PROGRAMMING IN BIOMOLECULAR COMPUTATION

Lars Hartmann

Neil D. Jones

Jakob Grue Simonsen

+

Visualization by Søren Bjerregaard Vrist

(All now or recently at the University of Copenhagen)

Conference: META 2010 (July 1, 2010)

Source: June 2010 conference CS2BIO Computer Science to Biology

UNIVERSALITY AND PROGRAMMING IN A BIOCHEMICAL SETTING

Turing completeness results for biomolecular computation:

- **Cardelli, Chapman, Danos, Reif, Shapiro, Wolfram,. . .**
- **Net effect: any computable function can be computed, in some sense, by various biological mechanisms.**
- **Not completely compelling from a programming perspective.**
- **Our aim: a computation model where**
  - **"program" is clearly visible and natural, and**
  - **Turing completeness is not artificial or accidental, but a natural part of biomolecular computation**

CONNECTIONS EXIST BETWEEN BIOLOGY AND COMPUTATION, but . . .

WHERE ARE THE PROGRAMS?

Our proposal: a model of computation that is

- **biologically plausible**: semantics by chemical-like reaction rules;
- **programmable** (a bit like low-level computer machine code);
- **uniform**: new "hardware" not needed to solve new problems;
- **stored-program**: programs = data;

  programs are **executable** and **compilable** and **interpretable**
- **universal**: all computable functions can be computed
- **Turing complete** in a strong sense: $\exists$ a universal algorithm

  (able to execute any program, asymptotically efficient)

BUT WHERE ARE THE PROGRAMS?

In existing models of biomolecular computation

it's hard to see anything like a program that realises or directs a computational process.

- In cellular automata, "program" is expressed only in the initial cell configuration, or in the global transition function
- Many examples: given a problem, authors cleverly devise a biomolecular system that can solve this particular problem
- The algorithm being implemented is hidden in the details of the system's construction, hard to see.

Our purpose is to fill this gap,

- to establish a biologically feasible framework in which
- programs are first-class citizens.

OTHER COMPUTATIONAL FRAMEWORKS

Circuits, BDDs, finite automata: Nonuniform, Turing incomplete

Turing machine:

- **Pro** Visible program; complete; universal machine exists
- **Con** Asymptotically slow: universal machine takes time $O(n^2)$ to simulate a program running in time $O(n)$

Other program-based models: Post, Minsky, LISP, RAM, RASP...

Complex, biologically implausible

Cellular automata: von Neumann, LIFE, Wolfram,...

- **Pro** Can simulate a Turing machine
- **Con** Complex, biologically implausible (synchronisation!)

  There is no natural universal cellular automaton.

  It's very hard to see "the program".

"DIRECT" PROGRAM EXECUTION

Write $[\![\texttt{program}]\!]$ for the meaning or net effect of running $\texttt{program}$:

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

- $\texttt{program}$ **is an active agent.**
- **It is activated (run) by applying the semantic function $[\![\_]\!]$.**
- **Some mechanism is needed to execute $\texttt{program}$, i.e., to apply $[\![]\!]$ to $\texttt{program}$ and $\texttt{data}_{in}$:**

  hardware ("wetware"?).

THE BIOLOGICAL WORLD IS NOT HARDWARE!

We must re-examine programming language assumptions.
Computers have programmer-friendly conveniences, e.g.,

- A large address space of randomly accessible data
- Pointers to data, perhaps at a great "distance" from the current program or data
- address arithmetic, index registers,...
- Unbounded fan-in: many pointers to the same data item...

None of these is biologically plausible!

Workarounds are needed

if we want to do biological programming.

FOR BIOLOGICAL PLAUSIBILITY

- There is no action at a distance: all effects achieved via chains of local interactions. Biological analog: signaling.
- There are no pointers to data (addresses, links, list pointers): To be acted on, a data value must be physically adjacent to an actuator. Biological analog: chemical bond between program and data.
- No nonlocal control transfer, e.g., unbounded GOTOs or remote procedure calls. Biological analog: a bond from one part of a program to another.
- A "yes": $\exists$ available resources to tap, i.e., energy to change the program control point, or to add data bonds. Biological analogs: ATP, oxygen, Brownian movement.

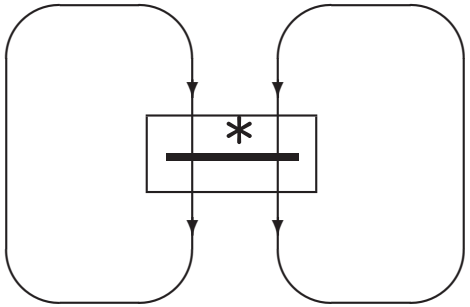
KEEPING THE FOCUS

How to structure a biologically feasible model of computation?

- **Idea: keep current program counter and data cursor always close to a focus point where all actions occur.**
- **How? Continually shift both program and data, to keep the active bits near the focus.**

Running program p: computing [[p]](d)

Program p **Data** d



□ = **Focus point for control and data**
(connects the APB and the ADB)

____ = **program-to-data bond: "the bug"**

A MOVIE IS WORTH DURATION$\times$FRAMERATE$\times$1000 WORDS

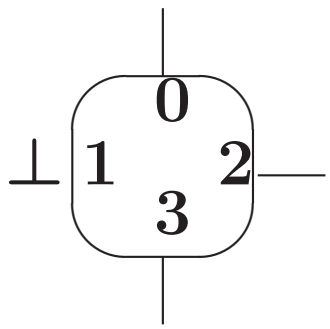
(largedataplay2.avi)

THE BLOB MODEL

Simplified view of a molecule and chemical interactions (Cardelli, Danos, Lanève,. . .).

Blobs are in a biological "soup" and are connected by symmetrical bonds linking their bond sites.

Picture of a blob:



4 bond sites and 8 cargo bits

Bond sites $0, 2$ and 3 are bound, and 1 is unbound

PROGRAM BLOBS AND DATA BLOBS

- A program p is (by definition) a connected assembly of blobs.

- A data value d is (also) a connected assembly of blobs.

At any moment during execution, i.e., computation of $[\![p]\!](d)$:

- The active program blob (APB) is in p.
- The active data blob (ADB) is in d.
- There is a bond $*$ ("the bug") between the APB and the ADB, at bond sites 0.

BLOB STRUCTURE (AS DATA OR AS PROGRAM)

A blob has 4 bond sites and 8 cargo bits (boolean values).

- A bond site can be: bound to another blob; or $\perp$ (unbound).
- 8 cargo bits of local storage.
- When used as program:
  - the activation cargo bit $= 1$.
  - the other 7 cargo bits contain an instruction
- When used as data:
  - the activation cargo bit $= 0$;
  - the other 7 cargo bits (and 4 bonds): no constraints.

ABOUT INSTRUCTIONS:

Instruction form:

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

Why exactly 4 bonds?

- **Predecessor (1 bond); true and false successors (2 bonds);**
- **plus one bond to link the APB to the ADB.**

It's almost a von Neumann machine code, but...

- **A bond is a two-way link between two adjacent blobs.**
- **A bond is not an address.**
- **There is no address space as in conventional computer (and hence: no address decoding hardware).**
- **Also: no registers (use the cargo bits instead).**

INSTRUCTIONS HAVE 8 BITS

Instruction	Description	Informal semantics (write :=: for 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 = $\perp$ then APB := APB.3 else APB := APB.2
`CHD b`	CHange Data	ADB := ADB.b; APB := APB.2
`INS b1 b2`	INSert new bond	ADB-new.b2 :=: ADB.b1; ADB-new.b1 :=: ADB.b1.bs;
	—	APB := APB.2
`SBS b1 b2`	SWap Bond Sites	ADB.b1 :=: ADB.b2; APB := APB.2
`SWL b1 b2`	SWap Links	ADB.b1 :=: ADB.b2.b1; APB := APB.2
`SWP3 b1 b2`	Swap bs3 on linked	ADB.b1.3 :=: ADB.b2.3; APB := APB.2
`FIN`	Fan IN	APB := APB.2 (two predecessors: bond sites 1 and 3)
`EXT`	EXiT program	

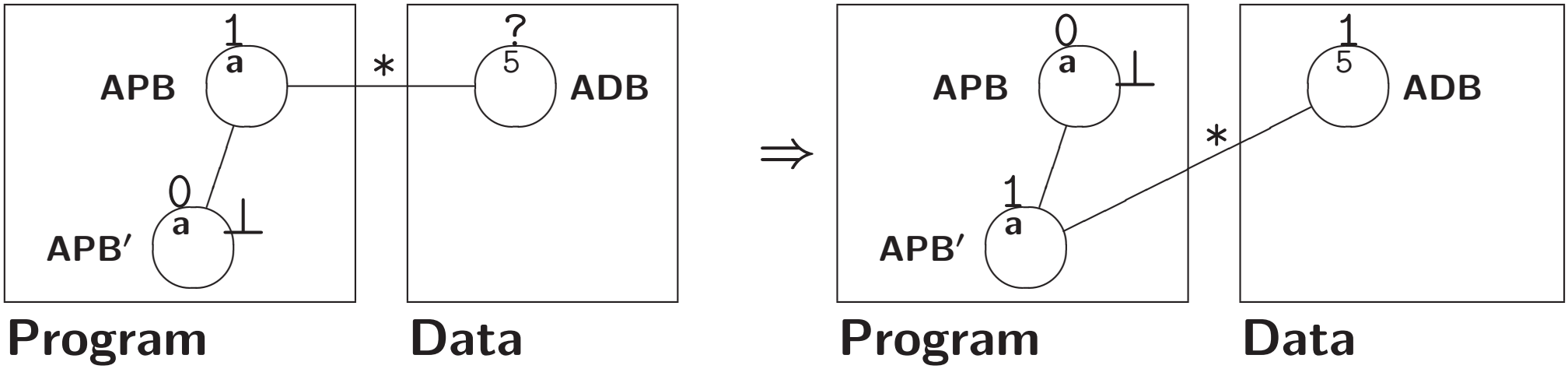
`SCG,...,EXT`: **Operation codes**

`b, b1, b2`: **Bond site numbers**

`c`: **Cargo site number**

`v`: **A one-bit value**

EXAMPLE: EFFECT OF `SCG 1 5` (SET CARGO BIT 5 TO 1)



- **"The bug" $\overset{*}{\text{—}}$ has moved:**
  - **before execution, it connected APB with ADB.**
  - **After: it connects successor APB′ with ADB.**
- **Also: activation bits 0, 1 have been swapped.**

Instruction syntax: **the 8-bit string 11001101 is grouped as**

$$\overbrace{1}^{a}\ \overbrace{100}^{SCG}\ \overbrace{1}^{v}\ \overbrace{101}^{c}$$

SEMANTICS OF `SCG 1 5` BY "SOMETHING LIKE" A CHEMICAL REACTION RULE

Instruction form: $\overbrace{1}^{a}\ \overbrace{100}^{SCG}\ \overbrace{1}^{v}\ \overbrace{101}^{c}$

$$\overbrace{B[1\ 100\ 1\ 101](*\text{---})}^{APB},\ \overbrace{B[0\text{-------}](\perp\text{---})}^{APB'},\ \overbrace{B[0\text{----}x\text{--}](*\text{---})}^{ADB}$$

$$\Rightarrow$$

$$\underbrace{B[0\ 100\ 1\ 101](\perp\text{---})}_{APB},\ \underbrace{B[1\text{-------}](*\text{---})}_{APB'},\ \underbrace{B[0\text{----}1\text{--}](*\text{---})}_{ADB}$$

(- = unchanged bond or cargo bit)

Similar style to: Danos and Laneve, Formal Molecular Biology.

A FURTHER EXAMPLE: APPENDING TWO LISTS

(Example film)

A WAY TO SHOW TURING COMPLETENESS

Language M is as powerful as L (write $L \leq M$) if

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

L and M are languages (biological, programming, whatever).

Aim: show that an interesting M is Turing complete.

One way: reduce an already Turing complete language , e.g.,

- **$L =$ two-counter machines 2CM.**
- **$M =$ a biomolecular system of the sort being studied.**
- **The technical trick: show how to construct**
  - **from any 2CM program,**
  - **a biomolecular M-system that simulates the given 2CM.**

ANOTHER WAY: SIMULATION BY INTERPRETATION

Turing completeness is usually shown by simulation, e.,g.,

- for any 2CM program you build a biomolecular system such that ...

But: the biomolecular system is usually built by hand. The effect: hand computation of the $\exists$ quantifier in

$$\forall p \exists q ([\![p]\!]^L = [\![q]\!]^M)$$

In contrast, Turing's original "Universal machine" (UM) works by interpretation, where $\exists$ is realised by machine.

- The UM can execute any TM program, if coded on the UM's tape along with its input data.
- Our research follows Turing's line, in a biological context:

  It does simulation by general interpretation, and not by one-problem-at-a-time constructions.

ANOTHER WAY: SIMULATION BY INTERPRETATION

Turing completeness is usually shown by simulation, e.,g.,

- for any 2CM program you build a biomolecular system such that ...

But: the biomolecular system is usually built by hand. The effect: hand computation of the $\exists$ quantifier in

$$\forall p \exists q ([\![p]\!]^L = [\![q]\!]^M)$$

In contrast, Turing's original "Universal machine" (UM) works by interpretation, where $\exists$ is realised by machine.

- The UM can execute any TM program, if coded on the UM's tape along with its input data.
- Our research follows Turing's line, in a biological context:

  It does simulation by general interpretation, and not by one-problem-at-a-time constructions.

PROGRAM EXECUTION BY INTERPRETATION

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

- **Now** `program` **is a passive data object: both** `program` **and** $\texttt{data}_{in}$ **are data for the interpreter.**

- `program` **is now executed by running the interpreter program.**

  (Of course, some mechanism will be needed to run the interpreter, e.g., hard-, soft- or wetware.)

- **Self-interpretation is possible, and useful in practice.**

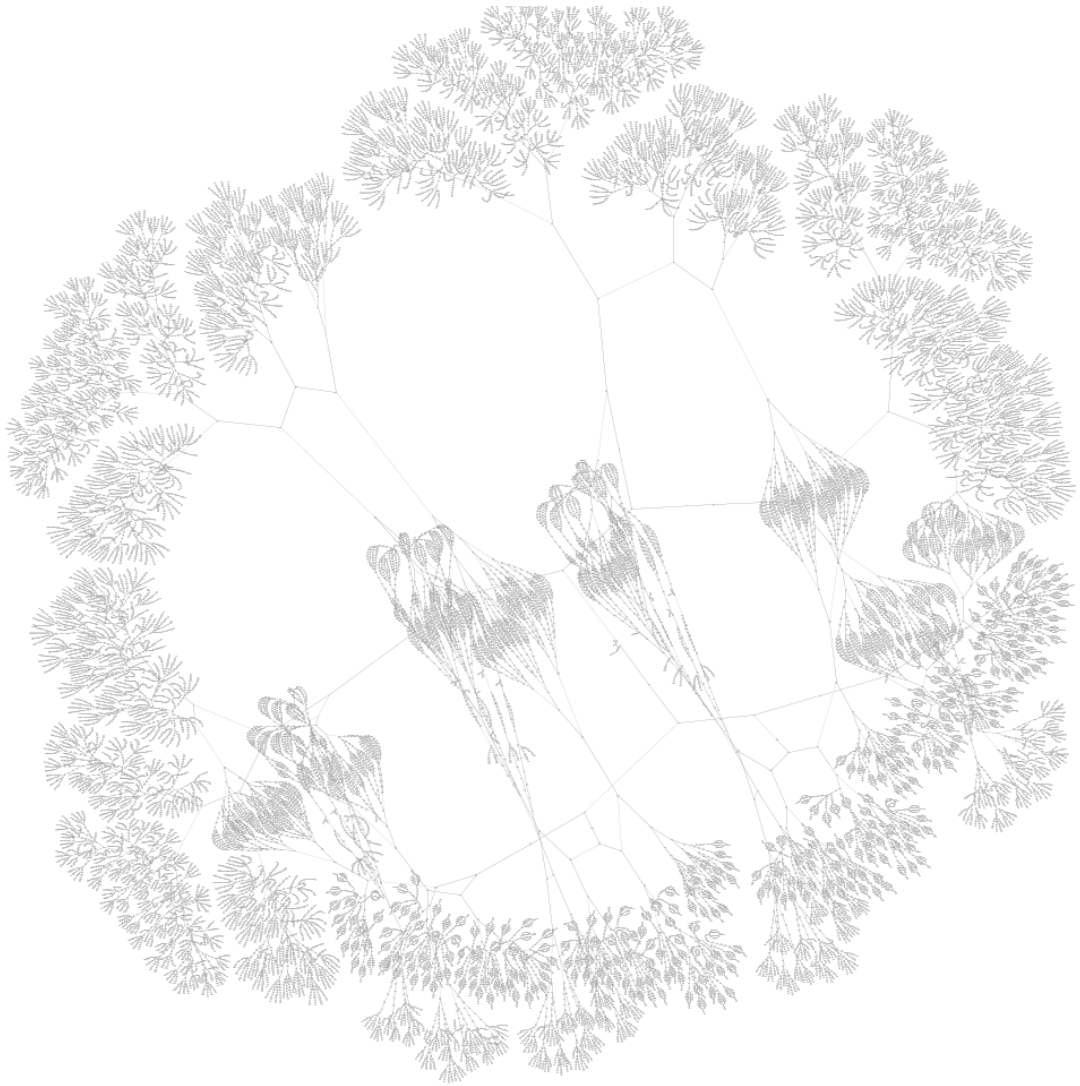
- **The Universal Turing machine is a self-interpreter.**

A "BLOB UNIVERSAL MACHINE"

We have developed a self-interpreter for the blob formalism – analogous to Turing's original universal machine.

This gives: Turing-completeness in a new biological framework.

BIRDS-EYE VIEW OF THE SELF-INTERPRETER



(Not shown: Each 'finger' along the periphery has a connection to the main control in the center)

CONTRIBUTIONS OF THIS WORK

- Programmable bio-level computation where programs = data.
- Blob semantics by abstract biochemical reaction rules.
- All computable functions are blob-computable:
  - Can do with one fixed, set of reaction rules (defining a fixed instruction set, i.e., a "machine language")
  - Don't need new rule sets (i.e., biochemical architectures) to solve new problems; it's enough to write new programs.
- (Uniform) Turing-completeness
- Promise of tighter analogy between universality and self-reproduction.
- Interpreters and compilers make sense at biological level, may give useful operational and utilitarian tools.

WHERE TO NOW?

Some points to address:

- Find a true, biological (not just "feasible") implementation of the fixed set of reduction rules in vitro.
- Programs are currently similar to classical machine code; this requires programmer skill. Solution: Devise an intermediate-level blob programming language.
- Still to analyse: The time or energy cost of performing a single program step (may depend on program/data). An appropriate and realistic cost model should be found.
- Bonus: This could initiate a study of computational complexity in the blob world.

THANK YOU!

Questions?