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)

National Institute of Informatics (11. October 2011)

Sources:
► Conference CS2BIO Computer Science to Biology (LNCS proceedings June 2010)
► Festschrift for Carolyn Talcott (to appear November 2011)
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

▶ **biochemically 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: ∃ a universal algorithm

    (able to execute any program, asymptotically efficient)
Does it make sense to have program execution in a biological context?

Evidence for “yes”: program-like behavior, e.g.,
▶ genes that direct protein fabrication, “switching on” and “switching off”, reproduction,

Many analogies to the world of programs (though not yet well understood). This work’s viewpoint is
▶ synthetic: concerned with building things, as in engineering and computer sciences
▶ in contrast to the analytic viewpoint common to the natural sciences, studying how nature really works.

Goal: Can program execution happen in a biological context?
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.
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”.
PROGRAM EXECUTION IN GENERAL

The authors’ starting point: programming languages, compilers, computability and complexity theory (no biology!)

Natural question: “can” program execution take place?

What is a program? Roughly . . .

▶ A set of instructions
▶ that specify a series (or set) of actions
▶ Actions are carried out when the instructions are executed (activated . . . )

In stored-program computation models (e.g., von Neumann)
▶ A program is a concrete object (a form of data)
▶ that can be replaced to specify different actions.

Thus the program is software and not hardware
“DIRECT” PROGRAM EXECUTION

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

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

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

The task of programming is, given a desired semantic meaning, to devise a program that computes it.
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.
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.

There is no nonlocal control transfer, e.g., unbounded GO-TOs or remote procedure calls. Biological analog: a bond from one part of a program to another.

A “yes” ∃ available resources to tap, i.e., energy to change the program control point, or to add data bonds. Biological analogs: ATP, oxygen, Brownian movement.
How to structure a biologically feasible model of computation?

▸ Idea: keep current **program cursor** 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 $\llbracket p \rrbracket(d)$

**Program** $p$  **Data** $d$

\[
\begin{array}{c}
\text{Focus point for control and data} \\
\text{(connects the APB and the ADB)}
\end{array}
\]

\[
\begin{array}{c}
\text{program-to-data bond: “the bug”}
\end{array}
\]
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: (Bond sites 0, 2 and 3 are bound, and 1 is unbound)

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

Here: Bond sites 0, 2 and 3 are bound, and 1 is unbound.
(Cargo bits not shown)
A MOVIE IS WORTH DURATION $\times$ FRAMERATE $\times$ 1000 WORDS

(Circle.avi)
A program $p$ is (by definition) a connected assembly of blobs.

The data apace 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 $\ast$ ("the bug") between the APB and the ADB, at bond sites $0$. 
A blob has **4 bond sites** and **8 cargo bits** (boolean values).

- A bond site can be:
  - bound to **exactly one** other blob; or
  - be ⊥ (unbound).

- A blob has **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.
WHAT HAPPENS AT THE PROGRAM-TO-DATA BOND?

Program p  Data d

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

* = program-to-data bond

An instruction can . . .

- **Move** the data cursor along bond 1 (or bond 2 or 3)
- **Branch**: is data cursor’s bond 1 empty or not? (or 2 or 3)
- **Branch**: is data cursor’s cargo bit $i = 1$ or 0? ($i = 1, 2, \ldots, 7$)
- **Insert** a new blob at bond 1 (or 2 or 3)
- **Swap**: interchange some bonds
- **Fan-in**: merge control from two predecessor instructions
Instruction form: (a blob read as an instruction)

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 program cursor and the data cursor.

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Informal semantics (write <code>:=:</code> for a two-way interchange)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCG v c</td>
<td>Set CarGo bit</td>
<td>ADB.c := v; APB := APB.2</td>
</tr>
<tr>
<td>JCG c</td>
<td>Jump CarGo bit</td>
<td>if ADB.c = 0 then APB := APB.3 else APB := APB.2</td>
</tr>
<tr>
<td>JB b</td>
<td>Jump Bond</td>
<td>if ADB.b = ⊥ then APB := APB.3 else APB := APB.2</td>
</tr>
<tr>
<td>CHD b</td>
<td>CHange Data</td>
<td>ADB := ADB.b; APB := APB.2</td>
</tr>
<tr>
<td>INS b1 b2</td>
<td>INSert new bond</td>
<td>ADB-new.b2 :=: ADB.b1; ADB-new.b1 :=: ADB.b1.bs; APB := APB.2</td>
</tr>
<tr>
<td>SBS b1 b2</td>
<td>SWap Bond Sites</td>
<td>ADB.b1 :=: ADB.b2; APB := APB.2</td>
</tr>
<tr>
<td>SWL b1 b2</td>
<td>SWap Links</td>
<td>ADB.b1 :=: ADB.b2.b1; APB := APB.2</td>
</tr>
<tr>
<td>SWP3 b1 b2</td>
<td>Swap bs3 on linked</td>
<td>ADB.b1.3 :=: ADB.b2.3; APB := APB.2</td>
</tr>
<tr>
<td>FIN</td>
<td>Fan IN</td>
<td>APB := APB.2 (two predecessors: bond sites 1 and 3)</td>
</tr>
<tr>
<td>EXT</td>
<td>EXiT program</td>
<td></td>
</tr>
</tbody>
</table>

**SCG,...,EXT:** Operation codes  
**b, b1, b2:** Bond site numbers  
**c:** Cargo site number  
**v:** A one-bit value
EXAMPLE: EFFECT OF \texttt{SCG 1 5} (SET CARGO BIT 5 TO 1)

- “The bug” — 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

\[
\begin{array}{cccc}
a & SCG & v & c \\
1 & 100 & 1 & 101 \\
\end{array}
\]
SEMANTICS OF SCG 1 5 BY "SOMETHING LIKE" A CHEMICAL REACTION RULE

Instruction form:

\[
\begin{align*}
&\text{APB} \quad \text{SCG} \quad \text{APB'} \quad \text{ADB} \\
B[1 \ 100 \ 1 \ 101](\ast - - -), &\quad B[0 - - - - - - - - ](\perp - - -), &\quad B[0 - - - x - - - ](\ast - - -) \\
\Rightarrow \\
B[0 \ 100 \ 1 \ 101](\perp - - -), &\quad B[1 - - - - - - - - ](\ast - - -), &\quad B[0 - - - 1 - - - ](\ast - - -) \\
\end{align*}
\]

(− = unchanged bond or cargo bit)

KAPPA model: Danos and Laneve, Formal Molecular Biology.
A FURTHER EXAMPLE: APPENDING TWO LISTS

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

$$\forall p \in L - \text{programs} \ \exists q \in M - \text{programs} \ (\left[p\right]^L = \left[q\right]^M)$$

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

▶ for any 2CM program you build a biomolecular system . . .

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

▶

\[[\text{interpreter}] (\text{program, data}_{in}) = \text{data}_{out}\]

▶ **Now** program is a **passive data object**: both program and 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.

▶ **Turing’s original “Universal machine”** was a self-interpreter.
We have programmed a self-interpreter for the blob formalism – analogous to Turing’s original universal machine. This gives: Turing-completeness in a new biological framework.
The interpreted program $p$ and its data $d$ are both data for interpreter.
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.

**Self-interpretation without asymptotic slowdown.** The blob data model (4 bond sites per bob) gives more efficient self-interpretation than Turing’s original universal machine. Overcomes a limitation built-in to the Turing model, namely asymptotic slowdown. The technical reason:

The time to interpret one blob instruction is bounded by a constant $c$

(that may depend on the program being interpreted)
(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:
  - This can be done with **one fixed, set of reaction rules** (defining a fixed instruction set, i.e., a “machine language”)
  - We don’t need **new rule sets** (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?

► Find a **true, biological** (not just “plausible”) implementation of the fixed set of reduction rules in vitro.

► Programs are currently similar to classical **machine code**; this requires (too much) programmer skill. Possible solutions:
  
  ► **Devise an intermediate-level blob programming language.**

  ► Describe/constrain program behavior and data structures by
    
    • **Static program analysis** (to describe); or
    
    • A **type system** (to constrain)

  ► **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.

  ► **Computational complexity,** e.g., **dimensionality** limitations.
References

THANK YOU!

Questions?