PROGRAMS = DATA = FIRST-CLASS CITIZENS IN A COMPUTATIONAL WORLD

Lars Hartmann
Neil D. Jones
Jakob Grue Simonsen
+ Visualization by Søren Bjerregaard Vrist

(All now or recently at the University of Copenhagen)

IMDEA and UCM Madrid (March 2012)

Sources:
- Conference CS2BIO Computer Science to Biology (ENTCS proceedings June 2010)
- Festschrift for Carolyn Talcott (November 2011, Springer Festschrift Series, LNCS vol. 7000)
- Article accepted to appear in Philosophical Transactions A of the Royal Society
1. A convincing analysis of the nature of computation
2. A very early model of computation (MOC)
3. The first programmers’ manual
4. Undecidability of the halting problem
5. Universal Turing machine (a self-interpreter)
6. Contributor to the “Confluence of ideas”: that 
   all sensible models of computation are equivalent, e.g.,
   ▶ Turing machine
   ▶ Lambda calculus
   ▶ Recursive function definitions
   ▶ String rewrite systems
75 YEARS OF MODELS OF COMPUTATION (just a few)

- Lambda calculus  
  Church 1936
- Turing machine  
  1936
- von Neumann architecture  
  1945
- Finite automata  
  Rabin and Scott
- Counter machine  
  Lambek and Minsky
- Random access machine (RAM)  
  Cook and Reckhow
- Random access stored program (RASP)  
  Elgot and Robinson
- Cellular automaton, LIFE,..  
  von Neumann, Conway, Wolfram
- Abstract state machine  
  Gurevich et al
- Text register machine  
  Moss
- Blob model  
  2010
Criteria

▶ How to compare
▶ How to improve
▶ Lacks, failings, inconveniences
▶ What are they suitable for?


New direction: biological computing

▶ Enormous potential (price, concurrency, automation, . . .)
▶ Many as-yet-unclear concepts
▶ Modeling versus(?) programming
Some dimensions of MOCS

- “Reasonable” machines? (van Emde Boas, Ugo dal Lago)
  \textit{PTIME} is the same on Turing machine and $\lambda$-calculus

- General problem-solving?
- Programmability
- Binding times
- Finiteness and uniformity
- Turing completeness
- Universal machine / self-interpreter

The Blob MOC:

- Originally motivated by biological computing.
- A different set of dimensions; may give some insight.
SRI is doing quality work to model biological systems using Maude, a term rewriting system implementation.

My reaction after a 2 month visit: where are the programs?

▶ Many, at the simulation level, i.e., Maude programs.

▶ But I could see no programs at the biological level.

A difference in perspective:

▶ Natural science is analytic: how does nature really work?

▶ Computer science is synthetic: build programmable systems.

This research project:

design a biology-like computing model with programs.
TWO DIFFERENT MEANINGS OF THE WORD “MODEL”

1. **Analytic** viewpoint common to the natural sciences: a “model” describes an already-existing reality.

   A good model describes the real world well, e.g., is usable to predict the outcome of not-yet-performed experiments.

2. **Synthetic** viewpoint in computer science or engineering (“model checking”). Given a problem specification, build a computer program or a hardware device to solve it.

   A good model satisfies the problem specification.

Context:

- The “confluence of ideas” had analytic overtones, suggesting that computability is a natural phenomenon.
- Turing’s work (machine design, programming) was synthetic.
CONNECTIONS EXIST BETWEEN BIOLOGY AND COMPUTATION

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.
  
  (Gödel numbers, 2-counter machine simulation, . . .)

- Our aim: a computation model where
  - “program” is clearly visible and natural, and
  - Turing completeness is not artificial or accidental or horribly inefficient, but a natural part of biomolecular computation
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: \( \exists \) 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, . . .

There are many not-yet-well understood analogies to the world of programs.
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.

▶ Many examples: given a problem, the researchers 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!)
► Program = start cell pattern? global transition function?
► There seems to be no natural universal cellular automaton.
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 \(\text{program}\):

\[
[\text{program}]\)(data_{\text{in}}) = data_{\text{out}}
\]

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

The task of programming is, given a desired semantic meaning, to devise a program that computes it.
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.
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.
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: (Bond sites 0, 2 and 3 are bound, and 1 is unbound)

```
    0
   1   2
    3
```

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)
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 $[[p]](d)$

![Diagram showing the relationship between program and data with focus points and bonds]

- Focus point for control and data (connects the PC and the DC)
- Program-to-data bond: “the bug”
WHAT HAPPENS AT THE PROGRAM-TO-DATA BOND?

Program \( p \)  Data \( d \)

\[
\begin{array}{ccc}
\text{Instruction} & \ast & \text{Focus point for control and data} \\
& & (\text{connects the PC and the DC}) \\\n\ast & = & \text{program-to-data bond}
\end{array}
\]

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 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
A MOVIE IS WORTH DURATION × FRAMERATE × 1000 WORDS

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

The data space is (also) a connected assembly of blobs.

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

- The program cursor (PC) is in \( p \).
- The data cursor (DC) is in \( d \).
- There is a bond \(*\) (“the bug”) between the PC and the DC, at bond sites 0.
EXAMPLE INSTRUCTION: \textbf{SCG 1 5}

\textbf{(SET CARGO BIT 5 TO 1)}

\begin{itemize}
  \item "The bug" \(\ast\) has moved:
    \begin{itemize}
      \item before execution, it connected PC with DC.
      \item After: it connects successor PC' with DC.
    \end{itemize}
  \item Control: activation bits 0, 1 have been swapped.
\end{itemize}

\textbf{Instruction syntax:} the 8-bit string 11001101 is grouped as

\[
\begin{array}{cccc}
  a & SCG & v & c \\
  1 & 100 & 1 & 101 \\
\end{array}
\]
Instruction form: (8 control bits and 4 bonds)

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

Why exactly 4 bonds?
- Predecessor (1 bond); true and false successors (2 bonds);
- 1 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 (though cargo bits can be used).
**INSTRUCTIONS HAVE 8 BITS**

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

as by Cardelli, Zavattaro,…

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

\[
\forall p \in L-\text{programs} \ \exists q \in M-\text{programs} \ (\llbracket p \rrbracket^L = \llbracket q \rrbracket^M)
\]

$L$ 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 . . .

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

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

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

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

Now program is a passive data object: both program and \text{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. Blob programs do not have to be encoded!

**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)
SOME DESIRABLE PROPERTIES OF A MOC

- **Existence of programs**; and **general problem solving**: a natural path from an informal algorithm to an MOC program.
- **Turing completeness**.
- **Uniformity** and **strong finiteness**: one set of hardware is enough for all problems.
- **Physical realizability**, e.g., execution possible without action at a distance, e.g., data pointers.
- **Programs as data objects**: Readability for universal machine. Writeability for program generation, e.g., a compiler.
- **Plausible program running times**, e.g., polynomially related to programming languages, e.g., $\lambda$-calculus.
Uniformity: one set of hardware is enough for all problems.
- Classic example: the universal Turing machine
- Non-example: the von Neumann architecture (maybe big, but it is finite!)

Applied to programs, uniformity implies strong finiteness: only an a priori fixed number of possible instructions. The blob MOC is strongly finite: one fixed set of instructions can compute any computable function.

(This looks unlikely – what about unbounded number of control states, or instruction labels?)

Well... we showed how to blob-simulate an arbitrary Turing machine on the architecture, using “fan-in” to implement control transfers.
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 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
- Interpreters and compilers make sense at biological level, may give useful operational and utilitarian tools.
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, data structures by static program analysis; or a type system.
- Program activation should be possible: once a program is generated, start executing it. Needs “stored program” model (as in von Neumann architecture or RASP).

Still to analyse: Time or energy to perform a single program step (may depend on program/data). An appropriate and realistic cost model, including code motion, should be found.

Concurrency (programs perhaps generated dynamically by one master program, analogous to biological reproduction.)
WHAT HAS NOT YET BEEN DONE

- Promise of tighter analogy between universality and self-reproduction.
- A usable higher-level programming language
- Find a true, biological (not just “plausible”) implementation of the fixed set of reduction rules in vitro.
- Computational complexity, e.g., limitations imposed by a 3-dimensional blob-space.
References


THANK YOU!

Questions?