and P. Malbos. The blob: A basic topological concept for "hardware-free" distributed computation. In Calude et al. (8), pages 151–163.

[20] T. A. Hansen, T. Nikolajsen, J. L. Träff, and N. D. Jones. Experiments with implementations of two theoretical constructions. In Meyer and Taitslin (32), pages 119–133.

[21] J. Hartmanis and R. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[22] L. Hartmann, N. Jones, J. Simonsen, and S. Vrist. Programming in biomolecular computation: Programs, self-interpretation and visualisation. *Scientific Annals of Computer Science*, 21:73–106, 2011.

[23] L. Hartmann, N. D. Jones, J. G. Simonsen, and S. B. Vrist. Computational biology: A programming perspective. In Agha et al. (4), pages 403–433.

[24] N. D. Jones. *Computability and Complexity from a Programming Perspective.* Foundations of Computing. MIT Press, Boston, London, 1 edition, 1997.

[25] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation.* Prentice Hall international series in computer science. Prentice Hall, 1993.

[26] S. C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112:727–747, 1935–1936.

[27] S. C. Kleene. On notation for ordinal numbers. *Journal of Symbolic Logic*, 3(4):150–155, 1938.

[28] D. E. Knuth and L. T. Pardo. Early development of programming languages. In J. Belzer, A. G. Holzman, and A. Kent, editors, *Encyclopedia of Computer Science and Technology*, volume 7, pages 419–493. Marcel Dekker Inc., 1977.

[29] J. Lambek. How to program an infinite abacus. *Mathematical bulletin*, 4(3):295–302, 1961.

[30] M. Margenstern, editor. *Machines, Computations, and Universality, 4th International Conference, MCU 2004, Saint Petersburg, Russia, September 21-24, 2004, Revised Selected Papers*, volume 3354 of *Lecture Notes in Computer Science.* Springer, 2005.

[31] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.

[32] A. R. Meyer and M. A. Taitslin, editors. *Logic at Botik '89, Symposium on Logical Foundations of Computer Science, Pereslav-Zalessky, USSR, July 3-8, 1989, Proceedings*, volume 363 of *Lecture Notes in Computer Science*. Springer, 1989.

[33] M. Minsky. Recursive unsolvability of Post's problem of 'tag'. *Annals of Mathematics*, 74(3):437–455, 1961.

[34] L. S. Moss. Recursion theorems and self-replication via text register machine programs. *Bulletin of the European Association for Theoretical Computer Science*, 89:171–182, 2006.

[35] P. Naur, J. Backus, F. Bauer, J. Green, C. Katz, J. McCarthy, A. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963.

[36] A. G. Oettinger. Automatic syntactic analysis and the pushdown store. *Proceedings of Symposia on Applied Mathematics*, 12, 1961.

[37] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:114–125, 1959.

[38] H. Rogers Jr. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

[39] M.-P. Schützenberger. On context-free languages and pushdown automata. *Information and Control*, 6(3):246–264, 1963.

[40] A. M. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[41] A. M. Turing. Computability and lambda-definability. *Journal of Symbolic Logic*, 2(4):153–163, 1937.

[42] A. M. Turing. Programmers' handbook for the Manchester electronic computer. Technical report, Manchester University Computing Laboratory, 1950.

[43] A. M. Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London, series B*, 641:37–72, 1952.

[44] J. von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, 1945.

[45] J. von Neumann. The general and logical theory of automata. In *Proceedings of the Hixon Symposium*, pages 1–31. John Wiley and Sons, 1951.

[46] H. Wang. A variant to Turing's theory of computing machines. *Journal of the Association for Computing Machinery*, 4(1):63–98, 1957.