Programming in Biomolecular Computation

Lars Hartmann, Neil D. Jones, Jakob Grue Simonsen[*]

{hartmann,neil,simonsen}@diku.dk
Department of Computer Science, University of Copenhagen (DIKU),
Copenhagen, Denmark

Abstract. Our goal is to provide a top-down approach to biomolecular computation. In spite of widespread discussion about connections between biology and computation, one question seems notable by its absence: **Where are the programs?** We introduce a model of computation that is evidently *programmable*, by programs reminiscent of low-level computer machine code; and at the same time *biologically plausible*: its functioning is defined by a single and relatively small set of chemical-like reaction rules. Further properties: the model is *stored-program*: programs are the same as data, so programs are not only executable, but are also compilable and interpretable. It is *universal*: all computable functions can be computed (in natural ways and without arcane encodings of data and algorithm); it is also *uniform*: new "hardware" is not needed to solve new problems; and (last but not least) it is *Turing complete* in a strong sense: a universal algorithm exists, that is able to execute any program, and is not asymptotically inefficient.
A prototype model has been implemented (for now in silico on a conventional computer). This work opens new perspectives on just how computation may be specified at the biological level.

Keywords: biomolecular, computation, programmability, universality.

1 Biochemical universality and programming

It has been known for some time that various forms of biomolecular computation are Turing complete [7,8,10,12,25,29,32,33]. The net effect is to show that any computable function can be computed, in some appropriate sense, by an instance of the biological mechanism being studied. However, the arguments for Turing universality we have seen are less than compelling from a *programming* perspective. This paper's purpose is to provide a better computation model where the concept of "program" is clearly visible and natural, and in which Turing completeness is not artificial, but rather a natural part of biomolecular computation. We begin by evaluating some established results on biomolecular computational completeness from a programming perspective; and then constructively provide an alternative solution. The new model seems biologically plausible, and usable for solving a variety of problems of computational as well as biological interest.

[*] © 2010 Published by Elsevier Science B. V.

It should be noted that while our model can support full parallelism (as often seen in biologically-inspired computing), it is not the foci of the paper, which are completeness and universality: we consider *one* program running on *one*, contiguous piece of data.

The central question: can program execution take place in a biological context? Evidence for "yes" includes many analogies between biological processes and the world of programs: *program-like behavior*, e.g., genes that direct protein fabrication; "switching on" and "switching off"; processes; and reproduction.

A clarification from the start: this paper takes a *synthetic* viewpoint, concerned with building things as in the engineering and computer sciences. This is in contrast to the ubiquitous *analytic* viewpoint common to the natural sciences, concerned with finding out how naturally evolved things work.

The authors' backgrounds lie in the semantics of programming languages, compilers, and computability and complexity theory; and admittedly not biology. We focus on the synthetic question **can**, rather than the usual natural scientists' analytical question **does**.

Where are the programs? In existing biomolecular computation models it is very hard to see anything like a program that realises or directs a computational process. For instance, in cellular automata the program is expressed only in the initial cell configuration, or in the global transition function. In many biocomputation papers the authors, given a problem, cleverly devise a biomolecular system that can solve this particular problem. However, the algorithm being implemented is hidden in the details of the system's construction, and hard to see, so the program or algorithm is in no sense a "first-class citizen". Our purpose is to fill this gap, to establish a biologically feasible framework in which programs are first-class citizens.

2 Relation to other computational frameworks

We put our contributions in context by quickly summarising some other computational completeness frameworks. **Key dimensions**: uniformity; programmability; efficiency; simplicity; universality; and biological plausibility. (Not every model is discussed from every dimension, e.g., a model weak on a dimension early in the list need not be considered for biological plausibility.)

Circuits, BDDs, finite automata. While well proven in engineering practice, these models don't satisfy our goal of computational completeness. The reason: they are *non-uniform* and so not Turing complete. Any single instance of a circuit or a BDD or a finite automaton has a control space and memory that are both finite. Consequently, any *general but unbounded* computational problem (e.g., multiplying two arbitrarily large integers) must be done by choosing one among an infinite family of circuits, BDDs or automata.

The Turing machine. *Strong points.* Highly successful for theoretical purposes, the Turing model is uniform; there exists a clear concept of "program";

and the "universal Turing machine" from 1936 is the seminal example of a self-interpreter. The Turing model has fruitfully been used to study computational complexity problem classes as small as PTIME and LOGSPACE.

Weak points. Turing machines do not well model computation times small enough to be realistically interesting, e.g., near-linear time. The inbuilt "data transport" problems due to the model's one-dimensional tape (or tapes, on a multi-tape variant) mean that naturally efficient algorithms may be difficult to program on a Turing machine. E.g., a time $O(n)$ algorithm may suffer *asymptotic slowdown* when implemented on a Turing machine, e.g., forced to run in time $O(n^2)$ because of architectural limitations. A *universal Turing machine* has essentially the same problem: it typically runs quadratically slower than the program it is simulating. Stiull greater slowdowns may occur if one uses smaller Turing complete languages, for instance the counter or Minsky register machines as used in [7,8,12,22].

Other computation models with an explicit concept of program. Numerous alternatives to the Turing machine have been developed, e.g., the Tag systems studied by Post and Minsky, and a variety of register or counter machines. Closer to computer science are recursive functions; the λ-calculus; functional programming languages such as LISP; and machines with randomly addressable memories including the RAM and, most relevant to our work, its stored-program variant the RASP [19]. These models rate well on some of the key dimensions listed above. However they are rather complex; and were certainly not designed with biological plausibility in mind.

Cellular automata. John von Neumann's pathbreaking work on cellular automata was done in the 1940s, at around the time he also invented today's digital computer. In [29] computational completeness was established by showing that any Turing machine could be simulated by a cellular automaton. Further, it was painstakingly and convincingly argued that a cellular automaton could achieve self-reproduction.Von Neumann's and subsequent cellular automaton models, e.g., LIFE and Wolfram's models[15,8,32], have some shortcomings, though. Though recent advances have remedied the lack of asynchronous computations [23], a second, serious drawback is the lack of *programmability*: once the global transition function has been selected (e.g., there is only one such in LIFE) there is little more that the user of the system can do; the only degree of freedom remaining is to choose the initial configuration of cell states. There is no explicit concept of a program that can be devised by the user. Rather, any algorithmic ideas have to be encoded in a highly indirect manner, into either the global transition function or into the initial cell state configuration; in a sense, the initial state is *both* program and input, but in the zoo of cellular automata proven to be universal, there seems to be no clear way to identify which parts of the initial state of the CA corresponds to, say, a certain control structure in a program, or a specific substructure of a data structure such as a list.

Biomolecular computation frameworks. We will see that the Turing-typical asymptotic slowdowns can be avoided while using a biomolecular computing model. This provides an advance over both earlier work on automata-based computation models (Turing machines, counter machines, etc.), and over some other approaches to biomolecular computing

A number of contributions exist in this area; a non-exhaustive list: [1,3,7,10,8,11,12,17,20,21,25,26,30,31,5,33] The list is rather mixed: Several of the articles describe concrete finite-automaton-like computations, emphasising their realisation in actual biochemical laboratory contexts. As such their emphasis is not on general computations but rather on showing feasibility of specific computations in the laboratory. Articles [7,8,12,20,33] directly address Turing completeness, but the algorithmic or programming aspects are not easy to see.

How our approach is different: Contrary to several existing models, our atomic notion (the "blob") carries a *fixed* amount of data and has a *fixed* number of possible interaction points with other blobs. Further, one *fixed* set of rules specify how local collections of blobs are changed. In this sense, our setup resembles specific cellular automata, e.g. Conway's game of life where only the initial state may vary. Contrary to cellular automata, there is both programs and data are very clearly identified ensembles of blobs. Further, we use a textual representation of programs closely resembling machine code such that each line essentially corresponds to a single blob instruction with parameters and bonds. The resulting code conforms closely to traditional low-level programming concepts, including use of conditionals and jumps.

Outline of the paper: Section 3 introduces some notation to describe program execution. Section 4 concerns the *blob model* of computation, with an explicit program component. Section 5 relates the blob model to more traditional computation models, and Section 6 concludes. Appendix A has more discussion of computational completeness; and Appendix B shows how a Turing machine may be simulated in the blob model – doable within a constant slowdown because of the flexibility of blobs when considered as data structures. Appendix C discusses the blob model's realisability in 3-dimensional space.

3 Notations: direct or interpretive program execution

What do we mean by a program (roughly)? An answer: a *set of instructions* that specify *a series (or set) of actions on data.* Actions are carried out when the instructions are executed (activated,...) Further, a program is software, not hardware. Thus a program should itself be a *concrete data object that can be replaced* to specify different actions.

Direct program execution: write $[\![\texttt{program}]\!]$ to denote the meaning or net effect of running `program`. A program meaning is often a function from data input values to output values. Expressed symbolically:

$$[\![\texttt{program}]\!](\texttt{data}_{in}) = \texttt{data}_{out}$$

The `program` is *activated* (run, executed) by applying the semantic function $[\![\_]\!]$. The *task of programming* is, given a desired semantic meaning, to find a program that computes it. Some mechanism is needed to execute `program`, i.e., to compute $[\![\texttt{program}]\!]$. This can be done either by hardware or by software.

Interpretive program execution: Here `program` is a passive data object, but it is now activated by running the *interpreter program.* (Of course, some mechanism will be needed to run the interpreter program, e.g., hardware or software.) An equation similar to the above describes the effect of interpretive execution:

$$[\![\texttt{interpreter}]\!](\texttt{program}, \texttt{data}_{in}) = \texttt{data}_{out}$$

Note that `program` is now used as data, and not as an active agent. Self-interpretation is possible and useful [18]; the same value $\texttt{data}_{out}$ can be computed by:

$$[\![\texttt{interpreter}]\!](\texttt{interpreter}, (\texttt{program}, \texttt{data}_{in})) = \texttt{data}_{out}$$

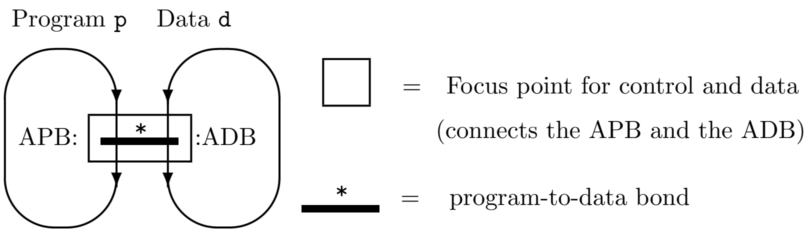
4 Programs in a biochemical world

Our goal is to express programs in a biochemical world. Programming assumptions based on silicon hardware must be radically re-examined to fit into a biochemical framework. We briefly summarize some qualitative differences.

- **There can be no pointers to data:** addresses, links, or unlimited list pointers. In order to be acted upon, a data value must be *physically adjacent* to some form of actuator. A biochemical form of adjacency: a chemical bond between program and data.
- **There can be no action at a distance**: all effects must be achieved via chains of local interactions. A biological analog: signaling.
- **There can be no nonlocal control transfer**, e.g., no analog to GOTOs or remote function/procedure calls. However some control loops are acceptable, provided the "repeat point" is (physically) near the loop end. A biological analog: a bond between different parts of the same program.
- On the other hand there exist available **biochemical resources** to tap, i.e., free energy so actions can be carried out, e.g., to construct local data, to change the program control point, or to add local bonds into an existing data structure. Biological analogs: Brownian movement, ATP, oxygen.

The above constraints suggest how to structure a biologically feasible model of computation. The main idea is to keep both 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 active program and data always in reach of the focus. The picture illustrates this idea for direct program execution.

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



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

4.1 The Blob model

We take a very simplified view of a (macro-)molecule and its interactions, with abstraction level similar to the Kappa model [12,7,14]. To avoid misleading detail questions about real molecules we use the generic term "blob" for an abstract molecule. A collection of blobs in the biological "soup" may be interconnected by two-way *bonds* linking the individual blobs' *bond sites*.

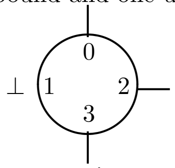
A *program p* is (by definition) a connected assembly of blobs. A data value d is (also) by definition a connected assembly of blobs. At any moment during execution, i.e., during computation of $[\![p]\!](d)$ we have:

- *One* blob in p is active, known as the *active program blob* or APB.
- *One* blob in d is active, known as the *active data blob* or ADB.
- A bond $*$, between the APB and the ADB, is linked at bond site 0 of each.

The data view of blobs: 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 can hold a bond – that is, a link to a (different) blob; or a bond 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.

A blob with 3 bond sites bound and one unbound:



Since bonds are in essence two-way pointers, they have a "fan-in" restriction: a given bond site can contain at most one bond (if not $\perp$).

The program view of blobs: Blob programs are sequential. There is no structural distinction between blobs used as data and blobs used as program.

A single, fixed set of instructions is available for moving and rearranging the cursors, and for testing or setting a cargo bit at the data cursor. Novelties from a computer science viewpoint: there are no explicit program or data addresses, just adjacent blobs. At any moment there is only *a single program cursor* and *a single data cursor*, connected by a bond written `*` above.

Instructions, in general. The blob instructions correspond roughly to "four-address code" for a von Neumann-style computer. An essential difference, though, 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. A blob's 4 bond sites contain links to other instructions, or to data via the APB-ADB bond `*`.

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. An instruction has an *operation code* (around 15 possibilities), and 0, 1 or 2 *parameters* that identify single bits, or bond sites, or cargo bits in a blob. See table below for current details. For example, `SCG v c` has 16 different versions since `v` can be one of 2 values, and `c` can be one of 8 values.

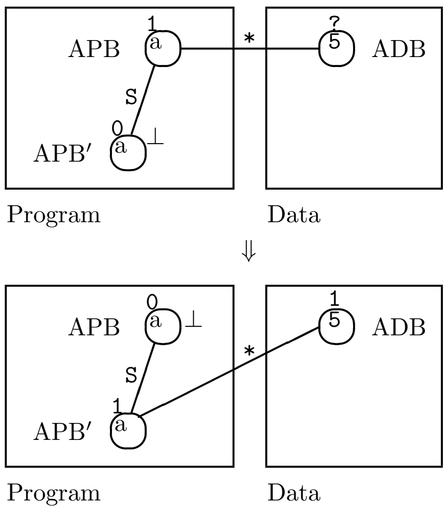
Why exactly 4 bonds? The reason is that each instruction must have a bond to its predecessor; further, a test or "jump" instruction will have two successor bonds (true and false); and finally, there must be one bond to link the APB and the ADB, i.e., the bond `*` between the currently executing instruction and the currently visible data blob. The `FIN` instruction is a device to allow a locally limited fan-in.

A specific instruction set (a bit arbitrary)**.** The formal semantics of instruction execution are specified precisely by means of a set of 128 biochemical reaction rules in the style of [12]. For brevity here, we just list the individual instruction formats and their informal semantics. Notation: `b` is a 2-bit bond site number, `c` is a 3-bit cargo site number, and `v` is a 1-bit value.

Numbering convention: the program APB and the data ADB are linked by bond `*` between bond sites 0 of the APB and the ADB. An instruction's predecessor is linked to its bond site 1; bond site 2 is the instruction's normal successor; and bond site 3 is the alternative "false" successor, used by jump instructions that test the value of a cargo bit or the presence of a bond.

Instruction	Description	Informal semantics (:=: is a two-way interchange)
`SCG v c`	Set CarGo bit	ADB.c := v; APB := APB.2
`JCG c`	Jump CarGo bit	if ADB.c = 0 then APB := APB.3 else APB := APB.2
`JB b`	Jump Bond	if ADB.b = ⊥ then APB := APB.3 else APB := APB.2
`CHD b`	CHange Data	ADB := ADB.b; APB := APB.2
`INS b1 b2`	INSert new bond	new.b2 :=: ADB.b1; new.b1 :=: ADB.b1.bs; APB := APB.2 Here "new" is a fresh blob, and "bs" is the bond site to which ADB.b1 was bound before executing `INS b1 b2`.
`SWL b1 b2`	SWap Links	ADB.b1 :=: ADB.b2.b1; APB := APB.2
`SBS b1 b2`	SWap Bond Sites	ADB.b1 :=: ADB.b2; APB := APB.2
`SWP1 b1 b2`	Swap bs1 on linked	ADB.b1.1 :=: ADB.b2.1; APB := APB.2
`SWP3 b1 b2`	Swap bs3 on linked	ADB.b1.3 :=: ADB.b2.3; APB := APB.2
`JN b1 b2`	Join b1 to linked b2	ADB.b1 :=: ADB.b1.b2; APB := APB.2
`DBS b`	Destination bond site	Cargo bits 0,1 := bond site number of destination for ADB.b
`FIN`	Fan IN	APB := APB.2 (bond site 3 is an alternative predecessor)
`EXT`	EXiT program	

An example in detail: the instruction `SCG 1 5`, as picture and as a rewrite rule. SCG stands for "set cargo bit". The effect of instruction `SCG 1 5` is to change the 5-th cargo bit of the ADB (active data blob) to 1. First, an informal picture to show its effect:



Note: the APB-ADB bond * has moved: Before execution, it connected APB with ADB. After execution, it connects APB′ with ADB, where APB′ is the next instruction: the successor (via bond `S`) of the previous APB. Also note that the activation bit has changed: before, it was 1 at APB (indicating that the APB was about to be executed) and 0 at ADB′. Afterwards, those two bit values have been interchanged.

Syntax: Code the above instruction as an 8-bit string: $\overbrace{1}^{a}\overbrace{100}^{SCG}\overbrace{1}^{v}\overbrace{101}^{c}$. Here activation bit $a = 1$ indicates that this is the current instruction (about to be executed). Operation code `SCG` (happens to be) encoded as 100; and binary numbers are used to express the new value: $v = 1$, and the number of the cargo bit to be set: $c = 5$.

The instruction also has four bond sites: $*PS\bot$. Here P is a bond to the predecessor of instruction `SCG 1 5`, S is a bond to its successor, and bond site 3 is not used. The full instruction, with 8 cargo sites and four bond sites can be written in form[1]: $B[11001101](*PS\bot)$.

Semantics: Instruction `SCG 1 5` transforms the three blobs APB, APB′ and ADB as in the picture above. This can be expressed more exactly using a rewrite rule as in [12] that takes three members of the blob species into three modified ones. For brevity we write "-" at bond sites or cargo sites that are not modified by the rule. Remark: the labels APB, ADB, etc. are not part of the formalism, just labels added to help the reader.

$$\overbrace{B[1\ 100\ 1\ 101](*\text{-}S\text{-})}^{APB},\ \overbrace{B[0\text{-------}](\bot S\text{--})}^{APB'},\ \overbrace{B[0\text{----}x\text{--}](*\text{---})}^{ADB}$$
$$\Rightarrow$$
$$\underbrace{B[0\ 100\ 1\ 101](\bot\text{-}S\text{-})}_{APB},\ \underbrace{B[1\text{-------}](\bot S\text{--})}_{APB'},\ \underbrace{B[0\text{----}1\text{--}](*\text{---})}_{ADB}$$

5 The blob world from a computer science perspective

First, an operational image: Any well-formed blob program, while running, is a collection of program blobs that is adjacent to a collection of data blobs, such that there is *one* critical bond (`*`) that links the APD and the ADB (the active program blob and the active data blob). As the computation proceeds, the program or data may move about, e.g., rotate as needed to keep their contact points adjacent (the APB and the ADB). For now, we shall not worry about the thermodynamic efficiency of moving arbitrarily large program and data in this way; for most realistic programs, we assume them to be sufficiently small (on the order of thousands of blobs) that energy considerations and blob coherence are not an issue.

5.1 The blob language

It is certainly small: around 15 operation codes (for a total of 128 instructions if parameters are included). Further, the set is irredundant in that no instruction's effect can be achieved by a combination of other instructions. There are easy

[1] B stands for a member of the blob "species".

computational tasks that simply cannot be performed by any program without, say, `SCG` or `FIN`.

There is certainly a close analogy between blob programs and a *rudimentary machine language*. However a bond is not an address, but closer to a two-way pointer. On the other hand, there is *no address space*, and *no address decoding hardware* to move data to and from memory cells. An instruction has an unusual format, with 8 single bits and 4 two-way bonds. There is no fixed word size for data, there are no computed addresses, and there are no registers or indirection.

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

5.2 What can be done in the blob world?

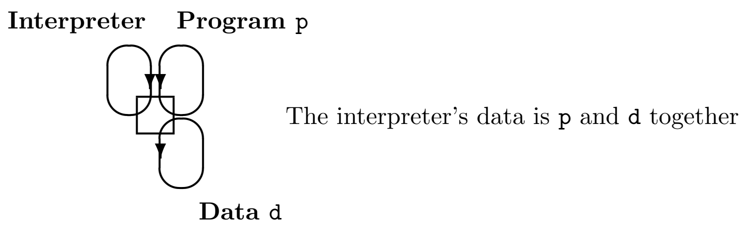
In principle the ideas presented and further directions are clearly expressible and testable in Maude or another tool for implementing term rewriting systems, or the kappa-calculus [7,9,12,14]. Current work involves programming a blob simulator. A prototype implementation has been made, with a functioning self-interpreter.

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. Appendix B shows how to generate blob code from a Turing machine, thus establishing Turing-completeness.

It seems possible to make an analogy between universality and self-reproduction that is tighter than seen in the von Neumann and other cellular automaton approaches. It should now be clear that familiar Computer Science concepts such as interpreters and compilers also make sense also at the biological level, and hold the promise of becoming useful operational and utilitarian tools.

5.3 Self-interpretation in the blob world

The figure of Section 4 becomes even more interesting when a program is executed interpretively, computing $[\![\texttt{interpreter}]\!](\texttt{p}, \texttt{d})$.



We have developed a "blob universal machine", i.e., a self-interpreter for the blob formalism that is closely analogous to Turing's original universal machine.

6 Contributions of This Work

We have for the first time investigated the possibility of programmable bio-level computation. The work sketched above, in particular the functioning of blob code, can all be naturally expressed in the form of abstract biochemical reaction rules. Further, we have shown molecular computation to be universal in a very strong sense: not only can every computable function be computed by a blob program, but this can all be done using *a single, fixed, set of reaction rules*: it is not necessary to resort to constructing new rule sets (in essence, new biochemical architectures) in order to solve new problems; it is enough to write new programs.

The new framework provides Turing-completeness efficiently and without asymptotic slowdowns. It seems possible to make a tighter analogy between universality and self-reproduction than by the von Neumann and other cellular automaton approaches.

It should be clear that familiar Computer Science concepts such as interpreters and compilers also make sense also at the biological level, and hold the promise of becoming useful operational and utilitarian tools.

References

1. L. M. Adleman. On constructing a molecular computer. In *DIMACS: series in Discrete Mathematics and Theoretical Computer Science*, pages 1–21. American Mathematical Society, 1996.
2. R. Backofen and P. Clote. Evolution as a computational engine. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic*, pages 35–55. Springer-Verlag, 1996.
3. D. Beaver. Computing with DNA. *Journal of Computational Biology*, 2(1):1–7, 1995.
4. Y. Benenson. Biocomputers: from test tubes to live cells. *Molecular BioSystems*, 5(7):675–685, 2009.
5. Y. Benenson, R. Adar, T. Paz-Elizur, Z. Livneh, and E. Shapiro. DNA molecule provides a computing machine with both data and fuel. In *Proc Natl Acad Sci U S A*, volume 100 of *Lecture Notes in Computer Science*, pages 2191–2196, 2003.
6. L. Cardelli and G. Zavattaro. On the computational power of biochemistry. In *AB '08: Proceedings of the 3rd international conference on Algebraic Biology*, pages 65–80, Berlin, Heidelberg, 2008. Springer-Verlag.
7. L. Cardelli and G. Zavattaro. Turing universality of the biochemical ground form. *Mathematical Structures in Computer Science*, 19, 2009.
8. P. Chapman. Life universal computer. *http://www.igblan.free-online.co.uk/igblan/ca/*, (November), 2002.
9. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
10. A. Danchin. Bacteria as computers making computers. *FEMS Microbiology Reviews*, 33(1):3 – 26, 2008.

11. V. Danos, J. Feret, W. Fontana, and J. Krivine. Abstract interpretation of cellular signalling networks. In *VMCAI*, volume 4905 of *VMCAI, Lecture Notes in Computer Science*, pages 83–97, 2008.
12. V. Danos and C. Laneve. Formal molecular biology. *Theor. Comp. Science*, 325:69 – 110, 2004.
13. P. Degano and R. Gorrieri, editors. *Computational Methods in Systems Biology, 7th International Conference, CMSB 2009, Bologna, Italy, August 31-September 1, 2009. Proceedings*, volume 5688 of *Lecture Notes in Computer Science*. Springer, 2009.
14. G. Delzanno, C. D. Giusto, M. Gabbrielli, C. Laneve, and G. Zavattaro. The *kappa*-lattice: Decidability boundaries for qualitative analysis in biological languages. In Degano and Gorrieri [13], pages 158–172.
15. M. Gardner. Mathematical recreations. *Scientific American*, October 1970.
16. M. L. Guerriero, D. Prandi, C. Priami, and P. Quaglia. Process calculi abstractions for biology. Technical report, University of Trento, Italy, Jan. 01, 2006.
17. M. Hagiya. Designing chemical and biological systems. *New Generation Comput.*, 26(3):295, 2008.
18. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science. Prentice Hall, 1993.
19. N. D. Jones. *Computability and complexity: from a programming perspective*. MIT Press, Cambridge, MA, USA, 1997.
20. L. Kari. Biological computation: How does nature compute? Technical report, University of Western Ontario, 2009.
21. L. Kari and G. Rozenberg. The many facets of natural computing. *Commun. ACM*, 51(10):72–83, 2008.
22. M. Minsky. *Computation: finite and infinite machines*. Prentice Hall, 1967.
23. C. L. Nehaniv. Asynchronous automata networks can emulate any synchronous automata network. *International Journal of Algebra and Computation*, 14(5-6):719–739, 2004.
24. T. Ran, S. Kaplan, and E. Shapiro. Molecular implementation of simple logic programs. *Nat Nano*, 4(10):642–648, Oct 2009.
25. E. Shapiro. Mechanical Turing machine: Blueprint for a biomolecular computer. Technical report, Weizmann Institute of Science, 1999.
26. E. Shapiro and Y. Benenson. Bringing DNA computers to life. *Scientific American*, 294:44–51, 2006.
27. C. Talcott. Pathway logic. In *Formal Methods for Computational Systems Biology*, volume 5016 of *LNCS*, pages 21–53. Springer, 2008. 8th International School on Formal Methods for the Design of Computer, Communication, and Software Systems.
28. A. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936-7.
29. J. von Neumann and A. W. Burks. *Theory of Self-Reproducing Automata*. Univ. Illinois Press, 1966.
30. E. Winfree. Toward molecular programming with DNA. *SIGOPS Oper. Syst. Rev.*, 42(2):1–1, 2008.
31. E. Winfree, X. Yang, and N. C. Seeman. Universal computation via self-assembly of DNA: Some theory and experiments. In *DNA Based Computers II, volume 44 of DIMACS*, pages 191–213. American Mathematical Society, 1996.
32. S. Wolfram. *A New Kind of Science*. Wolfram Media, January 2002.

33. P. Yin, A. J. Turberfield, and J. H. Reif. Design of an autonomous dna nanomechanical device capable of universal computation and universal translational motion. In *Tenth International Meeting on DNA Based Computers (DNA10)*, volume 3384 of *Lecture Notes in Computer Science*, pages 426–444, 2005.

A More on Turing completeness

How to show Turing completeness of a computation framework. This is typically shown by *reduction* from another problem already known to be Turing complete. Notation: let L and M denote languages (biological, programming, whatever), and let $[\![p]\!]^L$ denote the result of executing L-program p, for example an input-output function computed by p. Then we can say that language M is at least as powerful as L if

$$\forall p \in L\text{−programs } \exists q \in M\text{−programs } (\ [\![p]\!]^L = [\![q]\!]^M\ )$$

A popular choice is to let L be some very small Turing complete language, for instance Minsky register machines or two-counter machines (2CM). The next step is to let M be a biomolecular system of the sort being studied. The technical trick is to argue that, given any L-instance of (say) a 2CM program, it is possible to construct a biomolecular M-system that faithfully simulates the given 2CM.

Oddly enough, Turing completeness is not often used to show that certain problems *can* be solved by M-programs; but rather only to show that, say, the *equivalence* or *termination problems of M-programs are algorithmically undecidable* because they are undecidable for L, and the properties are preserved under the construction. This discussion brings up a central issue:

Simulation as opposed to interpretation. Arguments to show Turing completeness are (as just described) usually by *simulation*: for each problem instance (say a 2CM) one somehow constructs a biomolecular system such that ... (the system in some sense solves the problem). However, in many papers for each problem instance the construction of the simulator is done by hand, e.g., by the author writing the article. In effect the existential quantifier in $\forall p \exists q([\![p]\!]^L = [\![q]\!]^M)$ is computed by hand. This phenomenon is clearly visible in papers on cellular computation models: completeness is shown by simulation rather than by interpretation.

In contrast, Turing's original "Universal machine" simulates by means of *interpretation*: a stronger form of imitation, in which the existential quantifier is realised by machine. Turing's "Universal machine" is capable of executing an arbitrary Turing machine program, once that program has been written down on the universal machine's tape in the correct format, and its input data has been provided. Our research follows the same line, applied in a biological context: we show that simulation can be done by general interpretation, rather than by one-problem-at-a-time constructions.

Self-interpretation without asymptotic slowdown. The blob self-interpreter overcomes a limitation that seems built-in to the Turing model. Analysis of its running time reveals that the time taken to interpret one blob instruction is bounded by a constant that is *independent of the program being interpreted.* Intuitively, there are two reasons for this: First, there are no variables, pointers, or other features that add program-dependent amounts of time to the central self-interpretion loop. Second, the fact that every transfer of control in the interpreted program is to an adjacent program blob means that no program-dependent amount of time is spent on fetching the next instruction.

One consequence is that *constant time factors do matter*: the "linear hierarchy" results developed in [19] (Section 19.3 for the I language) also hold for the blob language. (The linear hierarchy result sounds intuitively obvious and natural, but in fact does not hold for many of the traditional models of computation, in particular does not hold for Turing machines with arbitrarily large alphabets or number of tapes.)

B Turing completeness of the blob model

We prove that any one-tape Turing machine with a single read/write head may be simulated by a blob program. The tape contents are always finite and enclosed between a left endmarker $\triangleleft$ and a right endmarker $\triangleright$.

B.1 Turing machine syntax

A Turing machine is a tuple $Z = (\{0,1\}, Q, \delta, q_{start}, q_{halt})$. The tape and input alphabet are $\{0,1\}$. (Blanks are not included, but may be encoded suitably by bits.) Q is a finite set of *control states* including distinct start and halting states $q_{start}, q_{halt} \in Q$. The *transition function* has type

$$\delta : \{0, 1, \triangleleft, \triangleright\} \times Q \to \mathcal{A} \times Q$$

where an *action* is any $A \in \mathcal{A} = \{L, R, W0, W1\}$. Notation: we write a Turing machine instruction as

$$\delta(q, b) \to (A, r)$$

meaning "In state q, reading bit b, perform action A and move to state r". Actions $L, R, W0, W1$ mean informally "move Left, move Right, Write 0, Write 1", respectively. For simplicity we assume that Turing machines may not both move and write on the tape in the same atomic step. (A "write-and-move" action may easily be implemented using two states and two steps.)

We also assume that every Turing machine satisfies the following *consistency assumptions*:

- If $\delta(q, \triangleleft) \to (A, r)$ is an instruction, then $A \in \{R\}$ (i.e. the machine never moves to the left of the left endmarker and cannot overwrite the endmarker).
- If $\delta(q, \triangleright) \to (A, r)$ then $A \in \{L, W0, W1\}$ (i.e. the machine never moves to the right of the right endmarker, but *can* overwrite the endmarker).

Definition 1. *Let M be a Turing machine. The* state graph *of M is the directed graph where the nodes are the states of M and there is a directed edge from q to r annotated (b, A) if there is an instruction $\delta(q, b) \to (A, r)$.*

B.2 Turing machine semantics

A total state has the form

$$\triangleleft\ b_1 \dots \overset{q}{b_i} \dots b_n\ \triangleright$$

where the b_j are tape symbols, and q is a control state. We define the *tape contents* of the machine to be everything enclosed between $\triangleleft$ and $\triangleright$.

The Turing machine defines a one-step transition relation between total states in the expected way (not spelled out here). Tapes may only grow to the right, not the left. We assume that if there is an instruction of the form $\delta(q, \triangleright) \to (W0, r)$ or $\delta(q, \triangleright) \to (W1, r)$ (i.e. the right endmarker is overwritten), then the tape is automatically extended to the right with a new endmarker to the immediate right of the previous endmarker.

Remark: the tape contents will always be finite after a finite number of computation steps.

Input/Output: A Turing machine Z computes a partial function

$$[\![Z]\!] : \{0,1\}^* \rightharpoonup \{0,1\}^*$$

- *Input*: The machine is in its start state with the tape head on the tape cell to the *immediate right* of the left endmarker $\triangleleft$. The input is the contents of the tape.
- *Output*: The machine is in its halt state. The output is the contents of the tape.

B.3 Compiling a Turing machine into a blob program

We describe a way to compile any Turing machine $Z = (\{0,1\}, Q, \delta, q_{start}, q_{halt})$ into blob program code *code*(Z) that simulates it. Compilation of a Turing machine into blob code is as follows:

- Generate blob code for each instruction $\delta(q, b) \to (A, r)$.
- Collect blob code for all the states into a single blob program.

Before describing the compilation algorithm, we explain how the blob code realises a step-by-step simulation of the Turing machine Z.

Turing machine representation by blobs At any time t in its computation, the Turing machine's tape $b_1 \dots b_i \dots b_n$ will represented by a finite sequence $B_1 \dots B_i \dots B_n$ of blobs. If at time t the Turing machine head is scanning tape symbol b_i, the active data blob will be the blob B_i. Arrangement: each B_i is linked to its predecessor via bond site 1, and to its successor via bond site 2.

The Turing machine's control state will correspond to the active program blob in *code*(Z).

The cargo bits of the "data blobs" are used to indicate the contents of the the tape cell:

- Cargo bit 0 is unused in the simulation.
- Cargo bit 1 is used to hold the bit occupied by the tape cell (if the blob represents either $\lhd$ or $\rhd$, the contents of cargo bit 1 is irrelevant).
- Cargo bit 2 is '1' iff the blob represents the *left* endmarker $\lhd$.
- Cargo bit 3 is '1' iff the blob represents the *right* endmarker $\rhd$.

Syntax of the generated code We will write the generated blob target program as straightline code with labels. For every instruction, the "next" blob code instruction to be executed is the one linked to the active program blob by the latter's "successor" bond site 2. Thus, in

```
SCG 0 5
EXT
```

the blob corresponding to `SCG 0 5` has its bond site 2 linked to the "predecessor" bond site 1 of the blob corresponding to `EXT`.

Code generation for each state Let $q \neq q_{halt}$ be a state. The four possible kinds of transitions on state q are:

$$\begin{array}{c}\delta(q,0) \to (A0,q0)\\ \delta(q,1) \to (A1,q1)\\ \delta(q,\lhd) \to (AL,qL)\\ \delta(q,\rhd) \to (AR,qR)\end{array}$$

where $q0,q1,qL,qR \in Q$, $A0,A1 \in \{L,R,W0,W1\}$, and $AL,AR \in \{L,W0,W1\}$.

We generate code for q as follows. For typographical reasons, $\lhd$ = EL and $\rhd$ = ER. The action code notations `[A0]` etc, is explained below, as is the label notation `<label>`. The initial `FIN` code may be safely ignored on the first reading.

```
Generate i-1 FIN  // Assume program port 2 is always "next" operation
                  // Each FIN is labeled as noted below
                  // The last FIN is bound (on its bond site 2) to
                  // the blob labeled 'Q' below.

Q:   JCG  2  QLE  // If 1, We're at left tape end
                  // By convention, bond site 3 of the APB is
                  // bound to the blob labeled QLE
     JCG  3  QRE  // If 1, We're at right tape end
     JCG  1  Q1   // We're not at any end. If '0' is scanned, move along
                  // (on bond site 2),
                  // otherwise a '1' is scanned, jump to Q1
```

```
                // (on bond site 3)
     [A0]       // Insert code for action A0
     FIN qA0q0  // Go to appropriate fanin before q0 (on bond site 2)

Q1:  [A1]       // Insert code for action A1
     FIN qA1q1  // Go to appropriate fanin before q1 (on bond site 2)

QLE: [AL]       // Insert code for AL
     FIN qELALqL // Go to appropriate fanin before qL (on bond site 2)

QRE: R[AR]      // Insert code for AR (with the R[ ]-function)
     FIN ERARqR // Go to appropriate fanin before qR (on bond site 2)

                // Code for q end
```

Code for q_{halt}:

```
Generate i-1 FIN  // Assume program port 2 is "next" operation always
                  // Each FIN is labeled as noted below
                  // The last FIN is bound (on its bond site 2) to
                  // the blob labeled 'Qh' below.

Qh: EXT
```

The JCG instructions test the data blob B_i to see which of the four possible kinds of transitions should be applied. Codes [A0], [A1], [AL], R[AR] simulate the effect of the transition, and the FIN after each in effect does a "go to" to the blob code for the Turing machine's next state. (This is made trickier by the fan-in restrictions, see Section B.3 below.)

Two auxiliary functions We use two auxiliary functions to generate code:

$$\texttt{[]} : \{L, R, W0, W1\} \longrightarrow \text{blobcode}$$

and

$$\texttt{R[]} : \{L, W0, W1\} \longrightarrow \text{blobcode}$$

Function [] is used for code generation on arbitrary tape cells, and R[] for code generation when the Turing machine head is on the right end marker where some housekeeping chores must be performed due to tape extension.

Code generation for instructions not affecting the right end of the tape

```
                        [W0]

SCG 0 1       // Set tape cell content to 0


                        [W1]
```

```
SCG 1 1          // Set tape cell content to 1
```

$[L]$

```
CHD  1           // Set ADB to previous blob (move tape left)
```

[R]

```
CHD  2           // Set ADB to next blob (move tape right)
```

Code generation for instructions that can extend the tape

R[W0]

```
 SCG  0 3        // Current blob is no longer at right tape end
 INS  2 1        // Insert new blob at bond port 2 on ADB
                 // (new tape cell). New blob is bound at site 1.
 CHD  2          // Change ADB to new blob      (move head right)
 SCG  1 3        // New blob is at the right end of the tape
 CHD  1          // Change ADB to original blob (move head left)
 SCG  0 1        // Write a ’0’ in the tape cell (as per W0).
```

R[W1]

```
 SCG  0 3        // Current blob is no longer at right tape end
 INS  2 1        // Insert new blob at bond port 2 on ADB
                 // (new tape cell). New blob is bound at site 1
 CHD  2          // Change ADB to new blob      (move head right)
 SCG  1 3        // New blob is right tape end
 CHD  1          // Change ADB to original blob (move head left)
 SCG  1 1        // Write a ’1’ in the tape cell (as per W1)
```

R[L]

```
R[L] = [L]       // Move to the left
        // TM does not move right at right tape end.
```

Control flow in the generated blob code A technical problem in code generation. We now explain the meaning of the somewhat cryptical comments such as "`Go to appropriate fanin before q1`" in Section B.3, and notations such as `qA0q0`.

The problem: while a pointer-oriented language allows an unbounded number of pointers into the same memory cell, this is not true for the blob structures (the reason is that a bond is intended to model a chemical connection between two molecules). This is a "fan-in" restriction on program (and data) syntax.

A consequence: blob program code may not contain more than one control transfer to a given instruction, unless this is done by a bond site different from the usual "predecessor" site 1. The purpose of the instruction `FIN` is to allow *two* entry points: one as usual by bond site 1, and a second by bond site 3.

The initial `FIN` code generated of Section B.3. This concerns the entry points into blob code for a Turing state q. Let i be the number of directed edges to q in the state graph (i.e., the number of "go to's" to q).

If $i \leq 1$, we generate no fanin blobs.

Otherwise, we generate $i-1$ fanin blobs before the code generated for q; these handle the i transitions to q. The blobs bound to the fanin nodes occur in the code generated for other states (perhaps from q to itself). For each transition $\delta(q', b) \to (A, q)$, a blob in the code generated for q' is bound to a single fanin blob for q. The fanin blob generated above, before the generated code for state q, is labeled by `q'bAq`.

C Dimensionality limitations

Limitation to three dimensions. The physical world imposes a dimensionality requirement we have not yet addressed: data and program code cannot be packed with a density greater than that allowed by three-dimensional Euclidean space. The idea of a biologically plausible computing model that must work in 3-space provokes several interesting questions.

Realisability in 3-space: In the blob model, following a chain of k bonds from the active data blob (at any time in a computation) should give access to at most $O(k^3)$ blobs. This is not guaranteed by the blob model as presented above; indeed, a blob program could build a complete 3-ary tree of depth k and containing 3^k blobs at distance k. This structure could not be represented in 3-space with our restrictions, and still have the intended semantic structure: that any two blobs linked by a bond should be adjacent in the biological "soup".

On dimensional limits in other computation models. The usual Turing machine has a fixed number of 1-dimensional tapes (though k-dimensional versions exist, for fixed k). Cellular automata as in [29,8,32] have a fixed 2-dimensional architecture. Dimensionality questions are not relevant to Minsky-style machines with a fixed number of registers, e.g., the two-counter machine.

Machines that allow computed addresses and indirection, e.g., the RAM, RASP, etc., have no dimensionality limitations at all, just as in the "raw" blob model: traversing a chain of k bonds from one memory can give access to a number of cells exponential in k (or higher if indexing is allowed).

3D complexity classes? The well-known and well-developed Turing-based computational complexity theory starts by restricted programs' running time and/or space. An possible analogy would be to limit the dimensionality of the data structures that a program may build during a computation.

Pursuing the analogy, the much-studied complexity class PTIME is quite large, indeed so large that dimensionality makes no difference: on any traditional model where data dimensionality makes sense, it would be an easy exercise to show that PTIME = PTIME3D. What if instead we study the class LINTIME of problems solvable in linear time (as a function of input size)? Alas, this smaller, realistically motivated class is not very robust for Turing machines, as small differences in Turing models can give different versions of LINTIME (Sections 18, 19, 25.6 in [19]). It seems likely though that the LINTIME class for blob machines is considerably more robust.

Conjecture: LINTIME3D $\subsetneq$ LINTIME on the blob model.

Another interesting question: does self-interpretation cause a need for higher dimensionality? We conjecture that this is not so for any one fixed interpreted program; but that diagonalisation constructions can force the necessary dimensionality to increase. This appears to be an excellent direction for future work.