

Cells Are Plausible Targets for High-Level Spatial Languages

Jacob Beal
MIT CSAIL
Cambridge, MA 02139
Email: jakebeal@mit.edu

Jonathan Bachrach
MIT CSAIL
Cambridge, MA 02139
Email: jrb@csail.mit.edu

Abstract—High level languages greatly increase the power of a programmer at the cost of programs that consume more resources than those written at a lower level of abstraction. This inefficiency is a major concern for the programming of biological systems: although advances in synthetic biology are beginning to allow bacteria to be programmed at an “assembly language” level, metabolic and chemical constraints currently place tight limits on the computational resources available. We find, however, that the semantics of the Proto spatial computing language appear to be a good match for engineered genetic regulatory networks, and particularly for describing the spatial differentiation necessary to construct tissues or organs. In this paper, we propose a mapping between Proto programs and standardized biological parts. We then demonstrate the plausibility of this mapping by applying it to a band detection program, finding that standard code optimization techniques can transform the inefficient program produced by the initial mapping into an efficient design equivalent to the Weiss laboratory’s hand-designed band detector[1].

I. INTRODUCTION

Advances in synthetic biology promise to soon give engineers intimate control over cells, with standardized biological parts allowing routine assembly of DNA to produce predictable behaviors of these designed biological systems.

If we wish to build complex multi-cellular systems, however, we are faced with a difficult compound programming problem: the engineer must describe not only how the population of cells should move, grow, and differentiate to form structures like tissues and organs, but also how this program should be distributed for execution on individual cells and how the individual cells should implement the program in DNA sequences. Thinking about all of these problems at once is extremely difficult, as can be seen in the problems suffered when programming distributed systems on ordinary computers.

For many biological systems, however, we may hope to cut this Gordian knot by viewing the aggregate of cells as a spatial computer. Bacterial colonies, biofilms, and developing embryos are all examples of spatial computers: the cells are distributed such that they fill a volume of space, and the time it takes for information to propagate from place to place in the population is strongly dependent on geometric distance. A spatial computing language such as Proto[2] might then be used to describe the behavior of the population of cells

at a high-level, delegating the routine details of distribution and implementation to the compiler.

Cells have tight constraints on the amount of resources available, however. In the systems currently being investigated by synthetic biologists, even relatively small programs impose high costs on the cell’s metabolism. In addition, the number of standardized parts currently available for constructing programs is small, and the lack of physical isolation in bacterial DNA inhibits part reuse, so programs with more than a few components cannot even be constructed, let alone inserted into cells and executed. Although both of these constraints may loosen in the future, it is unlikely that we will see a Moore’s law-style vast increase in available resources any time soon.

We are therefore left with a problem: on the one hand, high-level spatial languages appear to be necessary if we wish the engineering of complex multi-cellular systems to be humanly comprehensible. On the other hand, high-level languages generally carry a penalty in efficiency relative to hand-tuned systems constructed at a lower level of abstraction. Given the tight resource constraints of cells, if the efficiency penalty is significant, then we cannot expect to use high-level languages for programming cells.

In this paper we investigate the plausibility of high-level programming of cells using the Proto spatial computing language. Beginning with a Proto program to replicate the Weiss laboratory’s band-detector engineered bacterial system, we propose a mapping between Proto programs and standardized biological parts and use this to produce a by-hand compilation of the Proto program into a genetic regulatory network. We then demonstrate that application of ordinary code optimization techniques, again by hand, can transform the inefficient program produced by the mapping into an efficient design equivalent to the Weiss lab’s system. This result shows that, given a good match between programming model and biological parts, it is reasonable to believe that high-level languages such as Proto may be a viable approach to engineering complex biological systems.

II. BACKGROUND

This section briefly reviews the “band detector” system we use as a challenge problem, the BioBrick parts used for the low-level output of compilation, and the Proto spatial computing language that we propose as a candidate high-level

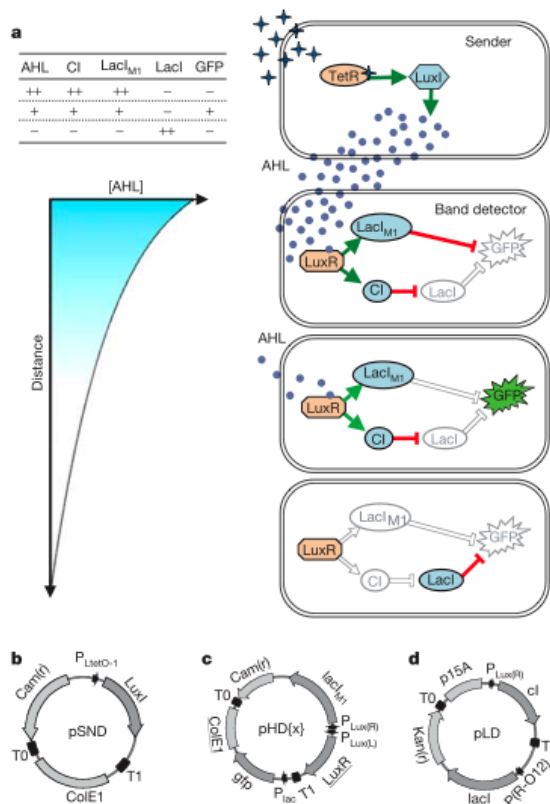


Fig. 1. The Weiss laboratory's engineered bacterial band detector, reprinted by permission from Macmillan Publishers Ltd: Nature ([1]), copyright (2005). The concentration of AHL diffused from sender cells (top right) serves as an input to a genetic regulatory network in receiver cells. When the AHL level is high or low, GFP is repressed, but when the AHL level is moderate, GFP is expressed. The sender plasmid is shown in (b), and the receiver plasmids in (c) and (d).

language for controlling engineered biological systems. We assume general familiarity on the part of the reader with both basic cell biology and programming languages.

A. Engineered genetic regulatory networks

Work in the area of synthetic biology has begun to produce libraries of standard biological parts, which can be readily assembled to create engineered genetic regulatory networks. When successful, these allow the design of living cells with predictable engineered behaviors. For example, Weiss has produced engineered "band detector" bacteria that use intercellular communication to create spatial patterns of fluorescence[1], [3]. We choose the band detector system as a target for replication because it is a known working engineered biological system with a spatial component.

Weiss's band-detector system (shown in Figure 1) uses two populations of bacteria. One population is senders: the presence of anhydrotetracycline (aTc) induces this population to produce the LuxI enzyme. LuxI catalyzes the production of AHL, a signalling chemical that diffuses through cell membranes. The second population are receivers: when AHL diffuses into cells of this population, it binds to the LuxR

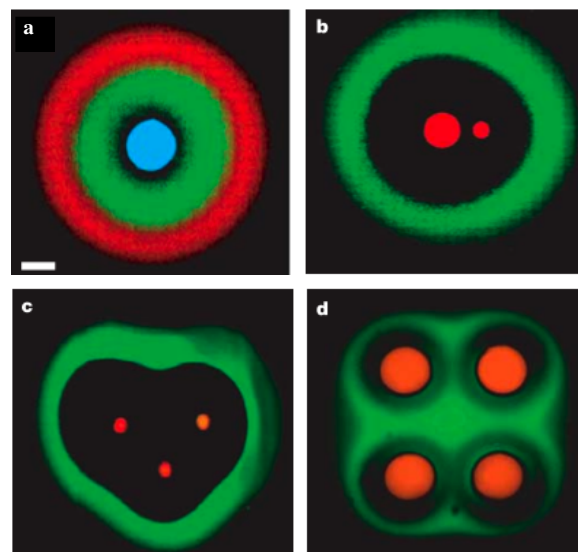
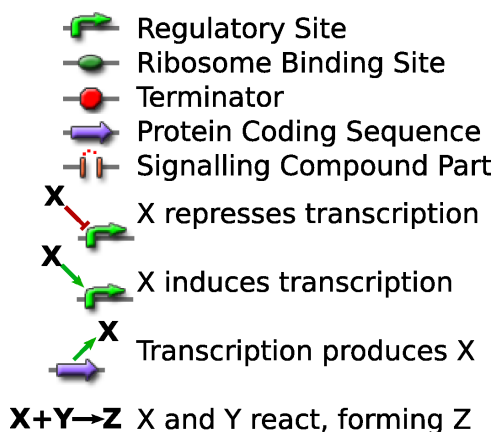


Fig. 2. Examples of the Weiss lab band detector in use, reprinted by permission from Macmillan Publishers Ltd: Nature ([1]), copyright (2005). The circular regions in the center are active senders, while the fuzzy colored areas are receivers expressing fluorescent protein.

transcription regulator and induces activity on two pathways. One pathway detects high levels of AHL, the other detects low levels, and both act to suppress production of a fluorescent protein such as GFP. Accordingly, cells at a moderate distance to the active senders, which receive moderate levels of AHL, fluoresce, while cells closer or farther away do not. Figure 2 shows images of band detector colonies in operation.

In this paper, we consider systems constructed from parts conforming to the BioBricks assembly standard. This standard, proposed by Knight in 2003[4], allows the assembly of conforming parts into arbitrary sequences, as any two BioBricks can be composed and the composition of any two BioBricks is itself a BioBrick: source plasmids are assembled to form a composite plasmid, following one of several protocols, and the composite plasmids are inserted into bacteria (generally *e. coli*). Publicly available BioBricks are collected in the Registry of Standard Biological Parts[5], [6] and have been used successfully by many laboratories, including undergraduates teams in the iGEM genetic engineering competition[7]. Although the band detector we wish to replicate is not constructed from BioBricks, we have chosen to consider BioBricks because their composition standard is amenable to automated design generation by a compiler.

Figure 3 shows the "language" of BioBricks that we use for describing engineered genetic regulatory networks, along with an intercellular communication composite part and a typical composite for a functional unit in a genetic regulatory network. A typical functional unit is composed of four parts: a ribosome binding site preceded by one or more regulatory sites and followed by one or more protein coding sequences, which are in turn ended with a transcription terminator. When at least one regulatory site is promoting binding and none are repressing it, DNA is transcribed into RNA, which is translated into the



Typical functional unit:

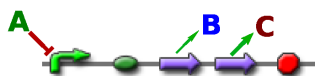


Fig. 3. Standard biological parts, as encoded by the registry, and the notation we use to show the interaction between parts and chemicals. Arrow size shows the strength of a relationship. Note that the signalling part is, itself a complex composite and may even be split between plasmids or cell populations, as in the Weiss lab design.

encoded proteins, increasing their concentration in the cell.

Computation is generally implemented through the influence of transcription factors on regulatory sites. We indicate this with arrows connecting chemical names to regulatory sites: a chemical name X and green arrow means a site that is normally inactive but can be induced to activity by the presence of X , while a red bar means a site that is normally active but can be repressed by the presence of X . The wider the arrow, the stronger the induction or repression, and therefore the lower concentration of transcription factor needed to affect behavior. If there is no arrow, then the regulatory region is constitutively active.

For example, the composite part shown in Figure 3 normally produces a standard amount of protein C and a small amount of protein B . When protein A is present, it suppresses transcription, and levels of B and C in the cell will drop.

In practice, assembly of BioBricks to produce working *in vivo* systems is fraught with complications. Major current challenges include maintenance of consistent signal levels, switching speed, interference with cell metabolism, unexpected secondary structure in composed sequences, and interference between transcription factors. As progress is being made in all of these areas, however, we feel it is reasonable to discuss computation on the abstract genetic regulatory networks that the BioBricks community aims to enable.

B. The Proto language

One promising approach to the challenges of spatial computing is to focus not on the network of devices, but on the continuous space that they occupy, using the *amorphous medium* abstraction. An amorphous medium[2] is a manifold

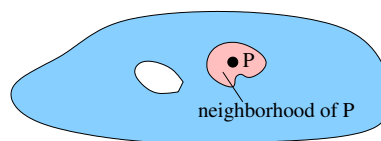


Fig. 4. An amorphous medium is a manifold where every point is a device that knows its neighbors' recent past state.

```

(def band-detect (signal lo hi)
  (and (> signal lo) (< signal hi)))

(let ((signal (diffuse (aTc) 0.8 0.05)))
  (green (band-detect signal 0.2 1)))
  
```

Fig. 5. Proto code for a band detector.

with a computational device at every point, where every device knows the recent past state of all other devices in its neighborhood (Figure 4). While an amorphous medium cannot, of course, be constructed, it can be approximated on the discrete network of a spatial computer.

Our language, Proto[2], uses the amorphous medium abstraction to factor programming a spatial computer into three loosely coupled subproblems: global descriptions of programs, compilation from global to local execution on an amorphous medium, and discrete approximation of an amorphous medium by a real network.

Proto is a functional language that is interpreted to produce a dataflow graph of operations on fields; for the purpose of this paper, we assume that all function calls are inlined in the graph, though that need not be the case in general. This program is then evaluated against a manifold to produce a field with values that evolve over time. Proto uses four families of operations: point-wise operations like $+$ that involve neither space nor time, restriction operations that limit execution to a subspace, feedback operations that establish state and evolve it in continuous time, and neighborhood operations that compute over neighbor state and space-time measures and summarize the computed values in the neighborhood with a set operation like *integral* or *minimum*.

With appropriate operators, compilation and discrete approximation are straightforward. Thus, Proto makes it easy for a programmer to carry out complicated spatial computations using simple geometric programs that are robust to changes in the network and self-scale to networks with different shape, diameter, density of nodes, and execution and communication properties[8].

For example, a band detector can be implemented using the short program shown in Figure 5, where *aTc* is a function for sensing aTc and *green* is an actuator that sets the level of GFP expression. On a wireless network an implementation of *diffuse* (whose parameters are an indicator function for sources, a diffusion constant, and a decay constant) adds five more lines of code, including feedback and neighborhood operations, but in our biological implementation it will be implemented directly using a signalling part. Figure 6 shows

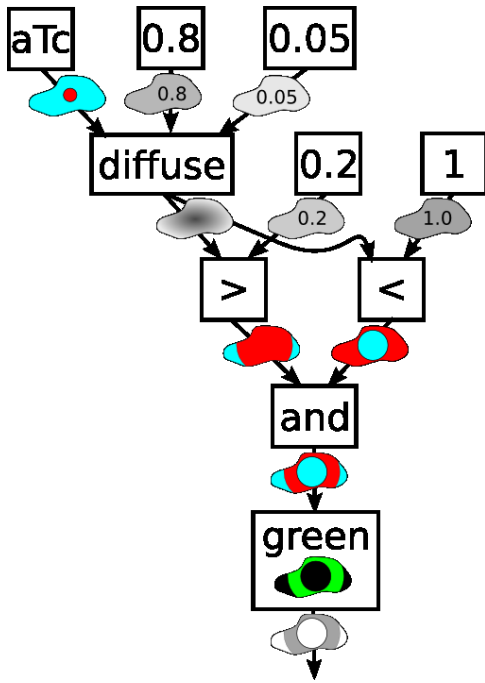


Fig. 6. A Proto program specifies a dataflow graph of operations on fields. When evaluated on a space, each operation produces a field of values over that space. Here the band detector program is shown evaluated on an irregularly shaped space, with scalar fields grey (lighter is less) and boolean fields colored (**true** is red, **false** is blue). The actuation produced by **green** is shown inside that operation.

the Proto band detector program interpreted to produce a dataflow graph, then evaluated against an irregularly shaped space.

Executing the Proto band detector in simulation produces results equivalent to Weiss’s band detector. Figure 7 shows execution on a network of 2000 simulated wireless devices distributed randomly through a 100 by 100 unit region with a 10 unit communication radius.

Other spatial computing approaches: Proto is only one of many high-level approaches to programming spatial computers. Other major approaches include viral code systems, (e.g. TOTA co-fields[9], paintable computing[10]) Regiment[11], which gathers streaming data from spatial regions, Kairos[12], which operates on abstract graphs, pattern languages like Origami Shape Language[13] and Growing Point Language[14], and modular robotics systems (e.g. [15]). Previous work toward a general purpose spatial computing language includes other work by Coore[16] and the Amorphous Medium Language[17]. Crystalline systems such as cellular automata are a special case of spatial computers, and insight from that area is beginning to be applied to spatial computing[18], including models of morphogenesis. Proto, however, appears uniquely well suited for translation into genetic regulatory networks, as is shown in the next section.

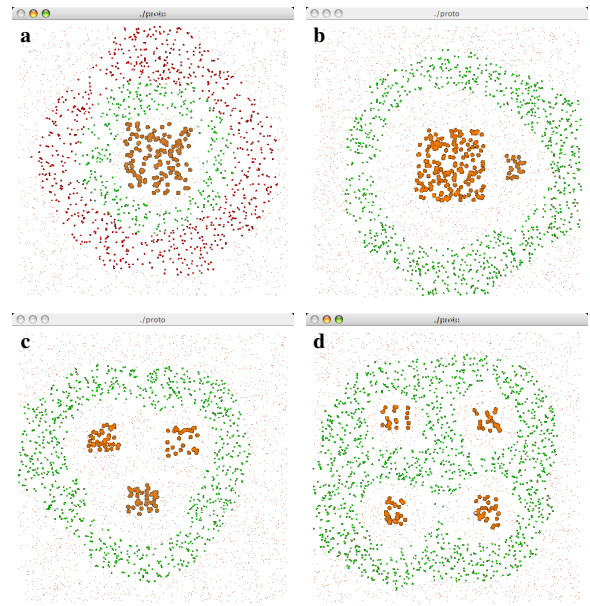


Fig. 7. A band detector implemented in Proto produces equivalent results to the Weiss lab band detector. Subfigure (a) creates the two color pattern by calling the band-detect function twice inside the **let**.

III. BIOLOGICAL IMPLEMENTATION OF PROTO PRIMITIVES

Proto’s semantics appear to be a good match for implementation in a genetic regulatory network. The continuous space abstraction used by Proto means that scaling to cell populations in the billions or trillions should be no challenge. Likewise, the parallel, unsynchronized operation of a genetic regulatory network should pose no problem for Proto because it is a purely functional language: state in Proto comes only from the values flowing over edges in the dataflow graph, just as state in the BioBrick genetic regulatory networks we are considering comes primarily from the concentration of chemicals within the cell.

We therefore propose to represent edges as chemicals (generally transcription factors) and Proto primitives as composed patterns of BioBricks. For purposes of this paper, we discuss implementations for only those families of primitives needed to implement the band detector.

a) Logical, Branching: Inverters were one of the first targets of work in synthetic biology (see, for example, [19]), and can be implemented as repression of a regulator, giving us a **not** operation. Figure 8(a) shows the pattern for a **not**: A is the input, which represses production of out . If the transfer curve between levels of A and out obeys the static discipline, then this composition of parts operates as an inverter, and the concentrations of A and out are interpreted as digital values.

Adding a second repressible regulator produces a **nor** that is repressed when either input A or B is present. If the inputs induce rather than repress, they implement an **or** operation instead. These operations can be used to build any logical primitive. For example, **and** can be built out of three inverters, effectively computing $(\text{nor}(\text{not } A) (\text{not } B))$.

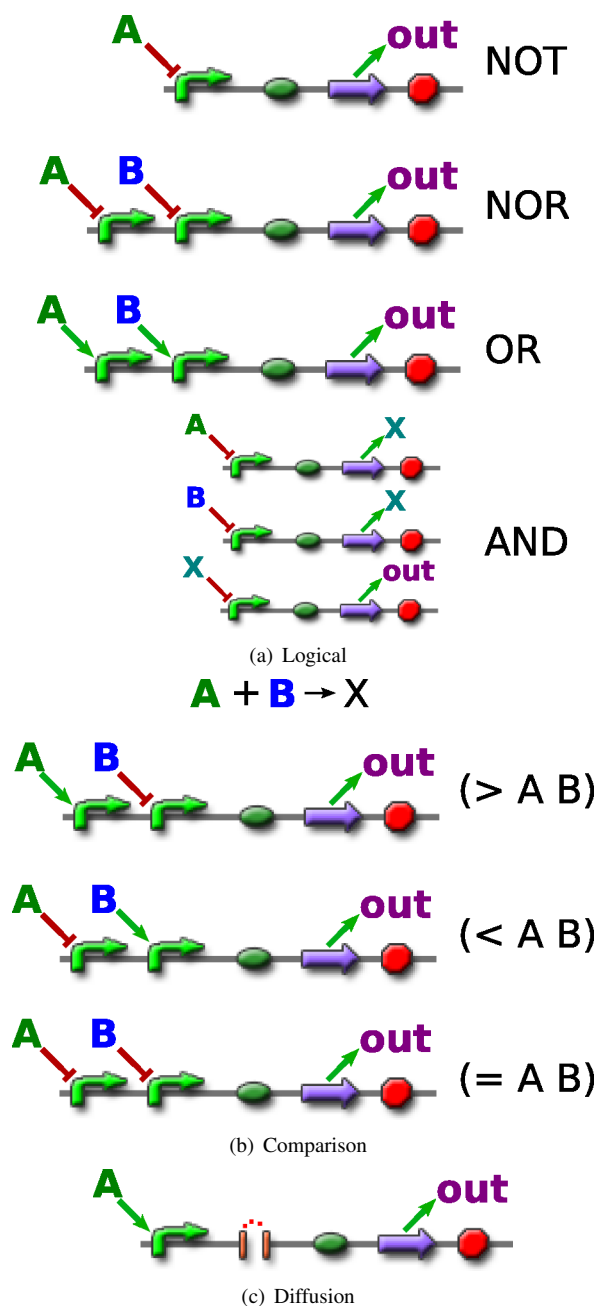


Fig. 8. Genetic regulatory network implementations for selected Proto primitives.

Branching operations can also be implemented using logical operations. For example, *where*—the Proto equivalent of *if*, restricting execution to a subspace—can be implemented by using the output of the test expression to repress the inactive branch.

b) Arithmetic: While digital arithmetic can be implemented using logical operators, doing so is extremely expensive. A one-bit full-adder, for example, requires five gates to implement. As an alternative, one might consider representing scalar numbers as analog values using chemical concentration.

In this approach, a numeric literal can be represented by a

constitutively active protein coding sequence. The $+$ operator can be implemented simply by having its operands be represented by the same chemical. Subtraction can be implemented using a chemical reaction: if chemicals A and B react to produce X , then $(- A B)$ may be represented by the level of A remaining for positive numbers and the level of B remaining for negative numbers. Other operations such as multiplication, division, and exponentiation are more complicated but might still be implementable using methods adapted from electronic analog computing.

There are significant questions that must be resolved to make such analog computation usable, particularly those of range, resolution, and timing. Nevertheless, given the limited resources available in cells, it seems advisable to pursue analog representations.

c) Comparison: Given a representation of scalars as analog values, comparison operators such as $<$ are effectively operating as analog-to-digital converters. A promoter with a sharp transition in its input/output transfer curve can be considered a one-bit analog-to-digital converter, testing whether a chemical is present in any significant concentration.

We can thus implement comparisons using chemical reactions to subtract the two chemicals being compared, then using the remnant that is not consumed by the reaction to induce or repress the output chemical (Figure 8(b)). This is a “fuzzy” comparison that produces a clear answer only when the difference is in the saturated region of the induction or repression transfer curve, but may suffice for many programs—certainly it should be sufficient for the band detector.

d) Neighborhood Operations: Communication in Proto is described implicitly through operations on the collection of values held by neighbors. This may be very hard to implement on biological systems, however, due to the difficulty of distinguishing which inputs come from which neighbors.

Much can be done using only diffusive communication, so for now rather than implement neighborhood operations, we implement only the *diffuse* function, as a special case using a signalling part (Figure 8(c)). The input is a boolean field—locations where it is true are sources of the diffusing chemical—and diffusion constants are set by the implementation of the signalling part, assuming that they are in a realizable range. Note that the signalling part is, itself a complex composite and may even be split between plasmids or cell populations, as in the Weiss lab design.

e) Sensors and Actuators: Finally, we need chemical sensors and actuators. In the case of the band detector these are simple. There is a promoter that is induced by *aTc*, so the *aTc* sensor primitive may be implemented simply using a composite part headed by that promoter. Conversely, the *green* actuator may be implemented with a composite part that includes a protein coding sequence for green fluorescent protein (GFP).

IV. COMPILATION

Given this mapping from Proto operators to biological parts, compilation is straightforward. First, as usual, the Proto code

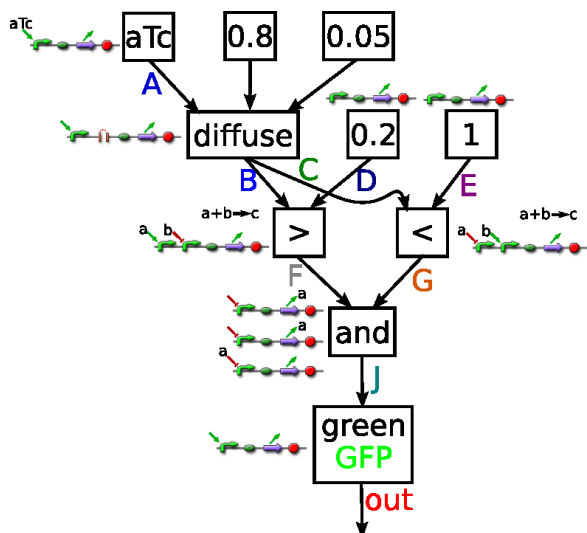


Fig. 9. Proto code is interpreted to produce a dataflow graph. Then each edge in the graph is assigned a chemical and each operation an implementing pattern of biological parts. Lower case letters are variables used to indicate where the same chemical must be used in a complex pattern, though that chemical is not yet assigned.

is interpreted to produce a data-flow graph of operations on streams of fields. To map this to a genetic regulatory network, each operator is assigned its corresponding pattern of biological parts, and each edge assigned a unique chemical. Figure 9 shows this first stage of compilation for the band detector code from Figure 5 (note that the diffusion operator is handled as a special case, folding the constants in immediately). The chemicals and biological annotations are then connected together to create an abstract genetic regulatory network that implements the program (Figure 10). The next stage, from abstract network to actual components incorporated into one or more plasmids and one or more cell populations, is not treated because it is already an active area of synthetic biology research.

The resulting biological design is complicated and redundant. This is no surprise: the naive translation of high-level programs into executable code often is. What is important is that we have a complete design that likely could be realized with a sufficient investment of laboratory work, and this naive design provides a starting point for the process of optimization.

V. OPTIMIZATION

Although the program shown in Figure 10 is an implementation of a band detector, it is much less efficient than the Weiss lab band detector. Applying common code optimization techniques as described below, however, produces an optimized program (Figure 11(f)) that is equivalent to the abstract genetic regulatory network for the Weiss lab's hand-tuned design. This surprisingly good result shows two things: that using high-level languages to create efficient biological designs is plausible, and that we may be able to use readily available compilation tools to do so.

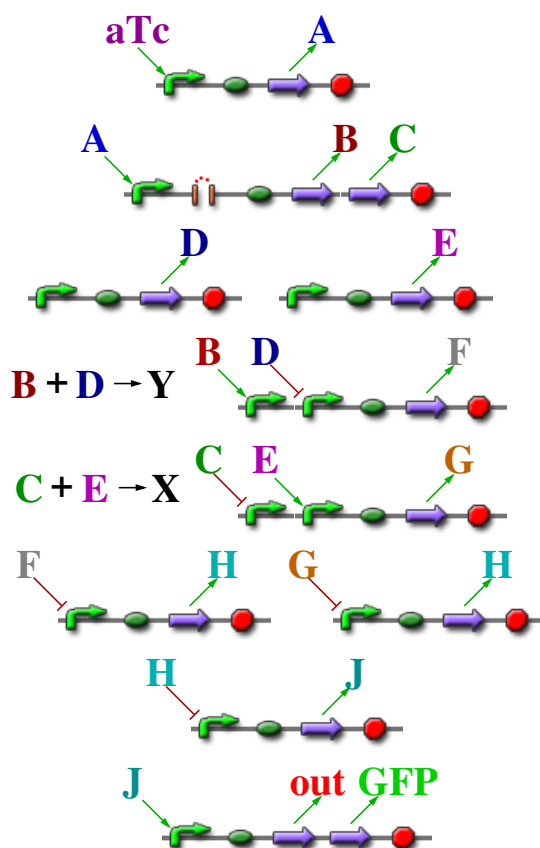


Fig. 10. The chemicals and biological parts assigned to a Proto dataflow graph are connected together to create an abstract genetic regulatory network implementing the program.

Table I shows the sum of various resources used by the Weiss lab design, the naive compiled Proto code, and the optimized version. The costs count aTc as an intercellular messenger and, for the Proto implementations, assume that the intercellular signalling part is implemented using parts BBa_F1610 and BBa_F2621 from the Registry of Standard Biological Parts[5], for a total cost of one signal-carrying chemical, one intercellular messenger, two coding regions, and two promoters. The naive compiled version is clearly much less viable: the greater resources required mean greater drain on a cell's resources as it operates and more potential interactions between components, making it much more difficult to produce a working *in vivo* system. Note, however, that these costs are for the abstract network only, as we are not treating the question of how the network is realized with particular BioBricks, then divided into plasmids and cell populations.

Many standard methods for optimizing computer programs, however, can be applied to such a genetic regulatory network. The naive network fairly begs to have such techniques applied: even a cursory glimpse shows that the *out* chemical is unneeded and that chemical *J* is redundant.

We applied standard code optimization techniques by hand in order to see how much improvement could be easily gained. Some (like copy propagation) should be immediately

| Resource | Hand Tuned | Naive | Optimized |
|--------------------------|------------|-------|-----------|
| Signal-carrying chemical | 3 | 11 | 3 |
| Protein coding sequence | 6 | 14 | 6 |
| Promoters | 5 | 14 | 5 |
| Intercellular messengers | 2 | 2 | 2 |
| Chemical reactions | 0 | 2 | 0 |

TABLE I
RESOURCES REQUIRED BY BAND DETECTOR: THE WEISS LAB’S
HAND-TUNED DESIGN VERSUS NAIVELY COMPILED PROTO AND
OPTIMIZED COMPILED PROTO.

applicable to any Proto program, while others (like constant elimination) are generic techniques which need to be applied differently on different platforms. Figure 11 shows the sequence of reductions from applying five common code optimization techniques, adapted to genetic regulatory networks.

A. Constant Elimination

Most computing idioms include special-purpose operations that implement particular computations quickly and simply. For example, computers often include increment and decrement instructions that compute much faster than general purpose addition.

Comparison with a constant is such an operation for genetic regulatory networks. Instead of representing the constant by a constitutively produced chemical species, we represent it by the strength of a promoter. For example, testing that B is above a low threshold can be done using a strong positive promoter and testing that C is below a high threshold can be done using a weak negative promoter. We can thus eliminate both reactions and chemicals D and E , producing Figure 11(a).

B. Algebraic Simplification

Sometimes briefly adding complexity to code enables greater simplification. For example, inverting the comparison operations (Figure 11(b)) produces a double negative than can be eliminated, producing the simpler design in Figure 11(c).

C. Dead Code Elimination

Values and expressions in a program are “live” if they will be used and “dead” if they will not be. Dead portions of a program can be freely eliminated without changing the function of a program. In this case, the *out* signal produced by the band detector is not used—we are interested instead in the GFP that is produced as a side effect of the last operation. We can thus snip the *out* chemical from the design, producing Figure 11(d).

D. Copy Propagation

When a value is copied to an intermediate variable, later expressions can equivalently use either the copy or the original. Copy propagation eliminates uses of the copy wherever possible, hoping to eliminate the intermediate value.

In this case, copy propagation sets H to regulate *GFP* directly, rather than being copied by the intermediate chemical J . Since J is no longer used, a round of dead code elimination removes it. A similar process eliminates chemical A , producing Figure 11(e).

E. Use-Definition Analysis

Finally, optimizations can be performed by analyzing a variable’s definition and use cases. For example, many compilers will move a computation out of a loop when its value is not affected by the progress of the loop.

In this case, we can make an optimization based on knowing that the uses of a value are positive promoters of the same strength. Since chemical C is used by only one positive promoter, we can get equivalent signal strength by returning the promoter to the standard level and modifying the expression level of C instead. This turns the C to H relation into a copy, so that a round of copy propagation and dead code elimination eliminate C , resulting in the design shown in Figure 11(f).

This final design is equivalent to the genetic regulatory network produced by the Weiss laboratory and shown in Figure 1.

VI. CONTRIBUTIONS

We have shown that a Proto program for band detection can be translated into a plausible design for a genetic regulatory network, compiling by hand with a mapping from Proto primitives to assemblages of standardized biological parts. Moreover, applying standard code optimization techniques to the compiled program can produce an optimized version equivalent to a hand-designed band-detection program that actually runs on bacteria.

These results suggest that it is reasonable to consider using high-level spatial computing languages such as Proto to design complex multi-cellular biological systems. There are, however, serious obstacles to making this vision a reality. Most pressing, of course, are the issues of compositionality, interference, and resource management that are a major focus of study in the synthetic biology community: real biological systems often show unexpected interactions between parts, have stochastic and noisy behavior, and show graded responses to stimuli, and it is an open question whether these can be managed well enough to allow *any* high-level language approach to design. Beyond this, there is a great gap of implementation between demonstrating that efficient compilation is possible and actually producing an effective optimizing compiler. Implementing an optimizing compiler is expected to be mainly a matter of software engineering, so since Proto is a free open-source project, we invite the interested reader to participate in doing so.

Given that the problems of implementation and distribution appear to be solvable, however, we may also ask what sort of high-level programs are useful for succinctly describing the behavior desired from multi-cellular biological systems. Besides bearing on the obvious questions of engineering, the struggle to answer this question may shed light on deep questions of development and evolution, for the problems faced by a growing engineered multi-cellular system are not unlike those of a developing embryo.

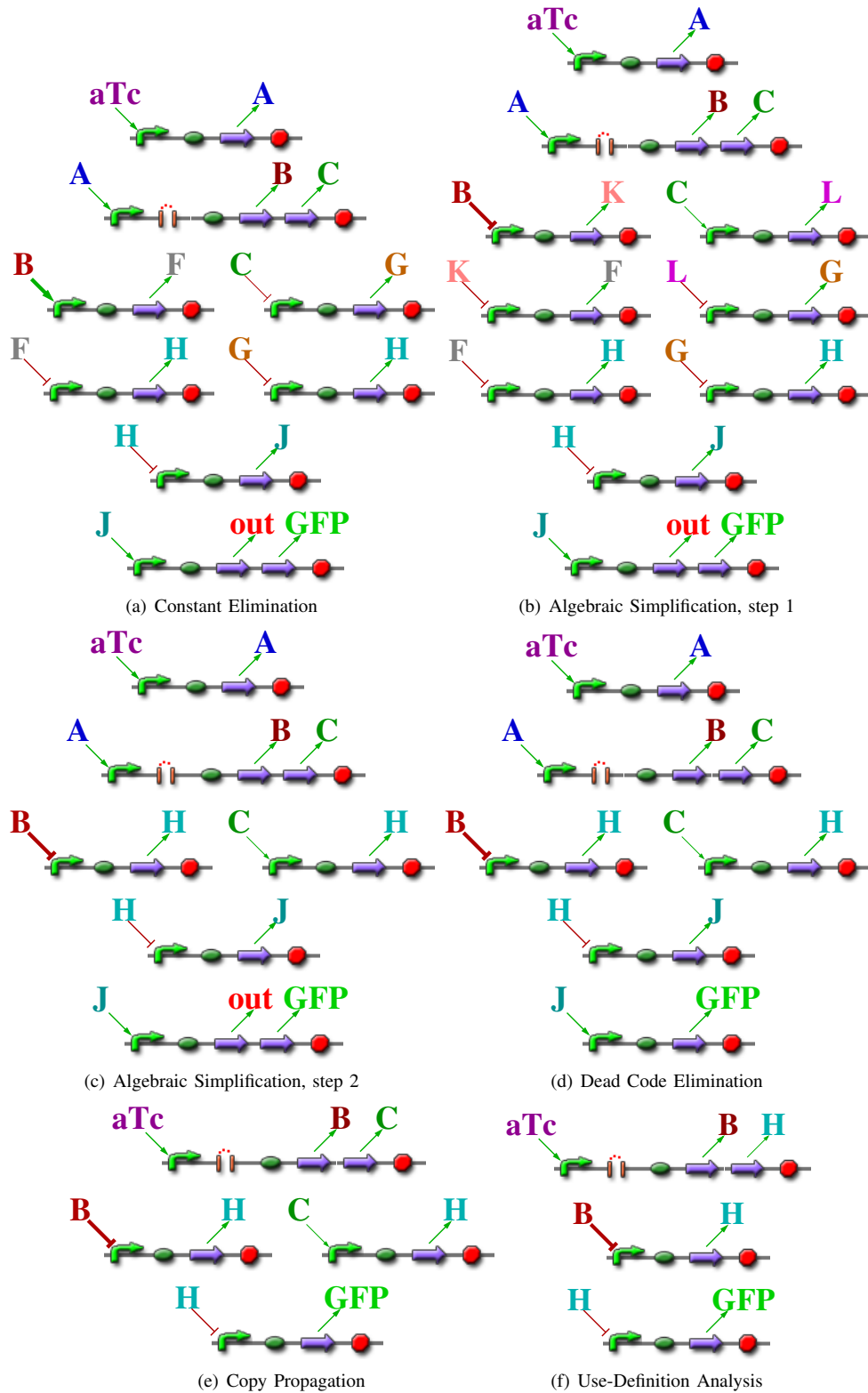


Fig. 11. Application of standard code optimization techniques greatly simplifies the initial genetic regulatory network from Figure 10. First, the comparisons are modified to get rid of constants (a), then inverted (b) to create a double-negative that can be eliminated (c). Next the unused *out* chemical is discarded (d), followed by unneeded copy stages (e), and a deamplifier changed into weak expression (f).

REFERENCES

- [1] Subhayu Basu, Yoram Gerchman, Cynthia H. Collins, Frances H. Arnold, and Ron Weiss, "A synthetic multicellular systems for programmed pattern formation," *Nature*, vol. 434, pp. 1130–1134, April 2005.
- [2] Jacob Beal and Jonathan Bachrach, "Infrastructure for engineered emergence in sensor/actuator networks," *IEEE Intelligent Systems*, pp. 10–19, March/April 2006.
- [3] Ryan McDaniel and Ron Weiss, "Advances in synthetic biology: on the path from prototypes to applications," *Current Opinion in Biotechnology*, vol. 16, pp. 476–483, 2005.
- [4] T.F. Knight, "Idempotent vector design for standard assembly of bio-bricks," Tech. Rep., MIT Synthetic Biology Working Group Technical Reports, 2003.
- [5] Barry Canton, Anna Labno, and Drew Endy, "Refinement and standardization of synthetic biological parts and devices," *Nature Biotechnology*, vol. 26, pp. 787–93, July 2008.
- [6] Reshma P Shetty, Drew Endy, and Jr Thomas F Knight, "Engineering biobrick vectors from biobrick parts," *Journal of Biological Engineering*, vol. 2, no. 5, 2008.
- [7] J. Brown, "The igem competition: building with biology," *IET Synthetic Biology*, vol. 1, pp. 3–6, 2007.
- [8] Jonathan Bachrach, Jacob Beal, and Takeshi Fujiwara, "Continuous space-time semantics allow adaptive program execution," in *IEEE SASO 2007*, July 2007.
- [9] M. Mamei and F. Zambonelli, "Programming pervasive and mobile computing applications: the tota approach," *ACM TOSEM*, to appear 2008.
- [10] William Butera, *Programming a Paintable Computer*, Ph.D. thesis, MIT, 2002.
- [11] Ryan Newton and Matt Welsh, "Region streams: Functional macro-programming for sensor networks," in *First International Workshop on Data Management for Sensor Networks (DMSN)*, Aug. 2004.
- [12] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan, "Macro-programming wireless sensor networks using *kairos*," in *DCOSS*, 2005, pp. 126–140.
- [13] Radhika Nagpal, *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*, Ph.D. thesis, MIT, 2001.
- [14] Daniel Coore, *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer*, Ph.D. thesis, MIT, 1999.
- [15] Kasper Stoy, "Controlling self-reconfiguration using cellular automata and gradients," in *8th int. conf. on intelligent autonomous systems (IAS-8)*, 2004.
- [16] Daniel Coore, "Towards a universal language for amorphous computing," in *Fifth International Conference on Complex Systems*, 2004.
- [17] Jacob Beal and Gerald Sussman, "Biologically-inspired robust spatial programming," Tech. Rep. AI Memo 2005-001, MIT, January 2005.
- [18] Daniel Yamins, *A Theory of Local-to-Global Algorithms for One-Dimensional Spatial Multi-Agent Systems*, Ph.D. thesis, Harvard, December 2007.
- [19] Ron Weiss, *Cellular Computation and Communications using Engineered Genetic Regulatory Networks*, Ph.D. thesis, MIT, 2001.