Programs as Polypeptides

Lance R. Williams*
University of New Mexico

Keywords
Actor model, asynchronous cellular automaton, artificial chemistry, combinator, nondeterminism, monad, self-replicating system, spatial computing

Abstract *Object-oriented combinator chemistry* (OOCC) is an artificial chemistry with composition devices borrowed from object-oriented and functional programming languages. *Actors* in OOCC are embedded in space and subject to diffusion; since they are neither created nor destroyed, their mass is conserved. Actors use programs constructed from combinators to asynchronously update their own states and the states of other actors in their neighborhoods. The fact that programs and combinators are themselves reified as actors makes it possible to build programs that build programs from combinators of a few primitive types using asynchronous spatial processes that resemble chemistry as much as computation. To demonstrate this, OOCC is used to define a parallel, asynchronous, spatially distributed self-replicating system modeled in part on the living cell. Since interactions among its parts result in the construction of more of these same parts, the system is *strongly constructive*. The system's high *normalized complexity* is contrasted with that of a simple composome.

1 Introduction

Much as Turing [36] had done when motivating his abstract computing machine by comparing it to a human "computer" executing programs with paper and pencil, von Neumann [38] began his study of *self-replication in the abstract*, by thinking about a *concrete* physical machine. As imagined, von Neumann's *kinematic automaton* assembled copies of itself from a supply of components undergoing random motion on the surface of a lake. The components consisted of girders, sensors, effectors, logic gates, and delays, together with tools for welding and cutting. It is unlikely that von Neumann ever intended to actually *build* a physical self-replicating machine. More likely, he regarded the kinematic automaton as a thought experiment, and abandoned it when he understood how the problems of self-reference, control, and construction that truly interested him could be rigorously formulated in the abstract domain of *cellular automata* (CAs).

By abandoning his kinematic automaton, von Neumann became the first "player" of a sometimes abstruse "game" that many others have played since [30]. This "game" has two parts and two pitfalls. Roughly speaking, the parts are: *define a model of computation*, and *define a self-replicating object (or system) in the model*. The two pitfalls, which must be avoided if the "game" is to be nontrivial, are: *making the computational model too abstract*, and *making the primitives too complex*. For example, it is trivial for a self-replicating object defined as a 1 to replicate in an array of 0s if physics is defined to be a Boolean OR operation in neighborhoods. It is equally trivial for a self-replicator composed of a robotic arm, a

* Department of Computer Science, University of New Mexico, Albuquerque, NM 87131, USA. E-mail: williams@cs.unm.edu

camera, and a computer to make copies of itself given a supply of robotic arms, cameras, and computers. Von Neumann himself was very conscious of the parts and pitfalls and discusses at length the tradeoffs the "game" presents.[1] His ingenious solution was (characteristically) a saddle point, combining a fiendishly simple model of computation and an enormously complex self-replicating object.

Nearly sixty years after von Neumann's death, no one has yet constructed a kinematic automaton of the kind he imagined [10]. However, because the "game" seems to many of us to still afford the best prospect by which to address the twin problems of the origin of life on Earth and its evolution into forms of increasing complexity, there is no shortage of new "players." Fortunately, current "players" are the beneficiaries of a wealth of biological science unknown to von Neumann [41], of significant advances in the science of computing that von Neumann and Turing played seminal roles in founding, and of a growing body of work in the interdisciplinary fields of artificial life and complex systems.

Unsurprisingly (given the preceding), this article contains descriptions of both a new model of computation and a self-replicating system defined using that model. The design of *our* model is strongly influenced by the belief that something important was lost when von Neumann adopted cellular automata as *his* model. More specifically, we believe that *conservation of mass*, a law that all machines that assemble copies of themselves from parts must obey, was (in effect) the baby thrown out with the bath water.

Our starting point is *artificial chemistry* [7], the study of the population dynamics of systems of constructible objects, which Fontana and Buss [9] called *constructive dynamical systems*. Like them, we looked to the field of computer science for inspiration, hoping to repurpose elements of modern *object-oriented* and *functional* programming languages as primitives and composition devices in an artificial chemistry. From object-oriented programming we borrowed the ideas of object composition and the association of programs with the data they operate on; from functional programming, we borrowed the idea of construction of programs by composition of program fragments, or *combinators*.

The first pitfall (excess abstraction) is avoided using a twofold strategy. First, we make our artificial chemistry concrete by embedding its constructed objects in a 2D space and relying solely on diffusion for dynamics. Significantly, to make this tractable, aggregates are treated as *masses* (unlike Arbib [2], who treated aggregates as *areas*), and mass is conserved, which makes it a more plausible host for a kinematic automaton of the sort imagined by von Neumann.[2] Second, as a guarantee of a different kind of realism, we insist that our artificial chemistry must be a *bespoke physics* as defined by Ackley [1]. More specifically, it must function as an abstract interface to a physically realizable *indefinitely scalable* computational substrate. This is consistent with the notion that kinematic automata defined using such interfaces, and which replicate by assembly of conserved parts, are tantamount to physical machines.

We avoid the second pitfall (complex primitives) by using (admittedly) complex primitives to build a self-replicating system composed of parts that are *still more complex*. However, these more complex parts are constructed by the system itself! It follows that the system is *strongly constructive* in the sense that interactions among its parts result in the construction of more of these same parts [7]. Our inspiration, the ribosome, allowed us to imagine programs as enzymes and to define a pair of representations for programs, one spatially distributed and inert, the other compact and metabolically active. The self-replicating system that resulted is a parallel, asynchronous, spatially distributed computation modeled in part on the living cell. See Figure 1.

The model of computation described in this article has features in common with prior work on automata and artificial chemistry. The idea of movable aggregates of complex automata has

1 See von Neumann [38, pp. 76–77] and Arbib [2, p. 179].

2 Abstract models can sometimes be improved by adding conservation principles that make them more concrete. For example, di Fenizio [6] showed how adding conservation of mass eliminates the more contrived elements of the artificial chemistry of Fontana and Buss [9].

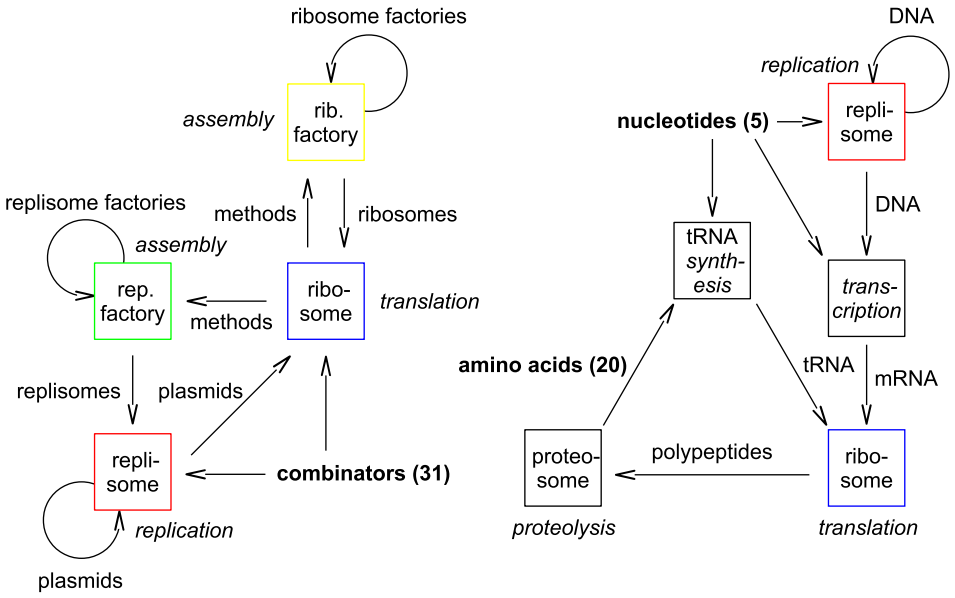

Figure 1. Self-replicating system of ribosome and replisome factories built in an object-oriented combinator chemistry (OOCC) (left). Fundamental dogma of molecular biology (right).

precedent in Arbib [2]. The idea that kinematic automata can be built in embedded artificial chemistries has precedent in Laing [18], Smith et al. [31], and Hutton [15]. The use of nested multisets for object composition in an artificial chemistry has precedent in Paun [27]. The use of sequences of combinators as molecules and conservation of mass has precedent in di Fenizio [6]. The idea of molecules as programs has precedent in Laing [18], Fontana and Buss [9], di Fenizio [6], and Hickinbotham et al. [13]. The idea that programs for stack machines are, by virtue of their high degree of composability, well suited to evolutionary computation has precedent in Spector and Robinson [33]. Finally, Taylor [35] has argued that embeddedness and competition for matter, energy, and space are necessary in artificial life systems capable of open-ended evolution.

2 Relative Complexities of Artificial Organisms and Virtual Worlds

Pattee [26] has described the simulation of an organism in a virtual world as an initial-value problem where organisms are contingent states subject to the non-contingent laws of physics. In the "game" of inventing both, there is a tradeoff between the *non-contingent complexity* of models of physical law, and the purely *contingent complexity* of artificial organisms defined inside those models. If physical law is too powerful, self-replication becomes trivial; life is too easy. Conversely, if physical law is not powerful enough, self-replication becomes impossible; life is too hard. It's possible that the most interesting "games," those resulting in a bootstrapping process that culminates in organisms capable of open-ended evolution, are grounded in physical law "just powerful enough." Von Neumann's universal replicator R_V and its CA virtual world CA_V suggest that interesting solutions to the "game" are saddle points, maximizing the ratio of contingent $K(R_V \mid CA_V)$ to noncontingent complexity $K(CA_V)$:

$$\frac{K(R_V \mid CA_V)}{K(CA_V)}, \tag{1}$$

where K is Kolmogorov complexity [17].[3] To explore this idea, let's consider a hierarchy of computational models; each model is built using an interface exposed by a more fundamental model. For example, the problem of simulating a CA with a more fundamental CA is described by Smith [32]. Among many other things, he showed that any CA with a Moore neighborhood (8 neighbors) can be simulated by a CA with a von Neumann neighborhood (4 neighbors) with an increase in space and a slowdown in time by constant factors that depend only on the numbers of states in the CAs being simulated:

$$CA_8\left(\mathbb{Z}^2\right) \leq_1 CA_4\left(\mathbb{Z}^2\right), \tag{2}$$

where CA_8 and CA_4 are CAs with Moore and von Neumann neighborhoods, $\mathbb{Z}^2$ is the integer lattice, and $(\leq_1)$ is Smith's $O(1)$ reduction.

Asynchronous cellular automata (ACAs) are much like cellular automata except that the local state is updated asynchronously [8]. It is possible to demonstrate by construction that any CA can be simulated by an ACA with an increase in space [24, 25] and a slowdown in time [3] by constant factors that depend only on the number of states and neighborhood size of the CA. Consequently,

$$CA\left(\mathbb{Z}^2\right) \leq_1 ACA\left(\mathbb{Z}^2\right), \tag{3}$$

where $(\leq_1)$ is Nakamura's $O(1)$ reduction. Our approach is premised on the idea that models in the *object-oriented combinator chemistry* (OOCC) defined in this article can be compiled to ACAs of one higher dimension.[4] Objects are instances of a recursive data type grounded in a small number of primitive types and closed under two forms of composition. The extra dimension is used to represent the internal structure of composed objects, and the size of this representation is defined as an object's *mass*:

$$OOCC\left(\mathbb{Z}^2\right) \leq_1 ACA\left(\mathbb{Z}^2 \times \mathbb{N}\right), \tag{4}$$

where $(\leq_1)$ is the hypothesized compilation process. Unlike Arbib [2], who assumed that arbitrarily large automata aggregates could be moved $O(1)$ distance in $O(1)$ time, we assume only that objects of mass m can be moved $O(1)$ distance in $O(m)$ time.

While the significance of our work does not depend on it, the hypothesized compilation process is intriguing because ACAs of dimension three or less can (in principle) be physically realized in hardware. Furthermore, this can be done in such a way that the abstract dimensions of space and time in the ACA (and of all models that have been $O(1)$ reduced to it) are coextensive with physical dimensions of space and time:

$$ACA\left(\mathbb{Z}^3\right) \leq_1 U, \tag{5}$$

where U is the physical universe. This is the basis for the claim that our artificial chemistry is a bespoke physics and that kinematic automata built with it are tantamount to physical machines.[5]

3 Kolmogorov complexity is correct only if the replicator contains no *untranslated information*. Indeed, a pure template replicator (e.g., [31]) might have very high Kolmogorov complexity, yet its normalized complexity is zero, since it is composed entirely of information that (apart from being copied) is never used.

4 Because objects are embedded in space and objects' states are updated *asynchronously* by operations that depend only on the states of objects in their neighborhoods, ACAs are the natural compilation target.

5 A *bespoke physics* is a task-specific software interface that defines *basic units* embedded in space and a set of *dynamical laws* that describe how the units move and interact. These dynamic laws are subject to *meta-laws*, including *indefinite scalability* and *global nondeterminism*. It is further stipulated that a bespoke physics must be implementable (at least in principle) in the physical universe. See Ackley [1, pp. 3–4].

Given a replicator R defined on top of a hierarchy of models reducible to U by $O(1)$ reduction $R \leq_1 M_N \leq_1 \cdots \leq_1 M_1 \leq_1 U$, the ratio of the contingent and noncontingent complexities of replicator R becomes

$$\frac{K(R \mid M_N)}{K(M_N \mid M_{N-1}) + \cdots + K(M_2 \mid M_1) + K(M_1)}. \tag{6}$$

The meaning of this quantity, which will henceforward be termed a replicator's *normalized complexity*, is best illustrated by an example. Codd [4] was able to significantly simplify von Neumann's replicator and its host CA. Although the contingent complexity of the Codd replicator is significantly less than that of the von Neumann replicator, we speculate that (were they calculated) their normalized complexities would be closer in value.

Langton [20] defined a much simpler *loop* replicator L_L on top of the Codd CA substrate. Its contingent complexity, $K(L_L \mid CA_C)$, is much less than that of the Codd replicator, $K(R_C \mid CA_C)$. Nehaniv [25] showed how the Codd CA substrate could be $O(1)$ reduced to an ACA and demonstrated the Langton *loop* running on top of the Codd CA running on top of this ACA. These results allow us to compare the normalized complexities of the Codd replicator and the Langton loop defined with respect to the same hierarchy of computational models:

$$\frac{K(L_L \mid CA_C)}{K(CA_C \mid ACA_N) + K(ACA_N)} < \frac{K(R_C \mid CA_C)}{K(CA_C \mid ACA_N) + K(ACA_N)}. \tag{7}$$

Hutton's work on *artificial cells* provides a second example [15]. In Hutton's virtual world, physical law takes the form of an artificial chemistry defined by a set of 34 graph rewrite rules. Hutton's artificial organism is a cell-like configuration of atoms $C_H + P_1$ containing a small (nonfunctional) information payload P_1. Significantly, Hutton demonstrated that both the cell and its payload are replicated by the reaction rules of the artificial chemistry. Because the payload P_1 is untranslated, the contingent complexity of Hutton's cell is $K(C_H \mid AC_{34})$.

Hutton subsequently extended AC_{34} by adding six rules for translating the payload P_1 into an *enzyme* E_1 capable of catalyzing an arbitrary reaction and used this enzyme to replace one of the graph rewrite rules, R_1. In doing so, the (nonfunctional) information payload becomes a (functional) partial genome, and some part of the system's complexity moves from the noncontingent to the contingent category. However, this exchange is insufficient to offset the increase in noncontingent complexity that results from the addition of the six rules. Consequently,

$$\frac{K(E_1 \mid P_1, AC_{40} - R_1) + K(C_H + P_1 \mid AC_{40} - R_1)}{K(AC_{40} - R_1)} < \frac{K(C_H \mid AC_{34})}{K(AC_{34})}, \tag{8}$$

where $K(E_1 \mid P_1, AC_{40} - R_1)$ is zero, since P_1 encodes E_1 using a process defined by the artificial chemistry $AC_{40} - R_1$.

3 An Object-Oriented Combinator Chemistry

Superficially, there is a similarity between the sequences of instructions that constitute a machine-language program and the sequences of nucleotides and amino acids that constitute the biologically important family of molecules known as *biopolymers*. It is tempting to view all of these sequences as "programs," broadly construed. However, machine language programs and biopolymers differ in (at least) one significant way, and that is the number of elementary building blocks from which they are constructed. The nucleotides that make up DNA and RNA are only of four types; the amino acids that make up polypeptides are only of twenty; and while bits might (like nucleotides)

serve as representational units, they cannot (like amino acids) act as functional units; this role can only be played by *instructions*.

While the instruction set of a simple random access stored program (RASP) computer can be quite small, the number of distinct operands that (in effect) modify the instructions is a function of the word size, and is therefore (at a minimum) in the thousands. Although they play many roles in machine language programs, operands are generally *addresses*. Some point to data, while others point to the targets of branch and function-call instructions, which is to say, other instructions. The fact that addresses play both roles is a consequence of the program–data equivalence that makes self-replication by *reflection* trivial for machine language programs in RASPs.

Sadly, even though they make life easy, RASPs (unlike ACAs) are not indefinitely scalable, because there is no $O(1)$ reduction of a RASP to the physical universe. Happily, self-replication can be achieved by other means *and* with greater physical realism. However, we must abandon the thing that makes self-replication in RASPs so easy, namely, the ability to randomly access the instructions of a program stored in memory using thousands of distinct addresses. For inspiration, we turn to the living cell.

DNA and RNA are copiable, transcribable, and translatable descriptions of polypeptides. DNA is (for the most part) chemically inert, while polypeptides are chemically active. Polypeptides cannot serve as representations of themselves (or for that matter of anything at all), because their enzymatic functions render this impossible. Information flows in one direction only. Watson and Crick [41] thought this idea so important that they called it "the fundamental dogma of molecular biology." It is the antithesis of program–data equivalence.

In this article we show how programs in a visual programming language can be (1) used to define the behaviors of *actors* [12] reified in a virtual world, and (2) compiled into sequences of *combinators* of a small number of fixed types. Unlike the instructions of machine-language programs, combinators possess no additional operands. Where machine language programs would use recursion or iteration, the combinators we define employ *nondeterminism*. The fact that combinators are both used to define behaviors and reified as actors in the virtual world is the key to constructing a self-replicating system with *semantic closure* [26], a system where both programs (phenome) and inert descriptions of programs (genome) are represented by sequences of combinators of a small number of fixed types.

There are three types of actors: *objects*, *methods*, and *combinators*. Objects and methods are like objects and methods in object-oriented programming. More specifically, objects are containers for actors, methods are programs that govern actors' behaviors, and combinators are the building blocks used to construct methods (see Figure 2). Like amino acids, which can be composed to form polypeptides, *primitive* combinators can be composed to form *composite* combinators. A method is just a composite combinator that has been repackaged, or *unquoted*. Prior to unquoting, combinators do not manifest behaviors, so unquoting might correspond (in this analogy) to the folding of a polypeptide chain into a protein.

Objects are *multisets* of actors. They are of four immutable types constructed using $\{\}_0$, $\{\}_1$, $\{\}_2$, and $\{\}_3$. For example, $\{x, y, z\}_2$ is an object of type two that contains three actors, x, y, z. Combinators are composed with (>=>), and quoted and unquoted using $()^-$ and $()^+$. More formally, datatypes *Actor*, *Combinator*, and *Method* can be recursively defined as follows:

$$Actor = Combinator \mid Method \mid \{Actor\}_0 \mid \cdots \mid \{Actor\}_3 \tag{9}$$

$$Combinator = c_1 \mid \cdots \mid c_N \mid Combinator >=> Combinator \mid Method^- \tag{10}$$

$$Method = Combinator^+, \tag{11}$$

where $c_1, \cdots, c_N$ are primitive combinators. Primitive combinators and empty objects have unit mass. The mass of a composite combinator is the sum of the masses of the combinators of which it is composed. The mass of an object is the sum of its own mass and the masses of the actors it contains. Since actors can neither be created nor be destroyed, mass is conserved.

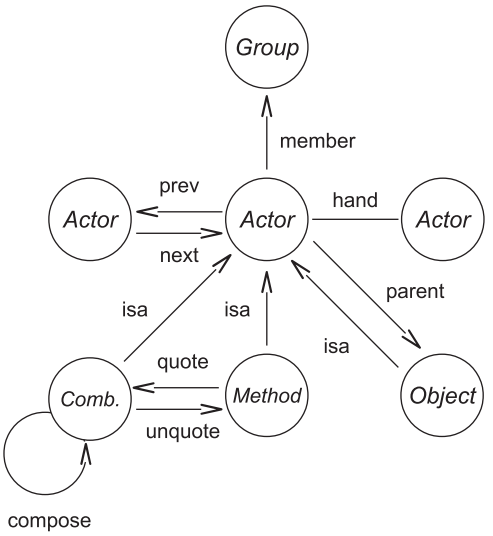

Figure 2. Actors are of three types. *Objects* are containers for actors, *methods* are programs that govern actors' behaviors, and *combinators* are the building blocks used to construct methods. Actors can also belong to *groups* and can form directed and undirected bonds with other actors.

Actors are *reified* by assigning them positions in a 2D virtual world. Computations progress when actors interact with other actors in their Moore neighborhoods by running methods. All actors are subject to *diffusion*. An actor's diffusion constant decreases inversely with its mass. This reflects the real cost of data transport in the (notional) $ACA(\mathbb{Z}^2 \times \mathbb{N})$ substrate. Multiple actors can reside at a single site, but diffusion never moves an actor to an adjacent occupied site if there is an adjacent empty site.

As with *membranes* in [27], objects can be nested to any level of depth.[6] The object that contains an actor (with no intervening objects) is termed the actor's *parent*. An actor with no parent is a *root*. Root actors (or actors with the same parent) can associate with one another by means of *groups* and *bonds*. Association is useful because it allows working sets of actors to be constructed and the elements of these working sets to be addressed in different ways.

The first way in which actors can associate is as members of a *group*. All actors belong to exactly one group, and this group can contain a single actor. For this reason, the group relation is an *equivalence relation* on the set of actors. A group of root actors is said to be *embedded*. All of the actors in an embedded group diffuse as a unit, and all methods run by actors in an embedded group (or contained inside such actors) share a finite time resource in a zero-sum fashion. Complex computations formulated in terms of large numbers of actors running methods inside a single object or group will therefore be correspondingly slow. Furthermore, because of its large net mass, the object or group that contains them will also be correspondingly immobile.

The second way in which actors can associate is by *bonding*. Bonds are short relative addresses that are automatically updated as the actors they link undergo diffusion. Because bonds are short (L_1 distance less than or equal to two), they restrict the diffusion of the actors that possess them. Undirected bonds are defined by the *hand* relation H, which is a symmetric relation on the set of actors, that is, $H(x,y) = H(y,x)$. Directed bonds are defined by the *previous* and *next* relations, P and N, which are inverse relations on the set of actors, that is, $P(x,y) = N(y,x)$. An actor can possess at most one bond of each type.

If the types of combinators and methods were defined by the sequences of primitive combinators of which they are composed, then determining type equivalence would be relatively expensive.

6 In object-oriented programming, this is termed *object composition*.

For this reason, we chose instead to define type using a simple recursive hash function that assigns combinators with distinct multisets of components to distinct types: The hash value of a composite combinator is defined as the product of the hash values of its components; primitive combinators have hash values equal to prime numbers.[7] Type equivalence for methods is defined in the same way, the types of combinators and methods being distinct due to the use of different constructors. Although this hash function is (clearly) not collision-free, it is quite good and it has an extremely useful property, namely, that composite combinators can be broken down (literally decomposed) into their primitive combinators by prime factorization. Because primitive combinators are of relatively few types, this operation is not prohibitively expensive.[8]

Apart from composition, containment, groups, and bonds, there is no other mutable persistent state associated with actors. In particular, there are no integer registers. Primitive combinators exist for addressing individual actors or sets of actors, using most of these relations. These and other primitive combinators for modifying actors' persistent states will be described later.

4 Nondeterministic Comprehensions

Monads have been described by Hughes [14] as a standard interface to libraries of "program fragments." More specifically, they are an abstract data type that allows programmers to define rules for composing "functions" that deviate from mathematically pure functions in prescribed ways [23]. Monads are intimately related to expressions in *set-builder notation*, which in functional programming are called *comprehensions*. Although many programming languages provide *list comprehensions*, set-builder notation can (in fact) be used with any monad [39]. Of particular relevance here, *nondeterministic comprehensions* define *superpositions* (not lists), and can be used when searching for exactly one set of variable assignments (among many) satisfying a set of constraints, e.g., an unspecified Pythagorean triple.[9] This makes them ideal left-hand sides for rules in a nondeterministic rule-based system, a description broad enough to encompass many artificial chemistries, including OOCC.

Sets can be converted to superpositions using McCarthy's [22] nondeterministic choice operator, *amb*. This operator has *type signature* $\{a\} \rightarrow \langle a \rangle$ and can be defined as follows:

$$\text{amb}\,\{\} = \langle\rangle \tag{12}$$

$$\text{amb}\,\{x, y\ldots\} = \langle x, y\ldots\rangle. \tag{13}$$

When *amb* is applied to a nonempty set, the branch of the nondeterministic computation that called *amb* forks, yielding a new branch for each element in the set. Conversely, empty sets cause branches to fail. When a branch fails, the underlying deterministic implementation backtracks. If all branches fail, the nondeterministic computation fails. The advantage of the monad interface is that programs built using it can exhibit complex behaviors (e.g., backtracking) simply by virtue of the type they return (e.g., superpositions).

The monad interface is defined by two operations called *unit* and *bind*. *Unit* transforms ordinary values into *monadic values*, for example, $\text{unit}_A\, x = \langle x \rangle$, where A is the superposition monad. Functions that take ordinary values and return monadic values are termed *Kleisli morphisms*. *Bind*, the infix operator (≫=), allows Kleisli morphisms to be applied to monadic values. This permits Kleisli morphisms to be chained; the output of one provides the input to the next.

7 We could instead use nested objects to label combinators so that they could be compared. This would be like using codons constructed from nucleotides to label amino acids in transfer RNAs.

8 This is analogous to the function in the cell that is performed by the molecular assemblies called *proteasomes* and the organelles called *lysosomes*.

9 Kiselyov et al. [16] call this *don't-care nondeterminism*.

Monads and comprehensions are intimately related. By way of illustration, consider the following nondeterministic comprehension that fails if n is prime and returns a factor of n if n is composite:

$$\langle x \mid x \in \langle 1 \,..\, n-1 \rangle,\ y \in \langle 1 \,..\, x \rangle,\ xy = n \rangle. \tag{14}$$

Wadler [39] showed that notation like the above is syntactic sugar for more primitive monadic expressions and described a process for translating the former into the latter. Comprehension *guards* (e.g., $xy = n$) are translated using the function

$$\text{guard}_M\ \ \mathit{True} = \text{unit}_M() \tag{15}$$

$$\text{guard}_M\ \ \mathit{False} = \text{zero}_M, \tag{16}$$

where M is the monad and $()$ is *void*. Because zero_A is $\langle\,\rangle$, if guard_A is applied to *False*, the branch of the computation that called guard_A fails. Conversely, if guard_A is applied to *True*, the branch continues. Using this device, the primality comprehension can be desugared as follows

$$\begin{aligned}\lambda n \rightarrow ((\text{unit}_A \cdot (-1))\ n \overset{A}{>\!\!>\!\!=} \text{amb} \cdot \iota \overset{A}{>\!\!>\!\!=} \lambda x \rightarrow ((\text{amb} \cdot \iota)\ x \overset{A}{>\!\!>\!\!=} \text{unit}_A \cdot (\times x) \\ \overset{A}{>\!\!>\!\!=} \text{unit}_A \cdot (=n) \overset{A}{>\!\!>\!\!=} \text{guard}_A \overset{A}{>\!\!>\!\!=} \lambda z \rightarrow \text{unit}_A\ x)),\end{aligned} \tag{17}$$

where (ιx) equals $\{1\ ..\ x\}$ and the value of z is ignored.

5 From Comprehensions to Dataflow Graphs

Our goal is to create programs composed solely of combinators. To maximize composability, these combinators should have a single type signature, yet the desugared comprehension above contains functions with many different types. However, we observe that if sets are used to represent sets, singleton sets are used to represent scalars, and nonempty and empty sets are used to represent *True* and *False*, then the type signatures

$$\rightarrow \boxed{f'} \rightarrow \quad :: \quad \{\mathbb{Z}\} \rightarrow \langle\{\mathbb{Z}\}\rangle, \tag{18}$$

$$\rightrightarrows \boxed{g'} \rightarrow \quad :: \quad \{\mathbb{Z}\} \rightarrow \{\mathbb{Z}\} \rightarrow \langle\{\mathbb{Z}\}\rangle \tag{19}$$

are general enough to describe all functions in the desugared comprehension. To prove this, we first observe that *amb*, with type signature $\{\mathbb{Z}\} \rightarrow \langle\mathbb{Z}\rangle$, can be lifted to a function amb', with type signature $\{\mathbb{Z}\} \rightarrow \langle\{\mathbb{Z}\}\rangle$, as follows:

$$\text{amb}'\{\} = \langle\rangle, \tag{20}$$

$$\text{amb}'\{x, y \ldots\} = \langle\{x\}, \{y\} \ldots\rangle. \tag{21}$$

We then devise a way to lift functions like ι with type signature $\mathbb{Z} \rightarrow \{\mathbb{Z}\}$. This is accomplished using the bind operator ($>\!\!>\!\!=_S$) for the set monad S. The bind operator behaves like

$$\{x, y \ldots\} \overset{S}{>\!\!>\!\!=} f = f\,x \cup f\,y \cup \cdots \tag{22}$$

and can be defined as follows:

$$(\overset{S}{>\!>\!=} f) = \text{join}_S \cdot (\text{map}_S f), \tag{23}$$

where *join*$_S$ is right fold of $(\cup)$, and

$$\text{map}_S f\{x, y\ldots\} = \{f\,x, f\,y\ldots\}. \tag{24}$$

Bind can then be used with *unit*$_A$ to lift ι into a function

$$\iota' = \text{unit}_A \cdot (\overset{S}{>\!>\!=} \iota) \tag{25}$$

with type signature matching f' as demonstrated below:

$$\iota'\{x, y\ldots\} = \langle \iota\, x \cup \iota\, y \cup \ldots \rangle. \tag{26}$$

Next we define two functions with type signatures matching f' to replace *guard*. The first causes a computation to fail when its argument is empty while the second does the opposite:

$$\text{some}'\{\} = \langle\rangle \tag{27}$$

$$\text{some}'\{x, y\ldots\} = \langle\{x, y\ldots\}\rangle \tag{28}$$

$$\text{none}'\{\} = \langle\{\}\rangle \tag{29}$$

$$\text{none}'\{x, y\ldots\} = \langle\rangle. \tag{30}$$

Finally, the desugared comprehension contains functions like (-1), $(\times)$, and $(=)$ that map scalars to scalars, yet we need functions that map sets to superpositions of sets. Fortunately, lifted forms for these functions with type signatures matching f' or g' are easily defined:

$$\text{pred}' = \text{unit}_A \cdot (\text{map}_S(-1)) \tag{31}$$

$$\text{times}'\,x'\,y' = \text{unit}_A\{x \times y \mid x \in x', y \in y'\} \tag{32}$$

$$\text{equals}'\,x'\,y' = \text{unit}_A\{x \mid x \in x', y \in y', x = y\}, \tag{33}$$

where x' and y' are of type $\{\mathbb{Z}\}$. Using these lifted functions and those defined previously, the nondeterministic comprehension for deciding primality can be further translated as follows:

$$\begin{aligned}\lambda n' \to (\text{pred}'\ n' \overset{A}{>\!>\!=}\ \iota'\ \overset{A}{>\!>\!=}\ \text{amb}'\ \overset{A}{>\!>\!=}\ \lambda x' \to (\iota'\ x'\ \overset{A}{>\!>\!=}\ \text{amb}'\ \overset{A}{>\!>\!=}\\ \text{times}'\ x'\ \overset{A}{>\!>\!=}\ \text{equals}'\ n'\ \overset{A}{>\!>\!=}\ \text{some}')),\end{aligned} \tag{34}$$

where n' is of type $\{\mathbb{Z}\}$. This was a lot of work, but we have reaped a tangible benefit, namely, all functions now have uniform type. This simplification allows nondeterministic comprehensions to be visualized as *dataflow graphs* with well-defined semantics. In Figure 3 (top), boxes with one input have type signatures matching f', and boxes with two inputs have type signatures matching g'. Arrows connecting pairs of boxes are instances of $(\gg\!=_A)$. Junctions correspond to values of common subexpressions bound to variable names introduced by λ-expressions. Lastly, (A) is *amb*′ and (S) is *some*′.

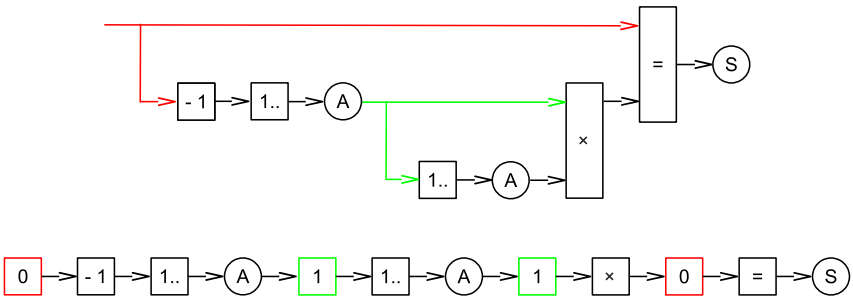

Figure 3. Nondeterministic dataflow graph for deciding primality (top). Dataflow graph compiled into a sequence of nondeterministic combinators (bottom).

6 From Dataflow Graphs to Combinators

The use of dataflow graphs as a visual programming language has a long history beginning with their introduction by Sutherland [34]. As programs, dataflow graphs have several disadvantages when compared to textual representations, including the need for specialized drawing tools to construct them, the complexity of the layout problem that construction presents, and the lack of good persistent data formats. Yet they also have compelling advantages. For example, because evaluation order is not completely specified in a dataflow graph, it can be optimized and (potentially) even parallelized by subsequent compilation processes. Most importantly, they make it possible to see at a glance where values are computed and the full extent of their use throughout a computation. For these reasons, we use dataflow graphs as a visual programming language in the remainder of this article, but only after (1) observing that the computations they specify can be represented by programs written in a small Haskell subset, and (2) describing a process for compiling programs in that subset into sequences of primitive combinators.

The programming language Haskell [28] includes a syntactic construct called *do* that simplifies the problem of defining monadic computations. The process of translating programs written using *do* notation into more primitive programs written using λ-expressions and (>>=) is straightforward. It is possible to define a grammar for a small subset of Haskell that uses *do* notation to represent computations specified by dataflow graphs:

$$d \;::=\; \text{`}\lambda\text{'}\ x\ \text{`}\rightarrow \text{do}\ \{\text{'}\ (l\ \text{`;'})^*\ \ l\ \text{`}\}\text{'}, \tag{35}$$

$$x \;::=\; \text{`}a\text{'} \mid \ldots \mid \text{`}z\text{'}, \tag{36}$$

$$l \;::=\; x\ \text{`}\leftarrow\text{'}s \mid s, \tag{37}$$

$$s \;::=\; \left(f'\ \text{`} \overset{A}{=\!\!<<} \text{'} \mid g'\ x\ \text{`} \overset{A}{=\!\!<<} \text{'}\right)^*\ \left(f'x \mid g'x\,x\right), \tag{38}$$

where (=<<) is just (>>=) with its arguments reversed. In effect, the grammar defines an embedded domain-specific language with semantics determined by the Haskell host language. The dataflow graph for deciding primality (Figure 3) can be represented as a sentence in this language as follows:

$$\begin{aligned}\lambda n' \rightarrow \text{do}\ \{\ & x' \leftarrow \text{amb}' \overset{A}{=\!\!<<} \iota' \overset{A}{=\!\!<<} \text{pred}'\ n'\ ; \\ & \text{some}' \overset{A}{=\!\!<<} \text{equals}'\ n' \overset{A}{=\!\!<<} \text{times}'\ x' \overset{A}{=\!\!<<} \text{amb}' \overset{A}{=\!\!<<} \iota'\ x' \\ & \}.\end{aligned} \tag{39}$$

The most significant difference between the above program and the dataflow graph (apart from the use of names to represent values of common subexpressions) is that data flows right-to-left within lines of the program, but left-to-right within paths of the graph.

One might assume that evaluation of dataflow graphs containing junctions would require an interpreter with the ability to create and apply anonymous functions, or *closures* [19]. These would contain the environments needed to look up the values bound to variable names introduced by λ-expressions. Happily, this turns out to be unnecessary. We now show how dataflow graphs can be evaluated by a stack machine and define a set of combinators that can be used to construct stack machine programs.[10]

In general, combinators apply functions to one (or two) values of type $\{\mathbb{Z}\}$ popped from the front of the stack and then push a result of type $\{\mathbb{Z}\}$ back onto the stack. Since dataflow graphs are nondeterministic, the stack machine is also. This means that each combinator f'' transforms a stack of sets into a superposition of stacks of sets:

$$\rightarrow \boxed{f''} \rightarrow \quad :: \quad [\{\mathbb{Z}\}] \rightarrow \langle[\{\mathbb{Z}\}]\rangle. \tag{40}$$

Unary operations f' can be converted to combinators of type f'' as follows:

$$f''(x' : s'') = \mathrm{map}_{\mathcal{A}}\,(: s'')\big(f'\,x'\big), \tag{41}$$

where stack s'' is of type $[\{\mathbb{Z}\}]$, $map_{\mathcal{A}}$ maps functions over superpositions, and $(: s'')$ is the function that pushes sets onto the front of s''. Note that f'' does not change the length of the stack; it consumes one value and leaves one value behind. Binary operators g' can also be converted to combinators of type f'' as follows:

$$g''(x' : y' : s'') = \mathrm{map}_{\mathcal{A}}\,(: s'')\,(g'\,x'\,y'). \tag{42}$$

Note that g'' decreases the length of the stack by one; it consumes two values and leaves one value behind. The combinator forms of *some*′ and *none*′ are slightly different; they do not push a result onto the stack. Instead, they pop the stack when a nondeterministic computation has yielded a satisfactory intermediate result (whether that is something or nothing) and fail otherwise:

$$\mathrm{some}''(x' : s'') = \begin{cases} \langle\rangle & \text{if } x' = \{\} \\ \mathrm{unit}_{\mathcal{A}}\,s'' & \text{otherwise} \end{cases} \tag{43}$$

$$\mathrm{none}''(x' : s'') = \begin{cases} \mathrm{unit}_{\mathcal{A}}\,s'' & \text{if } x' = \{\} \\ \langle\rangle & \text{otherwise.} \end{cases} \tag{44}$$

A function can be applied to a value within the stack by first pushing a copy of the value onto the top of the stack and then applying the function to the copy. This preserves the value within the stack for future use and eliminates the need for closures.

Accordingly, we define a set of combinators that copy and push values located at different positions within the stack:

$$x''_k(s'') = \mathrm{unit}_{\mathcal{A}}\,((s'' \mathbin{!!} (n - k)) : s''), \tag{45}$$

10 Other strategies are possible. For example, Kleisli morphisms are instances of an abstract data type more general than monads, called *arrows* [14]. Using arrow combinators, dataflow graphs can be directly represented as point-free expressions. Although these point-free expressions can be evaluated without further translation, they include binary combinators, for example, (&&&), that are non-associative. It follows that, had this strategy been adopted, methods would be represented by binary trees instead of unparenthesized sequences, and translation and replication processes implemented by computational ribosomes and replisomes would be correspondingly more complicated.

where $k \in [0\,..\,9]$, (!!) returns the element of a list with a given index, and n is the length of s''. With this last puzzle piece in place, we can finally do what we set out to do, namely, compile the comprehension into a sequence of combinators that can determine primality by nondeterministically transforming a compact and reference-free abstract machine state. Parsed representations of dataflow graphs are compiled using a function φ defined as follows:

$$\varphi\,(l_1\ \text{`;'}\ l_2) \;=\; \varphi\ l_1 \overset{A}{>\!\!=\!\!>} \varphi\ l_2 \tag{46}$$

$$\varphi\,(x\ \text{`}\leftarrow\text{'}\ s) \;=\; \varphi\ s \tag{47}$$

$$\varphi\,(s_2\ \text{`}\overset{A}{=\!\!<<}\text{'}\ s_1) \;=\; \varphi\ s_1 \overset{A}{>\!\!=\!\!>} \varphi\ s_2 \tag{48}$$

$$\varphi\ f' \;=\; f'', \tag{49}$$

where $(>\!\!=\!\!>)$ is the compose operation $f >\!\!=\!\!> g = (>\!\!>\!\!= g) \cdot f$ for Kleisli morphisms. Combinator x_0'' is associated with the name introduced by the initial binding ('λ' x_0 ' → do'), and combinators x_k'' with increasing indices are associated with names introduced by successive bindings (x ' ← ' s) in the parsed representation of the dataflow graph:

$$a = (x_0,\ \mathrm{x}_0'') \;:\; [(x,\ \mathrm{x}_k'') \mid (x\ \text{`}\leftarrow\text{'}\ s) \in [l_1\,..\,l_N] \mid k \in [1..9]], \tag{50}$$

where $[l_1\,..\,l_N]$ are the lines of the parsed representation. Function applications are compiled using a helper function σ, which, given a name, returns the combinator that will copy and push the value associated with that name onto the stack at run time:

$$\varphi\,(f'\ x_1) \;=\; \sigma\ x_1 \overset{A}{>\!\!=\!\!>} f'' \tag{51}$$

$$\varphi\,(g'\ x_1) \;=\; \sigma\ x_1 \overset{A}{>\!\!=\!\!>} g'' \tag{52}$$

$$\varphi\,(g'\ x_1\ x_2) \;=\; \sigma\ x_2 \overset{A}{>\!\!=\!\!>} \sigma\ x_1 \overset{A}{>\!\!=\!\!>} g'', \tag{53}$$

where $\sigma\ x = \mathrm{x}_k''$ for $(x, \mathrm{x}_k'') \in a$. Compiling the parsed representation of the dataflow graph for deciding primality with φ yields a point-free expression with no parentheses:

$$\begin{aligned}\mathrm{x}_0'' \overset{A}{>\!\!=\!\!>} \mathrm{pred}'' \overset{A}{>\!\!=\!\!>} \iota'' \overset{A}{>\!\!=\!\!>} \mathrm{amb}'' \overset{A}{>\!\!=\!\!>} \mathrm{x}_1'' \overset{A}{>\!\!=\!\!>} \iota'' \overset{A}{>\!\!=\!\!>} \mathrm{amb}''\\ \overset{A}{>\!\!=\!\!>} \mathrm{x}_1'' \overset{A}{>\!\!=\!\!>} \mathrm{times}'' \overset{A}{>\!\!=\!\!>} \mathrm{x}_0'' \overset{A}{>\!\!=\!\!>} \mathrm{equals}'' \overset{A}{>\!\!=\!\!>} \mathrm{some}'',\end{aligned} \tag{54}$$

the "polypeptide" of the article's title. In this expression, x_0'' is used to copy and push n' and x_1'' is used to copy and push x'. In Figure 3 (bottom) boxes are functions with type signatures matching f''. Arrows connecting pairs of boxes are instances of $(>\!\!=\!\!>_A)$. Lastly, Ⓐ is *amb″* and Ⓢ is *some″*.

Figure 4 illustrates the nondeterministic evaluation process. Although all combinators transform a stack of sets of integers into a superposition of stacks of sets of integers, only the *amb″* combinator introduces nondeterminism, because only the *amb″* combinator returns non-singleton superpositions. On the left, the stack initially contains just the singleton set {4}. The values of the shared subexpressions n and x in the comprehension are copied and pushed by the x_0'' and x_1'' combinators. These are drawn as 0 and 1 in the figure. The computation nondeterministically branches when it reaches *amb″* combinators, drawn as Ⓐ in the figure, which return non-singleton superpositions. Finally, the guard expression containing the *some″* combinator, drawn as Ⓢ in the figure, causes all branches of the computation except the one corresponding to $2 \times 2 = 4$ to fail.

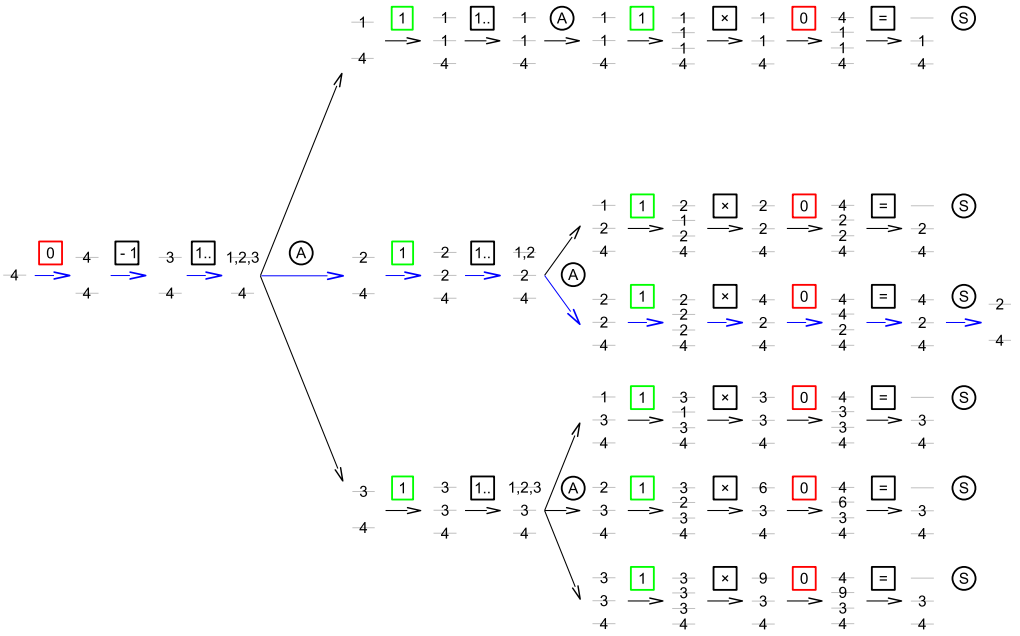

Figure 4. Evaluation of $\langle x \mid x \in \langle 1 \,..\, n-1 \rangle, y \in \langle 1 \,..\, x \rangle, xy = n \rangle$ by a nondeterministic stack machine. The computation begins at the left with the stack containing the singleton set {4} and nondeterministically forks when it reaches *amb"* combinators. Only one branch succeeds, returning the factor 2.

7 Reified Actor Comprehensions

The last two sections of the article demonstrated that: (1) nondeterministic comprehensions can be represented as dataflow graphs, and (2) dataflow graphs can be compiled into sequences of combinators that evaluate comprehensions by transforming the state of an abstract machine. In this section we describe a visual programming language for defining behaviors manifested by reified actors in a virtual world. To accomplish this, nondeterminism must be combined with additional effects to construct a monad more general than A, which we call R (for *reified actor*). In addition to representing superpositions, monad R provides mutation of a threaded global state and data logging so that methods composed of combinators can report the time they consume.[11] Since the structure of monadic programs is unchanged by the choice of monad (one of the advantages of the abstraction), all results from prior sections, for example, the process of compiling data flow graphs into combinators, apply. The boxes of dataflow graphs with one and two inputs now have types

$$\rightarrow \boxed{f'} \rightarrow \quad :: \quad \{Actor\} \rightarrow \langle \{Actor\} \rangle_R, \tag{55}$$

$$\rightrightarrows \boxed{g'} \rightarrow \quad :: \quad \{Actor\} \rightarrow \{Actor\} \rightarrow \langle \{Actor\} \rangle_R, \tag{56}$$

11 *Monad transformers* are a device for combining monadic effects [21]. In the present work, nondeterminism, threaded global state, and data logging are combined with mutation into a single monad by applying the *AmbT*, *ReaderT*, and *WriterT* monad transformers to the *IO* monad.

Table 1. Unary generators.

Name	Abbreviation	Definition
hands	|	actor sharing hand with *x*
prevs	<	actor with directed bond to *x*
nexts	>	actor with directed bond from *x*
bonds	$	union of hands, nexts, and prevs
neighbors	#	actors in neighborhood of *x*
parents	^	actor that contains *x*
contents	@	actors contained in *x*
members	&	members of group of *x*
others	+	other members of group of *x*

where $\langle\rangle_R$ is the type constructor of monad *R*. Arrows connecting boxes are instances of $(>\!\!>\!\!=_R)$. Dataflow graphs are compiled into combinators of type

$$\rightarrow \boxed{f''} \rightarrow \quad :: \quad [\{Actor\}] \rightarrow \langle [\{Actor\}] \rangle_R \tag{57}$$

and are composed with $(>\!\!=\!\!>_R)$.

Boxes and combinators can be divided into the categories: *generators*, *guards*, *relations*, and *actions*. Generators are unary operators that characterize sets of actors using the devices of groups, containment, bonds, and neighborhood (Table 1). They can be composed to address different sets. For example, an actor's siblings can all be addressed using the subgraph $\boxed{\hat{}} \rightarrow \boxed{@}$. Generators can also be composed with guards (Table 2). This can be used either to address single actors or to specify preconditions for actions. For example, the subgraph $\boxed{\hat{}} \rightarrow \boxed{@} \rightarrow (\text{A})$ addresses a single sibling, while the subgraph $\boxed{\#} \rightarrow (\text{N})$ fails if the actor has a neighbor.

Relations exist for testing equality and type equivalence (Table 3). They are binary operators and are generally applied to singleton sets in combination with guards to specify preconditions for actions. When applied to non-singleton sets, the equality operator and its negation compute set intersection and difference. The type equivalence operator and its negation also have natural generalizations. In addition to specifying preconditions for actions, relations can also be combined with generators to characterize sets.

Table 2. Unary guards.

Name	Abbreviation	Definition
amb	A	nondeterministic choice
some	S	Fail if empty.
none	N	Fail if nonempty.

Table 3. Binary relations.

Name	Abbreviation	Definition
same	=	set intersection
different	!=	set difference
union	++	set union
similar	~	all *x* type equivalent to some *y*
dissimilar	!~	all *x* type equivalent to no *y*

Actions for modifying actors' persistent states are the final category of boxes in dataflow graphs. Actions are rendered as gray boxes and are executed only after all non-actions have been evaluated and only if no guard has failed. All actions are reversible, but the masses and types of primitive combinators and empty objects are immutable. The full set of unary and binary actions is shown in Tables 4 and 5.

Although dataflow graphs make data dependences explicit, unlike *do* syntax they do not completely determine the order in which the primitive combinators of the compiled program are executed. Because generators, guards, and relations are purely functional, the order in which subsequences composed solely of combinators from these categories are evaluated does not matter. In contrast, the result of a sequence of actions that modify actors' persistent states generally depends on the order in which the component actions are executed. It follows that the visual programming language must include a device for specifying the order in which actions should be executed where it is not determined by data dependences.

Where data dependences determine order of execution, this order is followed. Where it would otherwise be underdetermined, two devices are introduced to specify execution order. First, all actions return their first (or only) argument if they succeed. This allows one action to provide the input to a second and (when employed) introduces a data dependence that determines execution order. Second, execution order can be explicitly specified using *control lines* [34]. These are rendered

Table 4. Unary actions.

Name	Abbreviation	Definition
drop	!|	Delete hand of *x*.
deletePrev	!<	Delete directed bond to *x*.
deleteNext	!>	Delete directed bond from *x*.
remove	!^	Remove *x* from its parent object.
empty	!@	Remove actors contained in *x*.
quit	!+	Remove *x* from its group.
decompose	>!>	Reduce *x* >=> *y* to *x* and *y*.
quote	'	Suppress behavior.
unquote	!'	Express behavior.

Table 5. Binary actions.

Name	Abbreviation	Definition
grab	|	Create hand between *x* and *y*.
makePrev	<	Create directed bond from *y* to *x*.
makeNext	>	Create directed bond from *x* to *y*.
join	+	*x* joins group of *y*.
insert	@	Place *x* inside *y*.
compose	>=>	Replace *x* with *x* >=> *y*.

as dashed lines and are further distinguished from ordinary inputs and outputs by the fact that they enter and exit from the tops and bottoms of boxes and not the left or right sides.

In addition to nondeterminism and mutable threaded state, instances of monad *R* also possess a data logging ability that is used to instrument combinators so that methods composed of them can report the time they consume. Because the unit of time is one primitive operation of the abstract machine, most primitive combinators increase the logged time by one when they are run. Significantly, this occurs on all branches of the nondeterministic computation until a branch succeeds, so that the full cost of simulating nondeterminism on a (presumed) deterministic substrate by means of backtracking is accounted for. Two kinds of combinators increase the logged time by amounts other than one. Since the time required to compute set intersections and differences is the product of the sets' lengths, for binary relations the logged time is increased by this value instead (which equals one in the most common case of singleton sets). Finally, actions that change the position of an actor (e.g., *insert* and *join*) pay an additional time penalty proportional to the product of the actor's mass and the L_1 distance moved.

Ideally, the object-oriented combinator chemistry (OOCC) described in this article would be implemented on top of an $ACA(\mathbb{Z}^2 \times \mathbb{N})$ substrate so that competing self-replicating programs directly incur the costs of the physical resources they consume. The abstract computational resources of time and space would be $O(1)$ reducible to concrete physical resources of time and space in the host substrate. Actors in an embedded group might share a single processor or might jointly occupy a 2D area of fixed size that collects a fixed amount of light energy per unit time. The effect would be the same; the number of primitive abstract machine operations executed per unit time by the processor (or in the area) would be fixed.

For the time being, we implement the combinator chemistry as an event-driven simulation using a priority queue [11] and carefully account for the time consumed by actors in embedded groups. Event times are modeled as Poisson processes associated with embedded groups, and event rates are consistent with the aggregate consumption by actors in groups of finite time resources. Events are of two types. When a *diffusion event* is at the front of the queue, the position of the group in its neighborhood is randomly changed (as previously described). Afterwards, a new diffusion event associated with the same group is enqueued. The time of the new event is a sample from a distribution with density $f_D(t) = De^{-Dt/(ms)}/(ms)$, where m is mass, s is distance, and D is the ratio of the time needed to execute one primitive operation to the time needed to transport a unit mass a unit distance. As such, D defines the relative cost of computation and data transport in the $ACA(\mathbb{Z}^2 \times \mathbb{N})$ substrate.

When an *action event* is at the front of the queue, a method is chosen at random from among all actors of type method in the group. After the method is run, the time assigned to the new action event is a sample from a distribution with density $f_A(t) = e^{-t/c}/c$, where c is the time consumed by the method.

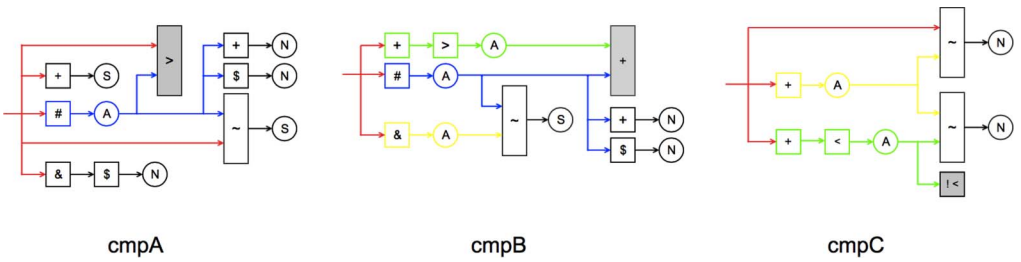

Figure 5. Three methods defining a composome.

8 Defining Behaviors Using Dataflow Graphs

In this section we illustrate the use of dataflow graphs to define behaviors by constructing a simple self-replicating entity called a *composome*. Composomes are quasi-stationary molecular assemblies that preserve *compositional information* [29]. As self-replicating entities, they possess very low normalized complexity, because they do not construct the parts of which they are composed, and individually, these parts are more complex than the composome itself. Nevertheless, a composome serves as a good first example, and we can construct one by defining a set of behaviors using dataflow graphs and reifying them as an embedded group of methods:

$$X = \{\text{cmpA},\ \text{cmpB},\ \text{cmpC}\}, \tag{58}$$

where cmpA, cmpB, and cmpC are the three dataflow graphs in Figure 5 reified as methods, and {} denotes an embedded group. The composome's first two methods run in the mother group (the group being copied), while its third runs in the daughter group (the copy). Because methods contained in different embedded groups run in parallel, they do not compete for cycles; this decreases the time required for self-replication.

- If cmpA is in a group with others ([+]→(S)) but no members of its group have bonds ([&]→[$]→(N)), then it finds an unbonded actor ([$]→(N)) in its neighborhood ([#]→(A)) similar to itself ([~]→(S)) with no others in its group ([+]→(N)) and creates a *next* bond with it using the [>] combinator; see Figure 6a.

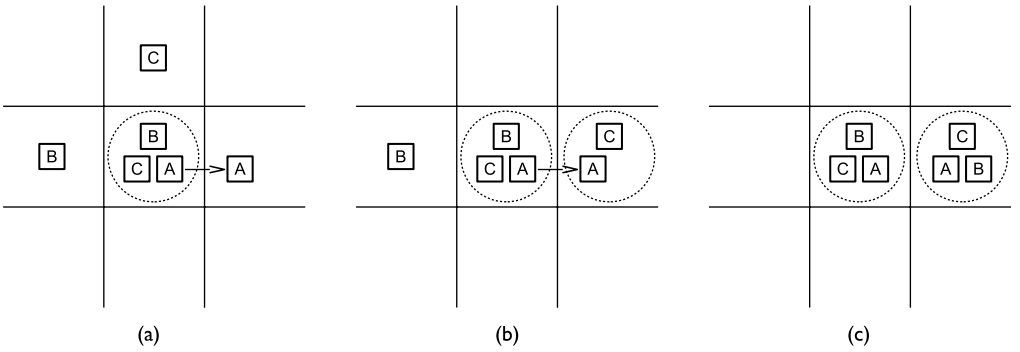

Figure 6. Self-replication by composome. (a) CmpA forms a *next* bond with another cmpA instance in its neighborhood. (b) CmpB finds a cmpC instance in its neighborhood and adds it to the daughter group. (c) CmpC (in the daughter group) deletes the bond joining the mother and daughter groups after cmpB is added.

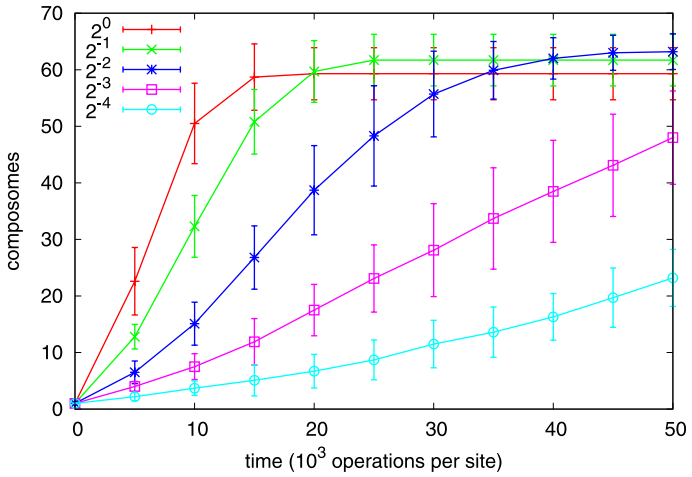

Figure 7. Average number of composomes (ten runs) as a function of time (10^3 operations per site) for diffusion constant $D = 2^0, \ldots, 2^{-4}$. Error bars show ±1 standard deviation.

- CmpB first uses [+]→[>]→(A) to verify that it is in the mother group. If it also has an unbounded ([$]→(N)) neighbor ([#]→(A)) similar to a member of its group ([&]→(A)→[~]→(S)), and if the neighbor is not already in a group ([+]→(N)), then it adds the neighbor to the daughter's group using the [+] combinator; see Figure 6b.
- CmpC first checks to see if it is in the daughter group. It does this by verifying that there is a group member with a *prev* bond ([+]→[<]→(A)); this group member can only be of type cmpA. It then verifies that there is also a group member ([+]→(A)) not similar to either the cmpA instance or itself ([~]→(N)). By a process of elimination, this group member must be of type cmpB. Since the daughter group contains the complete set of methods, it can delete the *prev* bond that joins the cmpA instances of the mother and daughter groups using the [!<] combinator; see Figure 6c.

When composome X is placed into the virtual world with a supply of the methods that compose it, the following reaction occurs:

$$X + \text{cmpA} + \text{cmpB} + \text{cmpC} \rightarrow 2\,X. \tag{59}$$

Because cmpB does not check to see whether the composome already possesses a method before adding it to the daughter group, the fraction of reactants converted to complete composomes (self-replication efficiency) will be significantly less than 100%.

The goals of the first experiment were to (1) verify that the composome successfully self-replicates, (2) characterize the composome's self-replication efficiency, and (3) investigate the effect of varying the diffusion constant D on the rate of composome replication. To achieve these goals, 100 instances of each of the three methods composing the composome (enough for 100 composomes to be constructed ideally) were reified as actors and distributed uniformly on a grid of size 32 × 32. The experiment consisted of five trials in which the value of the diffusion constant was varied. Each trial was repeated for ten runs, and each run ended when the time exceeded 50 × 10^3 operations per site. The results are shown in Figure 7. We observe that the self-replication efficiency is approximately 60% and that the replication rate increases with the diffusion constant, which is consistent with a process dominated by the cost of data transport.

9 Genes and Enzymes

Biological enzymes can be reified as chains of nucleotides or amino acids. The first can be read and copied but are spatially distributed and purely representational; the second are representationally opaque but compact and metabolically active. Dataflow graphs can be compiled into sequences of primitive combinators and reified in analogous ways: *genes* can be read and copied but do not manifest behaviors; *enzymes* manifest behaviors but cannot be read or copied. A *gene* is a spatially extended chain of actors of type combinator linked with *directed* bonds:

$$G_i = >_{j=1}^{|G_i|} c_i(j), \tag{60}$$

where $c_i(j)$ is combinator j of gene i, and $(>)$ are directed bonds. As in the genomes of living cells, sets of genes that are expressed together can be grouped together. A *plasmid* is a sequence of one or more genes joined with *undirected* bonds:

$$P = |_{i=1}^{|P|} >_{j=1}^{|G_i|} c_i(j), \tag{61}$$

where $(|)$ are undirected bonds. An additional undirected bond $c_{|P|}(|G_{|P|}|) \mid c_1(1)$ closes the chain. While plasmids are spatially distributed chains of multiple actors, *enzymes* are single actors of type method:

$$E_i = (>\!\!=\!\!> {}_{j=1}^{|G_i|}\, c_i(j))^+, \tag{62}$$

where $(>\!\!=\!\!>)$ is Kleisli composition and $()^+$ is the constructor for actors of type method. In addition to plasmids, composed of genes, a minimum self-replicating system might contain objects of three types. *Ribosomes* translate genes into enzymes, and *replisomes* copy plasmids. *Factories* are copiers of compositional information, namely, the sets of enzymes and objects that comprise ribosomes, replisomes, and factories themselves. A self-replicating system like this would possess *semantic closure* [26], because it would construct the parts that compose it (enzymes) from descriptions contained within itself (genes). Unlike the living cell (where enzymes are sequences of amino acids and genes are sequences of nucleotides), enzymes and genes are built from the same elementary building blocks, that is, combinators.

10 Ribosomes

Biological ribosomes are arguably the most important component of the fundamental dogma [41]. They translate descriptions of proteins encoded as sequences of nucleotides into polypeptides—sequences of amino acids, the building blocks of proteins. A *computational ribosome* translates a plasmid into one or more enzymes by traversing genes while composing combinators from the neighbourhood, matching those composing the gene. In functional pseudocode, the ribosome evaluates the following expression:

$$\mathrm{map}_|((()^+ \cdot (\mathrm{fold}_>(>\!\!=\!\!>)))\mathrm{P}, \tag{63}$$

where $map_|$ maps functions over the genes G_i that compose plasmid P, and $fold_>$ is right fold over the combinators $c_i(j)$ that compose a gene.

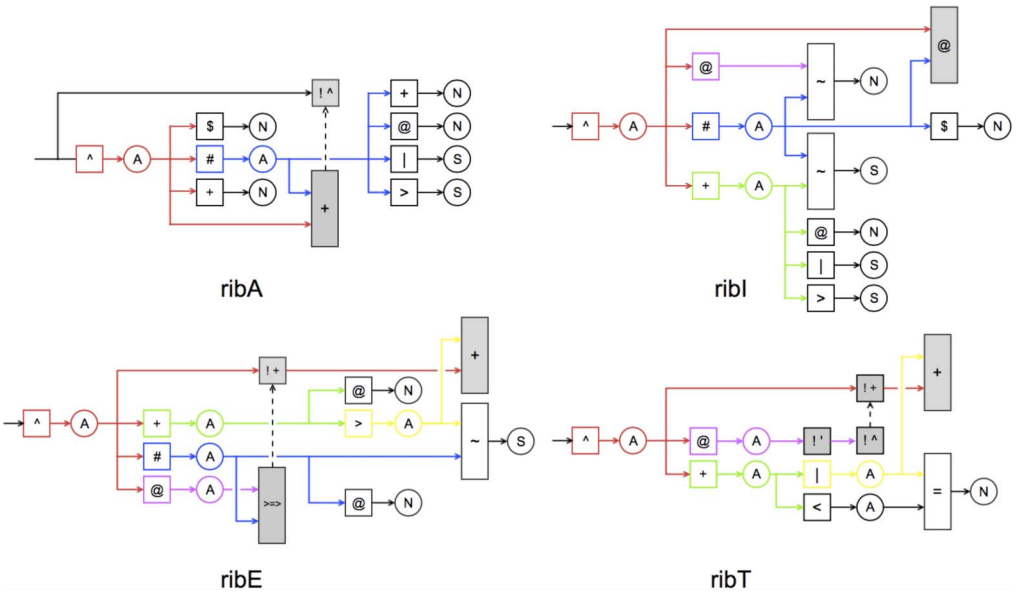

Figure 8. Four methods defining a ribosome.

Biological ribosomes translate messenger RNA into polypeptides, using a four-stage process of *association*, *initiation*, *elongation*, and *termination* [37]. A computational ribosome can be constructed by defining four enzymes with analogous functions (Figure 8) and placing them inside an actor of type object:

$$R = \{\text{ribA},\ \text{ribI},\ \text{ribE},\ \text{ribT}\}_0. \tag{64}$$

RibA first attaches R to the plasmid by adding it to the group of the initial combinator of some gene, $c_i(1)$. Afterwards, ribA is expelled from the ribosome, and R becomes R'. This is advantageous for three reasons. First, ribA has mass; expelling it decreases the transport cost of the ribosome as it traverses the plasmid. Second, ribA consumes time; expelling it prevents it from competing with the ribosome's other (still necessary) enzymes. Third, as part of a self-replicating system of ribosome

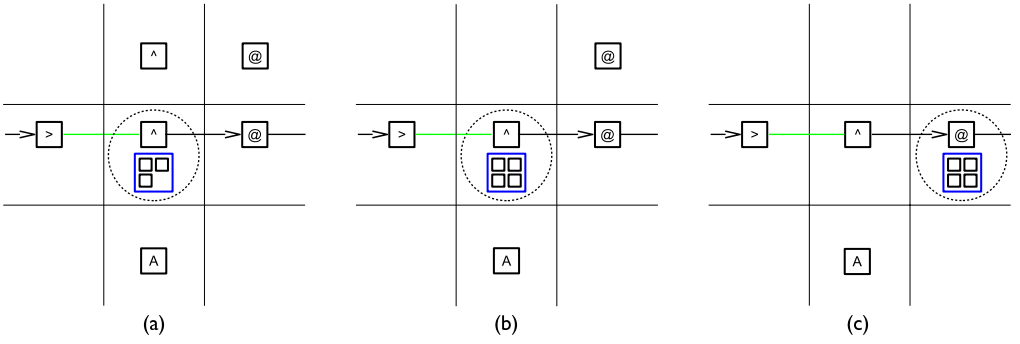

Figure 9. (a) Ribosome attaches itself to plasmid at gene origin (marked by *hand* bond) and ejects ribA. (b) Combinator from neighborhood matching initial combinator is placed inside ribosome. (c) Combinator from neighborhood matching next combinator of plasmid is composed with combinator inside ribosome, and ribosome advances.

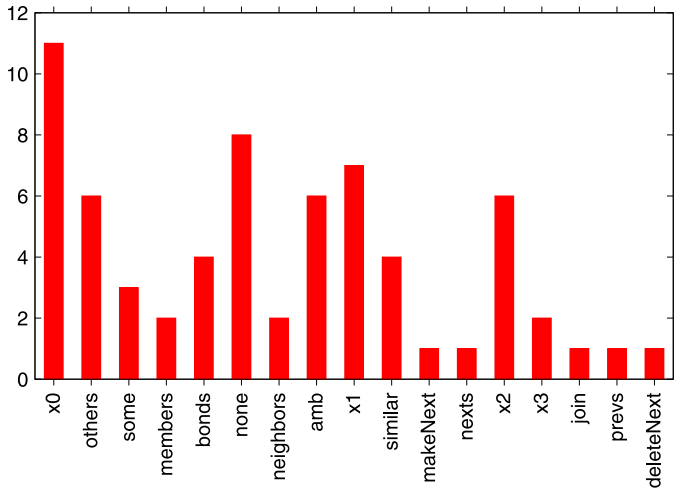

Figure 10. Distribution of primitive combinators constituting a composome.

and replisome factories, the expelled ribA instance can be recycled, that is, used to construct additional ribosomes; see Figure 9a.

After ribosome attachment, ribI finds an actor in the neighborhood with type matching $c_i(1)$ and places it inside R'; see Figure 9b. When R' is at position j on the plasmid, ribE finds a neighbor with type matching $c_i(j+1)$ and composes it with the combinator contained in R', that is, with $c_i(1) \gtrdot \cdots \gtrdot c_i(j)$. It then advances the position of R' to $j+1$ by following the *next* bond; see Figure 9c. This process continues until R' reaches the last combinator in the gene, $c_i(|G_i|)$, which possesses a *hand* bond, at which point ribT promotes the combinator to a method, expels the method, and moves the ribosome across the bond.

If plasmid P and ribosome R are placed in the virtual world with a supply of primitive combinators $\Sigma_P \Sigma_C h(p, c)c$, then the ribosome manufactures the enzymes $\Sigma_P E_p$ described by the plasmid

$$P + R + \Sigma_P \Sigma_C h(p, c)c \rightarrow P + R' + \text{ribA} + \Sigma_P E_p, \tag{65}$$

where C is the set of 42 primitive combinators, and $h(p, c)$ is the number of combinators of type c in G_p and E_p, that is, the gene and enzyme reifications of behavior p.

The goals of the second experiment were to (1) demonstrate that ribosomes correctly translate plasmids into enzymes, and (2) determine the efficiency of parallel translation of a plasmid by multiple ribosomes. Parallel translation by multiple ribosomes is a well-known phenomenon in living cells [40]. To achieve these goals, a plasmid containing three genes,

$$P_3 = cmpA \mid cmpB \mid cmpC, \tag{66}$$

was reified on a grid large enough to contain a circle of radius $r = (n/d\pi)^{1/2}$, where n is the plasmid length and $d = 0.75$ is the combinator density. As raw material, n combinators with distribution matching P_3 (see Figure 10) were also reified inside the circle. The diffusion constant $D = 10^{-2}$. The experiment consisted of five trials in which the number of ribosomes was varied. As combinators were composed by ($\gtrdot$), they were replaced within the circle so that the combinator density remained constant for the entire run. Each trial was repeated for ten runs, and each run ended when time exceeded 10^5 operations per site. The results are shown in Figure 11. We observe that the rate of gene expression increases linearly with the number of ribosomes, achieving a speedup of m times for m ribosomes.

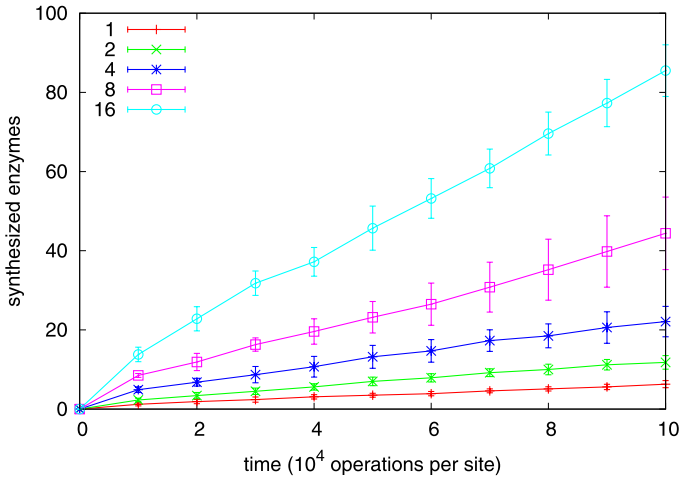

Figure 11. Average number of enzymes (ten runs) synthesized by $2^0,\ldots,2^4$ ribosomes as a function of time (10^4 operations per site). Error bars show ±1 standard deviation.

11 Replisomes

A *quine* is a program that prints itself. All quines consist of two parts. Conventionally called *program* and *data*, they may be thought of as *phenome* and *genome*. All quines work the same way. Active program transforms passive data in two ways, producing new instances of both program and data. Equivalently, the *mother* quine's genome is *translated* and *replicated*, yielding the *daughter* quine's phenome and genome. The forms of the phenome and genome, and the nature of the translation and replication processes, differ from quine to quine. Living cells are, in effect, *reified quines*, and the processes of genome translation and replication are performed by molecular machines called ribosomes and replisomes,[12] respectively. We have already defined a computational ribosome, that is, an object that translates inert descriptions of behaviors encoded by a plasmid (genes) into behaviors reified as methods able to do actual work (enzymes). We now turn our attention to the problem of defining a *computational replisome*, an object that will replicate plasmids. In functional pseudocode, the replisome evaluates the following expression:

$$(\text{fold}_{|}\ (\ |\ )\ \cdot\ \text{map}_{|}\ (\text{fold}_{>}\ (>)))\ P, \tag{67}$$

where (|) and (>) are functions that create undirected and directed bonds, $fold_{|}$ is right fold over the genes G_i that constitute the plasmid P, and $fold_{>}$ is right fold over the combinators $c_i(j)$ that constitute a gene.

Biological replisomes copy plasmids in pairs. Replication begins when two replisomes are assembled at the plasmid's *replication origin*. Each replisome manages one *replication fork*. The replication forks move away from the replication origin in opposite directions, and replication is finished when the pair of replisomes reunite at a position on the plasmid opposite the origin. A computational replisome can be designed that works in a similar way. As in a cell, there are two replication forks. However, unlike a cell, only one moves; the other is stationary. The replisome manages the

12 We use this term to refer to the set of DNA polymerase enzymes [5] that collectively perform DNA replication in the living cell.

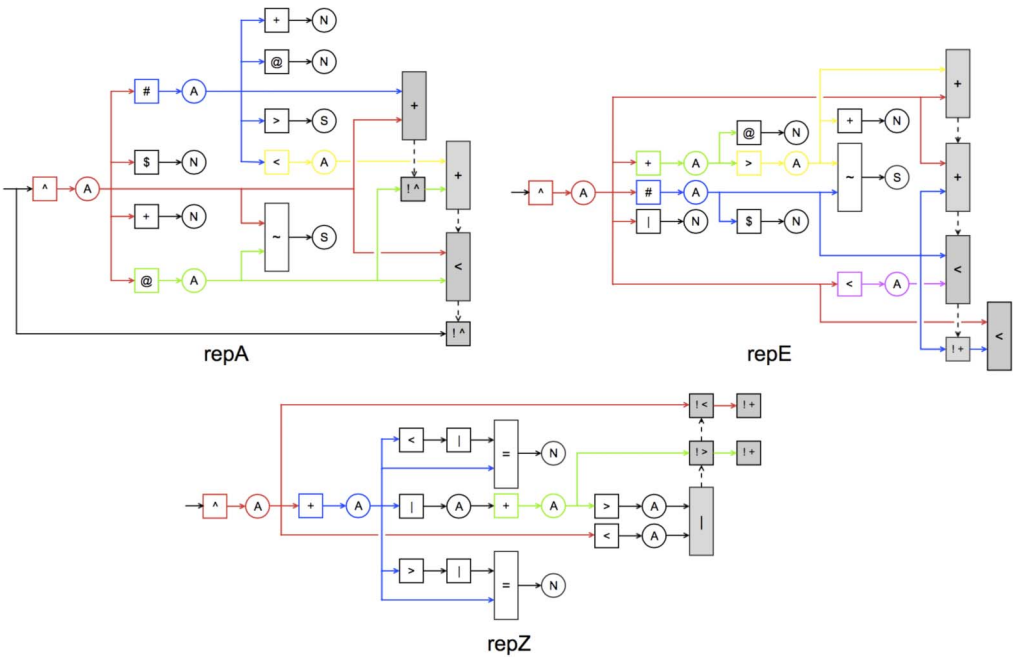

Figure 12. Three of five methods defining a replisome.

active replication fork. It is initially an object containing five enzymes (three of which are shown in Figure 12) and an empty object of the same type as itself:

$$Q = \{\text{repA},\ \text{repE},\ \text{repF},\ \text{repY},\ \text{repZ}, \{\}_2\}_2. \tag{68}$$

A replisome is depicted schematically in Figure 13a. RepA first causes the replisome to attach to the plasmid. It does this by adding the replisome to the group of one of the plasmid's combinators. Note that it can do this at any point, that is, there is no privileged replication origin. Afterwards, the stationary replication fork is marked by attaching the empty object $\{\}_2$ contained

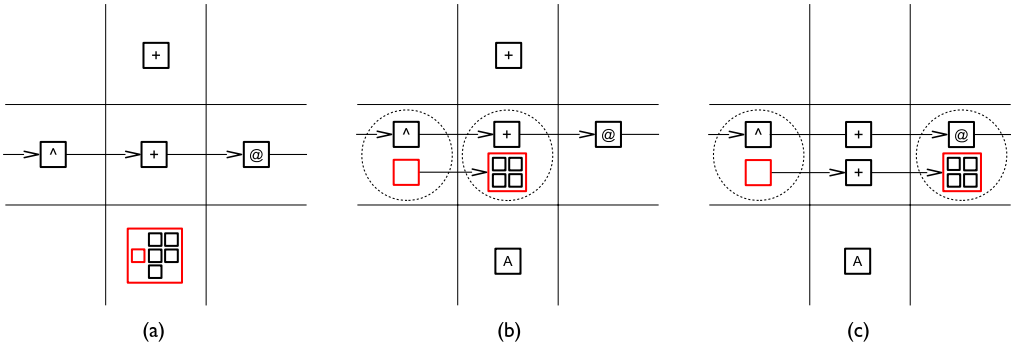

Figure 13. (a) A short segment of a plasmid and a replisome containing five enzymes and an empty object *marker* of the same type as itself. (b) Replisome attaches itself to the plasmid by joining the group of one of its combinators; attaches the marker to the combinator preceding its own attachment site; forms a directed bond with the marker; and ejects repA. (c) Replisome advances along the plasmid after splicing a combinator of the correct type from the neighborhood into the directed bond that trails it. This is the first combinator of the daughter plasmid.

within the replisome to the combinator that precedes the replisome's own attachment site. RepA creates a directed bond from the marker to the replisome and then ejects itself, since it is no longer needed; see Figure 13b.

RepE and repF govern the motion of the replication fork. RepE finds a combinator in the neighborhood matching the combinator attached to replisome $\mathcal{Q}'$. It moves $\mathcal{Q}'$ in the increasing direction (by joining the group of the combinator that follows the replisome's own attachment site) and splices the neighbor into the growing chain (the incomplete *daughter* plasmid) that trails it; see Figure 13c. RepF (not shown in Figure 12) is very similar except that it moves the replication fork through the *hand* bonds that mark the boundaries between genes.

Replication is complete when $\mathcal{Q}'$ encounters a marker, or more precisely, when it finds a marker attached to the combinator that follows its own attachment site. In the most common case, the replisome and marker are situated within a single gene. RepY (not shown in Figure 12) recognizes this situation and creates the final *next* bond, completing the daughter plasmid. Very infrequently, the replisome and marker straddle a boundary between two genes. RepZ recognizes this situation and creates the final *hand* bond. In both cases, the replisome and marker are detached from the plasmid.

If plasmid P and replisome $\mathcal{Q}$ are placed in the virtual world with a supply of primitive combinators $\Sigma_P\, \Sigma_C h(p,\, c)\, c$, then the replisome copies the plasmid

$$P + \mathcal{Q} + \Sigma_P\, \Sigma_C h(p,\, c)\, c \;\rightarrow\; 2\, P + \mathcal{Q}' + \mathrm{repA} + \{\}_2, \tag{69}$$

where C is the set of 42 primitive combinators and $h(p, c)$ is the number of combinators of type c in G_p and E_p, that is, the gene and enzyme reifications of behavior p.

The goals of the third experiment were to (1) demonstrate that replisomes correctly copy plasmids, and (2) determine the efficiency of parallel copying of a plasmid by multiple replisomes. To achieve these goals, the plasmid P_3 was reified on a grid large enough to contain a circle of radius $r = (n/d\pi)^{1/2}$, where n is the plasmid length and d is the combinator density. As raw material, n combinators with distribution matching P_3 (see Figure 10) were also reified inside the circle. The diffusion constant $D = 10^{-2}$. The experiment consisted of 32 trials of ten runs each in which the number of replisomes m and the combinator density d were varied. As combinators were incorporated by bonding into the daughter plasmid, they were replaced within the circle, and r

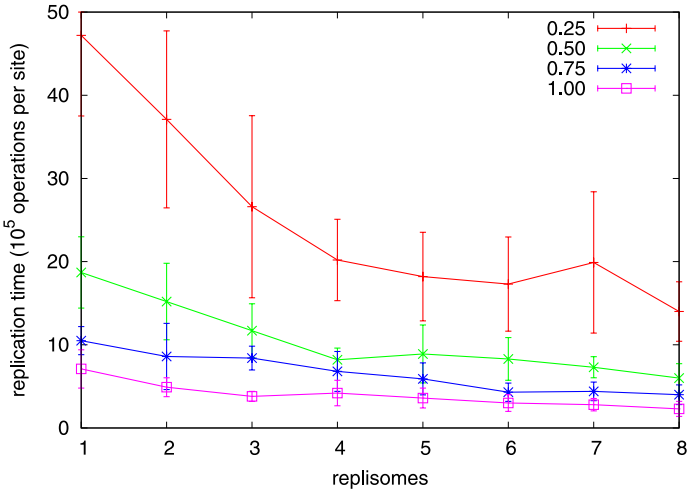

Figure 14. Average time (10^5 operations per site) required for plasmid replication (ten runs) as a function of the number of replisomes for four different combinator densities. Error bars show ±1 standard deviation.

was increased so that combinator density remained constant for the entire run. Runs were continued until either (1) all replisomes detached from the mother plasmid, or (2) the number of combinators copied exceeded the plasmid length. In the second case, the run was disregarded and repeated. This can happen for two reasons, the more common being that the replisome does not recognize the marker and makes a second circuit of the plasmid, resulting in a daughter plasmid of twice the length. Less often, one (or more) replisomes attach to the (incomplete) daughter plasmid instead of the mother, resulting in one (or more) *granddaughter* plasmids.

The average times required for plasmid replication are shown in Figure 14. We observe that, unlike plasmid translation by multiple ribosomes, the speedup for plasmid replication by multiple replisomes is far less than m times for m replisomes. This is not surprising, since the maximum speedup only happens when all replisomes simultaneously attach to the mother plasmid at points uniformly far apart. Finally, we observe that the degree of speedup is a function of combinator density, with larger speedups occurring for lower densities.

12 A Self-Replicating System of Ribosome and Replisome Factories

In prior sections we defined computational ribosomes and replisomes and demonstrated their ability to translate and copy a plasmid containing descriptions of three methods composing a simple composome. An arguably more interesting experiment would be to construct a plasmid containing the genes defining ribosomes and replisomes themselves, so that ribosomes would synthesize the enzymes that compose both. However, unlike composomes, ribosomes and replisomes do not assemble themselves. To perform this function, we need an additional machine to collect the finished enzymes and place them inside objects of the correct types; we call this machine a *factory*. Abstractly, factories are copiers of *compositional information*, which is heritable information distinct from the *genetic information* copied by replisomes, and which ribosomes translate into enzymes. Concretely, factories are objects containing a specific set of

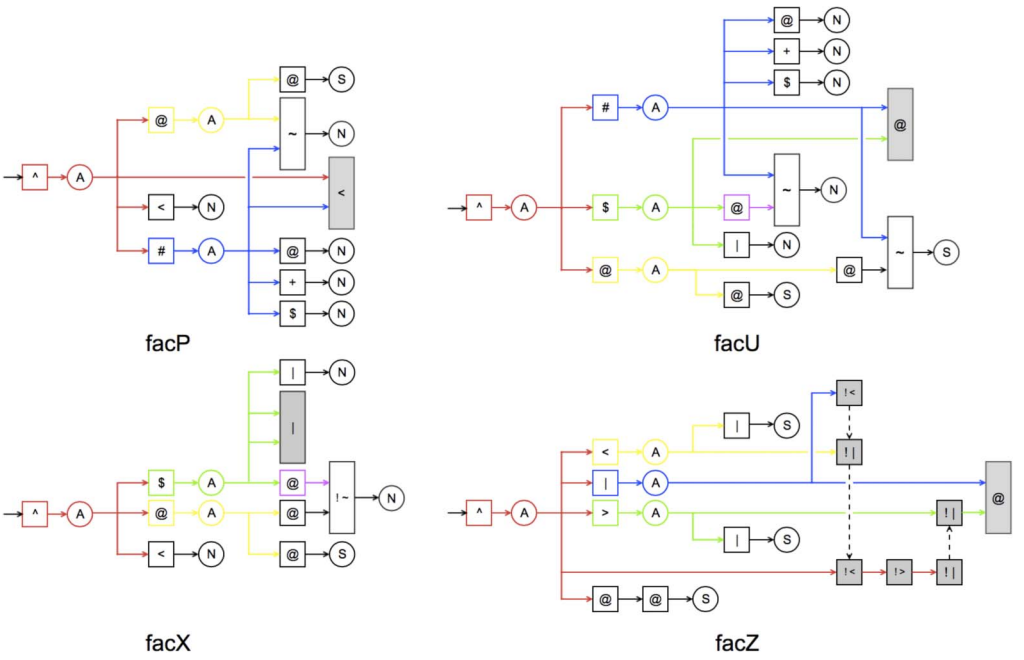

Figure 15. Four of eight methods defining a factory.

enzymes (four of which are shown in Figure 15) and a *model*, which can be either a ribosome or a replisome; see Figure 16a. A factory's enzymes can be grouped into four categories:

- FacP, facN, and facH form *prev*, *next*, and *hand* bonds with empty objects from the neighborhood. The order in which these three events occur is more or less random. Because the enzymes are very similar, only facP is shown in Figure 15. The objects $\{\}_k$ bonded to the mother factory by *prev* and *next* bonds will become new instances of the model. The object $\{\}_{k+1}$ bonded to the mother factory by the *hand* bond will become the daughter factory; see Figure 16b.
- FacU moves enzymes with types matching contents of the model into the incomplete model instances. FacV (not shown in Figure 15) moves enzymes with types matching contents of the mother factory into the incomplete daughter factory.
- FacX uses the generalized set difference operator ($\boxed{!\sim}$) to verify that a new model instance, that is, *product*, has all of the enzymes that the old model contains. If so, it marks the product as complete, using a self-directed *hand* bond. FacY (not shown in Figure 15) does the same thing for the daughter factory but uses a self-directed *prev* bond to indicate completeness.
- FacZ checks to see that both products have self-directed *hand* bonds and also that the daughter factory has a self-directed *prev* bond. If so, it (1) deletes the *prev* and *next* bonds connecting the mother factory and the products; (2) moves one of the completed products into the daughter factory (to serve as its model); and (3) deletes the *hand* bond connecting the mother and daughter factories; see Figure 16c.

Given the above enzymes, it is now possible to define a self-replicating ribosome factory:

$$F_{\mathrm{R}} = \{\mathrm{facP}, \mathrm{facN}, \mathrm{facH}, \mathrm{facU}, \mathrm{facV}, \mathrm{facX}, \mathrm{facY}, \mathrm{facZ}, R\}_1. \tag{70}$$

When placed in the virtual world with a supply of empty objects $\{\}_0$ and $\{\}_1$ and enzymes composing ribosomes $\Sigma_R E_r$ and factories $\Sigma_F E_f$, the ribosome factory constructs a new ribosome factory and a new ribosome:

$$F_{\mathrm{R}} + 2\{\}_0 + \{\}_1 + 2\Sigma_R E_r + \Sigma_F E_f \;\rightarrow\; 2F_{\mathrm{R}} + R. \tag{71}$$

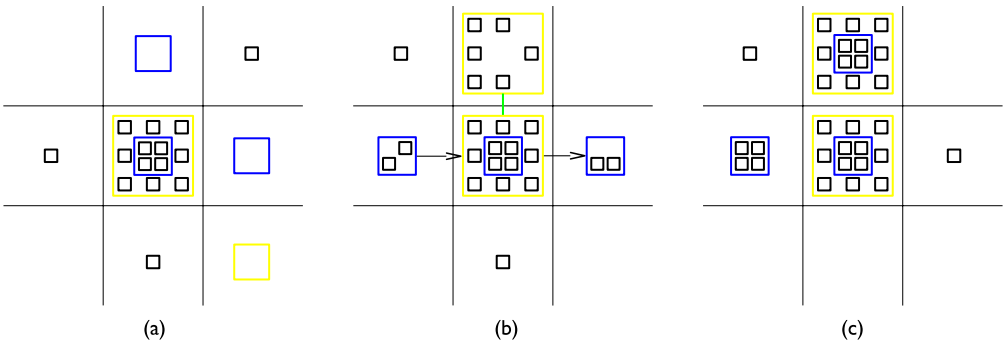

Figure 16. (a) Self-replicating ribosome factory contains eight enzymes and a model ribosome. (b) Directed bonds connect the factory to partially assembled ribosomes while an undirected bond connects it to a partially assembled daughter factory. (c) One of the product ribosomes becomes the model for the daughter factory while the second is available to synthesize enzymes for the encompassing self-replicating system.

A self-replicating replisome factory can be defined similarly:

$$F_{\mathcal{Q}} = \{\text{facP, facN, facH, facU, facV, facX, facY, facZ, } \mathcal{Q}\}_3. \tag{72}$$

When placed in the virtual world with a supply of empty objects $\{\}_2$ and $\{\}_3$ and enzymes composing replisomes $\Sigma_{\mathcal{Q}} E_q$ and factories $\Sigma_F E_f$, the replisome factory constructs a new replisome factory and a new replisome:

$$F_{\mathcal{Q}} + 4\{\}_2 + \{\}_3 + 2\Sigma_{\mathcal{Q}} E_q + \Sigma_F E_f \rightarrow 2F_{\mathcal{Q}} + \mathcal{Q}. \tag{73}$$

Note that the left-hand side of the reaction contains four empty objects $\{\}_2$ instead of two; the extras are the markers contained by the new replisome and replisome model.

We now have all of the components needed to build a self-replicating system of ribosome and replisome factories. Unlike the composome, which copied itself solely by reflection, this self-replicating system is a quine that translates and replicates a self-description reified as a data structure within the virtual world itself:

$$\begin{aligned} P_{17} = {} & \textit{ribA} \mid \textit{ribI} \mid \textit{ribE} \mid \textit{ribT} \mid \textit{repA} \mid \textit{repE} \mid \textit{repF} \mid \textit{repY} \mid \textit{repZ} \\ & \mid \textit{facP} \mid \textit{facN} \mid \textit{facH} \mid \textit{facU} \mid \textit{facV} \mid \textit{facX} \mid \textit{facY} \mid \textit{facZ}. \end{aligned} \tag{74}$$

This *genome* consists of a single plasmid containing 586 combinators of 31 types composing 17 genes. The minimum *phenome* required for bootstrapping the self-replicating system consists of a replisome factory $F_{\mathcal{Q}}$, a ribosome factory F_{R}, and a ribosome R. When genome P_{17} and phenome $F_{\mathcal{Q}} + F_{\mathrm{R}} + \mathrm{R}$ are placed in the world with a supply of empty objects $\{\}_k$ and combinators $\Sigma_{P_{17}} \Sigma_C h(p, c)c$, the system increases the numbers of all of its component parts:

$$\begin{aligned} & P_{17} + F_{\mathcal{Q}} + F_{\mathrm{R}} + \mathrm{R} + 2\{\}_0 + \{\}_1 + 4\{\}_2 + \{\}_3 + 3\Sigma_{P_{17}} \Sigma_C h(p, c)c \rightarrow \\ & \quad 2P_{17} + 2F_{\mathcal{Q}} + 2F_{\mathrm{R}} + \mathcal{Q}\prime + \mathrm{repA} + \{\}_2 + \mathrm{R} + \mathrm{R}' + \mathrm{ribA}. \end{aligned} \tag{75}$$

Note that there are three instances of $\Sigma_{P_{17}} \Sigma_C h(p, c)\, c$ on the left side of the equation. The ribosome R consumes the first two, making two full circuits of the plasmid synthesizing the system's enzymes

$$P_{17} + \mathrm{R} + 2\Sigma_{P_{17}} \Sigma_C h(p, c)c \rightarrow P_{17} + \mathrm{R}' + \mathrm{ribA} + 2\Sigma_{\mathcal{Q}} E_q + 2\Sigma_{\mathrm{R}} E_r + 2\Sigma_F E_f, \tag{76}$$

while the replisome $\mathcal{Q}$ (assembled by $F_{\mathcal{Q}}$) uses the last copying the plasmid.

It is useful to compare the normalized complexity of the self-replicating system of ribosome and replisome factories with that of the composome defined earlier. Recall that the composome X is composed of 3 enzymes: cmpA, cmpB, and cmpC; these enzymes are in turn composed of 66 combinators of 17 types. Because the enzymes are defined outside the system, their complexity is non-contingent, and the composome's normalized complexity is quite low:

$$\frac{K(\{\text{cmpA, cmpB, cmpC}\} \mid \text{cmpA} + \text{cmpB} + \text{cmpC})}{K(\text{cmpA} + \text{cmpB} + \text{cmpC} \mid \mathit{OOCC}_{17}) + K(\mathit{OOCC}_{17} \mid \mathit{ACA}) + K(\mathit{ACA})}, \tag{77}$$

where $K(\text{cmpA} + \text{cmpB} + \text{cmpC} \mid \mathit{OOCC}_{17})$ is the portion of the composome's non-contingent complexity contained in its three enzymes.

In contrast, the self-replicating system of ribosome and replisome factories is composed of 17 different behaviors reified as both genes and enzymes; these genes and enzymes are in turn composed of $3 \times 586 = 1758$ combinators of 31 types. However, because the enzymes are defined within the system itself (by the genes), their complexity (unlike that of the composome's enzymes)

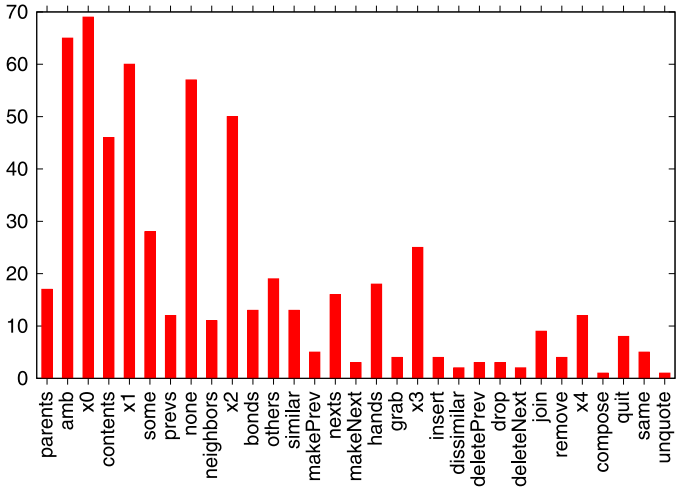

Figure 17. Distribution of primitive combinators composing a self-replicating system of ribosome and replisome factories.

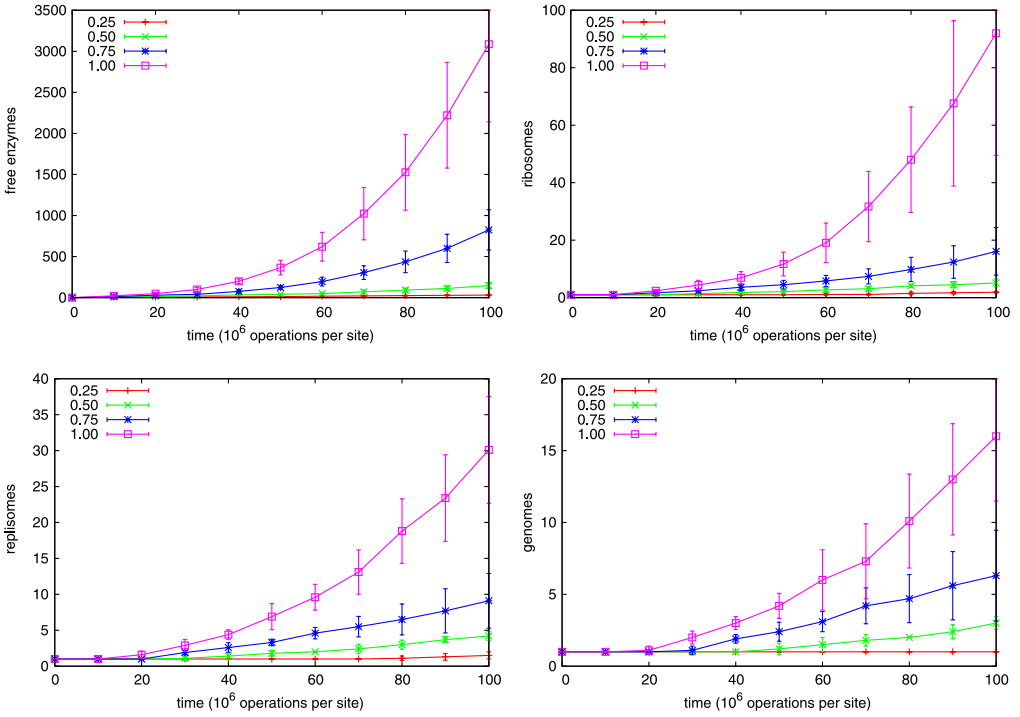

Figure 18. Average numbers of free enzymes, ribosomes (and ribosome factories), replisomes (and replisome factories), and complete genome copies (ten runs) as a function of time (10^6 operations per site) for four different combinator densities. Error bars show ±1 standard deviation.

is contingent. Consequently, the self-replicating system of ribosome and replisome factories possesses significantly higher normalized complexity than the composome:

$$\frac{K(F_Q + F_R + R \mid \Sigma_{P_{17}} E_p) + K(\Sigma_{P_{17}} E_p \mid P_{17}, R, OOCC_{31}) + K(P_{17} \mid OOCC_{31})}{K(OOCC_{31} \mid ACA) + K(ACA)}, \tag{78}$$

where $K(F_Q + F_R + R \mid \Sigma_{P_{17}} E_p)$ and $K(P_{17} \mid OOCC_{31})$ are the compositional and genetically encoded portions of the self-replicating system's contingent complexity, and $K(\Sigma_{P_{17}} E_p \mid P_{17}, R, OOCC_{31})$ is zero because the plasmid P_{17} encodes the enzymes $\Sigma_{P_{17}} E_p$ using a process defined by the ribosome R and the object-oriented combinator chemistry $OOCC_{31}$.

The goal of the final experiment was to demonstrate a self-replicating system of ribosome and replisome factories. To achieve this goal, the plasmid P_{17} was reified on a grid large enough to contain a circle of radius $r = (n/d\pi)^{1/2}$, where n is the plasmid length and d is the combinator density. As raw material, n combinators with distribution matching P_{17} (see Figure 17) and 2 empty objects of each of 4 types were also reified inside the circle. The neighborhood was of size 3×3, and the diffusion constant $D = 3 \times 10^{-2}$. The experiment consisted of four trials in which the combinator density d was varied. When a combinator was composed by (>=>) or incorporated by bonding into a daughter plasmid, a new combinator of the same type was added to the circle. When a method was inserted into an empty object, a new empty object of the same type was added to the circle. When a combinator was incorporated into a daughter plasmid or a method was inserted into an empty object, r was increased so as to keep the density of combinators and empty objects constant. Each trial was repeated for ten runs, and each run ended when the time exceeded 10^8 operations per site.

The results are shown in Figure 18. We observe that the system robustly replicates all of its components many times over. We further observe that (on average) ribosomes are synthesized at a higher rate than replisomes and both are synthesized more frequently than the genome. This reflects the fact that control within the parallel distributed computation is open-loop with no feedback. The system therefore suffers to some degree from a data race between ribosome and replisome factories; since ribosomes are shorter, ribosome factories tend to win the competition for the shared resource of factory enzymes. The fact that there are fewer genome copies than replisomes can be explained by the fact that multiple replisomes often bind to the same plasmid.

13 Conclusion

Sixty years after von Neumann conceived his self-replicating automaton, it remains a paragon of non-biological life. The rules governing CAs seem simple and physical, and partly for this reason, the automaton von Neumann constructed using them is uniquely impressive. Yet, perhaps because RASPs are (in comparison to CAs) such powerful hosts, self-replicating programs in conventional programming languages seem somehow less convincing; all self-replicating programs must lift themselves up by their own bootstraps, yet not all programs lift themselves the same distance. With respect to models of computation, one might assume that increased expressive power can only come at the expense of realism. The model of computation introduced in this article demonstrates that this is not the case. The field of programming languages has made remarkable advances in the years since von Neumann conceived his automaton. Modern functional programming languages like Haskell bear little resemblance to the machine languages that are native to RASPs. The concept of combinators as the building blocks of programs is well established in functional programming. This article has demonstrated that the behaviors of actors in a 2D virtual world can be defined using a visual programming language and compiled into sequences of combinators with simple, well-defined semantics. Using programs constructed from combinators, reified actors asynchronously update the states of actors in their neighborhoods and (in doing so) function as the elements of parallel distributed computations. Because actors report the time they use to perform these updates, and

pay for it in the simulation, the object-oriented combinator chemistry described in this article can be viewed as an abstract interface to an ACA, an indefinitely scalable model of computation with the affordances of a natural physics. In summary, this article has demonstrated that it is possible to build programs that build programs from combinators of a small number of predefined types using asynchronous spatial processes that resemble chemistry as much as computation.

Acknowledgments

Special thanks to Elena Sharnoff. Thanks also to Dave Ackley, George Bissias, Joe Collard, Stephen Harding, Tim Hutton, Lee Jensen, Barry McMullin, Lee Spector, and Darko Stefanovic.

References

1. Ackley, D. (2013). Bespoke physics for living technology. *Artificial Life*, *34*, 381–392.
2. Arbib, M. A. (1966). Simple self-reproducing universal automata. *Information and Control*, *9*(2), 177–189.
3. Berman, P., & Simon, J. (1988). Investigations of fault-tolerant networks of computers. In *ACM Symposium on the Theory of Computing* (pp. 66–77).
4. Codd, E. F. (1968). *Cellular automata*. London: Academic Press.
5. del Solar, G., Giraldo, R., Ruiz-Echevarría, M. J., Espinosa, M., & Díaz-Orejas, R. (1998). Replication and control of circular bacterial plasmids. *Microbiology and molecular biology reviews*, *62*(2), 434–464.
6. di Fenizio, P. S. (2000). A less abstract artificial chemistry. In *Proceedings of the 7th International Conference on the Simulation and Synthesis of Living Systems (ALIFE)* (pp. 49–53).
7. Dittrich, P., Ziegler, J. C., & Banzhaf, W. (2001). Artificial chemistries: A review. *Artificial Life*, *7*(3), 225–275.
8. Fatès, N. (2013). A guided tour of asynchronous cellular automata. In *Cellular automata and discrete complex systems: 19th International Workshop* (pp. 15–30).
9. Fontana, W., & Buss, L. W. (1996). *The barrier of objects: From dynamical systems to bounded organizations*. International Institute for Applied Systems Analysis.
10. Freitas, R. A., & Merkle, R. C. (2004). *Kinematic self-replicating machines*. Georgetown, TX: Landes Bioscience.
11. Gillespie, D. T. (1977). Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, *81*(25), 2340–2361.
12. Hewitt, C., Bishop, P., & Steiger, R. (1973). A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 235–245).
13. Hickinbotham, S., Clark, E., Stepney, S., Clarke, T., Nellis, A., Pay, M., & Young, P. (2011). Molecular microprograms. In *European Conference on Artificial Life (ECAL)* (pp. 297–304).
14. Hughes, J. (1998). Generalising monads to arrows. *Science of Computer Programming*, *37*, 67–111.
15. Hutton, T. J. (2004). A functional self-reproducing cell in a two-dimensional artificial chemistry. In *Proceedings of the 9th International Conference on the Simulation and Synthesis of Living Systems (ALIFE)* (pp. 444–449).
16. Kiselyov, O., Shan, C., Friedman, D. P., & Sabry, A. (2005). Backtracking, interleaving, and terminating monad transformers (functional Pearl). *ACM SIGPLAN Notices*, *40*(9), 192–203.
17. Kolmogorov, A. (1965). Three approaches to the quantitative definition of information. *Problems of Information Transmission*, *1*(1), 1–7.
18. Laing, R. A. (1977). Automaton models of reproduction by self-inspection. *Journal of Theoretical Biology*, *66*(1), 437–456.
19. Landin, P. J. (1964). The mechanical evaluation of expressions. *The Computer Journal*, *6*(4), 308–320.
20. Langton, C. G. (1984). Self-reproduction in cellular automata. *Physica D: Nonlinear Phenomena*, *10*(1), 135–144.
21. Liang, S., Hudak, P., & Jones, M. (1995). Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95* (pp. 333–343).

22. McCarthy, J. (1963). A basis for a mathematical theory of computation. In P. Braffort & D. Hirschberg (Eds.), *Computer programming and formal systems* (pp. 33–70). Amsterdam: North-Holland.

23. Moggi, E. (1991). Notions of computation and monads. *Information and Computation*, *93*(1), 55–92.

24. Nakamura, K. (1974). Asynchronous cellular automata and their computational ability. *Systems, Computers, Controls*, *5*(5), 58–66.

25. Nehaniv, C. L. (2004). Asynchronous automata networks can emulate any synchronous automata network. *International Journal of Architectural Computing*, *14*(5–6), 719–739.

26. Pattee, H. (1995). Evolving self-reference: Matter, symbols, and semantic closure. *Communication and Cognition—Artificial Intelligence*, *12*, 9–27.

27. Paun, G. (1998). Computing with membranes. *Journal of Computer and System Sciences*, *61*, 108–143.

28. Peyton Jones, S., Hughes, J., Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinze, R., Hudak, P., et al. (1999). *Report on the programming language Haskell 98* (Technical Report YALEU/DCS/RR-1106). New Haven, CT: Yale University.

29. Segré, D., Ben-Eli, D., & Lancet, D. (2000). Compositional genomes: Prebiotic information transfer in mutually catalytic noncovalent assemblies. In *Proceedings of the National Academy of Sciences of the U.S.A.*, *97*(8), 4112–4117.

30. Sipper, M. (1998). Fifty years of research on self-replication: An overview. *Artificial Life*, *4*(3), 237–257.

31. Smith, A., Turney, P. D., & Ewaschuk, R. (2003). Self-replicating machines in continuous space with virtual physics. *Artificial Life*, *9*(1), 21–40.

32. Smith, A. R. (1971). Cellular automata complexity trade-offs. *Information and Control*, *18*(5), 466–482.

33. Spector, L., & Robinson, A. (2002). Genetic programming and auto-constructive evolution with the Push programming language. *Genetic Programming and Evolvable Machines*, *3*(1), 7–40.

34. Sutherland, W. R. (1966). *The on-line graphical specification of computer procedures*. Ph.D. thesis, MIT.

35. Taylor, T. (2001). Creativity in evolution: Individuals, interactions and environments. In P. Bentley & D. Corne (Eds.), *Creative evolutionary systems* (pp. 79–108). San Francisco: Morgan Kaufman.

36. Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Journal of Mathematics*, *58*(5), 345–363.

37. von der Haar, T. (2012). Mathematical and computational modelling of ribosome movement and protein synthesis: An overview. *Computational and Structural Biotechnology Journal*, *1*(1), 1–7.

38. von Neumann, J. (1966). *Theory of self-replicating automata*. Urbana: University of Illinois Press.

39. Wadler, P. (1990). Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP '90*.

40. Warner, J., Knopf, P. M., & Rich, A. (1963). A multiple ribosomal structure in protein synthesis. *Proceedings of the National Academy of Sciences of the U.S.A.*, *49*, 122–129.

41. Watson, J. D., & Crick, F. H. (1953). Molecular structure of nucleic acids. *Nature*, *171*(4356), 737–738.