DOI: http://dx.doi.org/10.7551/978-0-262-33027-5-ch033

Programs as Polypeptides

Lance R. Williams[1]

[1]Department of Computer Science, University of New Mexico, Albuquerque, NM 87131
williams@cs.unm.edu

Abstract

We describe a visual programming language for defining behaviors manifested by reified actors in a 2D virtual world that can be compiled into programs comprised of sequences of combinators that are themselves reified as actors. This makes it possible to build programs that build programs from components of a few fixed types delivered by diffusion using processes that resemble chemistry as much as computation.

Introduction

Self-replicating programs have been defined using computational models that vary in *expressiveness* and *verisimilitude*. If we adopt the definition used in the field of programming languages (Felleisen, 1990), then expressiveness varies along a spectrum that begins with *cellular automata* (CA) defined using lookup tables, increases with *artificial chemistries* based on symbol rewrite rules, and peaks in (more or less) conventional programming languages (which themselves vary along a spectrum that begins with machine language and ends in high-level languages like Lisp).

By verisimilitude, we mean providing an interface with the affordances and limitations of a natural physics. Models with high verisimilitude define *virtual worlds*. Because CAs are spatially embedded and governed by simple rules defined on local neighborhoods, most would say that the verisimilitude of CAs is high. However, since state is updated everywhere synchronously, and this (unlike a natural physics) requires a global clock, CAs are not *indefinitely scalable* (Ackley, 2013). Because *asynchronous cellular automata* (ACA) do not suffer from this limitation yet are just as powerful (Nakamura, 1974; Berman and Simon, 1988; Nehaniv, 2004), ACAs are the gold standard in virtual worlds.

Many artificial chemistries lack verisimilitude because the symbols that the rewrite rules transform are not embedded in any physical space (Berry and Boudol, 1990; Paun, 1998; Fontana and Buss, 1999). Others have far greater resemblance to real physical systems (Laing, 1977; Smith et al., 2003; Hutton, 2004). These assign symbols to positions in a virtual world, restrict interactions to local neighborhoods, and rely on diffusion for data transport.

Programs written in conventional programming languages generally require a *random access stored program* (RASP) computer to host them.[1] Because of *program-data equivalence*, RASPs permit relatively simple solutions to the self-replication problem based on *reflection.* Yet self-replicating programs written in conventional programming languages are (in effect) stuck in boxes; it makes no difference whether it is one big box (Ray, 1994) or many little boxes interacting in a virtual world (Adami et al., 1994); because they read, write, and reside in random access memories, the programs themselves are fundamentally non-physical.

In the game of defining virtual worlds and creating self-replicating programs inside those worlds, there is a tradeoff between the non-contingent complexity of *physical law* and the purely contingent complexity of the *initial conditions* that define a program and its self-description. We propose that the ratio of contingent and non-contingent complexity is positively correlated with the property that Pattee (1995) calls *semantic closure*. Ideally, we would like to pursue an approach that combines the expressiveness of conventional programming languages with the physical verisimilitude of ACAs while maximizing the ratio of contingent and non-contingent complexity. To do this, we need to break programs out of their boxes; we need reified programs that assemble copies of themselves from reified building blocks; we need to imagine *programs as polypeptides.*

Superficially, there is a similarity between the sequences of instructions that comprise a machine language program and the sequences of nucleotides and amino acids that comprise 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 comprise DNA and RNA are only of four types; the amino acids that comprise polypeptides are only of twenty; and while bits might conceivably play the passive represen-

[1]See Williams (2014) for a notable exception.

Lance R. Williams (2015) Programs as Polypeptides. Proceedings of the European Conference on Artificial Life 2015, pp. 150-157

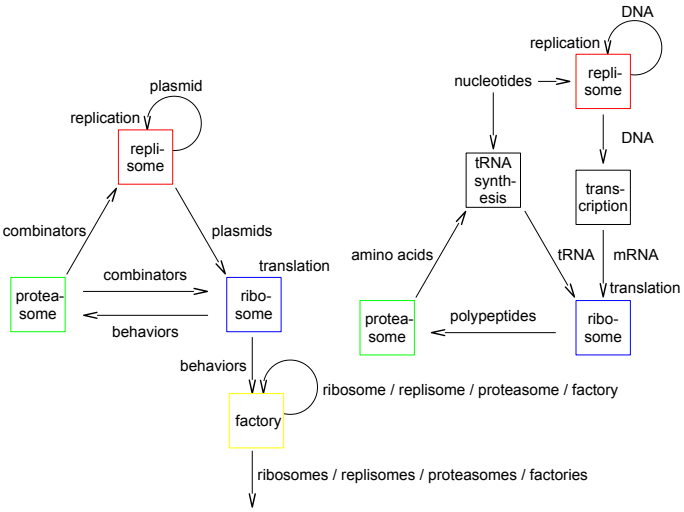

Figure 1: Framework proposed in this paper (left). Fundamental dogma of molecular biology (right).

tational role of nucleotides, they can not play the active functional role of amino acids; this role can only be played by instructions. While the instruction set of a simple RASP can be quite small, the number of distinct *operands* that (in effect) modify the instructions is a function of the word size of the machine, and is therefore (at a minimum) in the thousands.[2] The implication for the study of self-replicating programs is profound: while biopolymers can be assembled by physical processes from building blocks of a few fixed types, it is impossible to construct machine language programs for a RASP this way.

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 can not 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 (1953) thought this idea so important that they called it "the fundamental dogma of molecular biology." It is the antithesis of the program-data equivalence which makes reflection possible. See Figure 1.

Combinators are functions with no free variables. In this paper we show how programs in a visual programming language just as expressive as machine language can be compiled into sequences of combinators of only forty two types. Where machine language programs would use iteration, the programs that we compile into combinators employ *nondeterminism*. The paper culminates in the experimental demonstration of a *computational ribosome*, a 'machine' in a 2D virtual world that assembles programs from combinators using inert descriptions of programs (also comprised of combinators) as templates.

[2] Although they play many roles in machine language programs, non-register operands are generally *addresses*.

Reified Actors

Actors are created using three different constructors: $[\,]^-$ creates *combinators*, $[\,]^+$ creates *behaviors*, and $[\,]_k$ creates *objects*. Like amino acids, which can be composed to form polypeptides, *primitive* combinators can be composed to form *composite* combinators. Behaviors are just combinators that have been repackaged with the $[\,]^+$ constructor. Prior to repackaging, combinators do not manifest their function; this might correspond (in our analogy) to the folding of a polypeptide chain into a *protein*.

Objects are containers that can contain other actors. Each is one of four immutable types: $[\,]_0$, $[\,]_1$, $[\,]_2$ and $[\,]_3$. For example, $[x, y, z]_2$ is an object of type two that contains three actors, x, y and z. 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 destroyed, mass is conserved.

Actors are reified by assigning them positions in a 2D virtual world. Computations progress when actors interact with other actors in their 8-neighborhoods by means of the behaviors they manifest. 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 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 Paun (1998), objects can be nested to any level of depth. 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 be associated 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, groups define 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 behaviors manifested by actors in an embedded group (or contained inside such an actor) share a finite time resource in a zero sum fashion. Complex computations formulated in terms of large numbers of actors manifesting behaviors 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

Lance R. Williams (2015) Programs as Polypeptides. Proceedings of the European Conference on Artificial Life 2015, pp. 150-157

them. Undirected bonds are defined by the *hand* relation H, which is a *symmetric relation* on the set of actors, *i.e.,* $H(x,y) = H(y,x)$. Directed bonds are defined by the *previous* and *next* relations, P and N, which are *inverse relations* on the set of actors, *i.e.,* $P(x,y) = N(y,x)$.

If the types of combinators and behaviors were defined by the sequences of primitive combinators of which they are composed, then determining type equivalence would be relatively expensive. 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 values of composite combinators are defined as the product of the hash values of their components; primitive combinators have hash values equal to prime numbers.[3] Type equivalence for behaviors is defined in the same way, the types of combinators and behaviors 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 components by prime factorization.[4]

Apart from composition, containment, group 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.

Non-deterministic Comprehensions

Sets can be converted into *superpositions* using the non-deterministic choice operator (McCarthy, 1963):

$$\begin{aligned} \text{amb}\,\{\} &= \langle\rangle \\ \text{amb}\,\{x, y\ldots\} &= \langle x, y\ldots\rangle. \end{aligned}$$

When *amb* is applied to a non-empty set, it causes the branch of the non-deterministic computation that called *amb* to fork. Conversely, empty sets cause the branch to fail. When a branch fails, the deterministic implementation backtracks.

Monads are an abstract datatype that allows programmers to define rules for composing functions that deviate from mathematically pure functions in prescribed ways. Multivaluedness (represented by sets) and non-determinism (represented by superpositions) are just two examples. The monad interface is defined by two operations called *unit* and *bind*. *Unit* transforms ordinary values *a* into *monadic values*, *e.g.,* $\text{unit}_A\, x = \langle x\rangle$ where A is the superposition monad. Functions like *unit* that take ordinary values and return monadic values are termed *monadic functions. Bind* (the infix operator '$>\!\!>\!\!=$' in Haskell) allows monadic functions to be applied to monadic values. This permits monadic functions to be chained; the output of one provides the input to the next.

Monads are intimately related to set builder notation or *comprehensions*. By way of illustration, consider the following non-deterministic comprehension that fails if n is prime and returns a (non-specified) factor of n if n is composite:

$$\langle\, x \mid x \in \langle 1\,..\,n-1\rangle\,,\, y \in \langle 1\,..\,x\rangle\,,\, xy \;=\; n\,\rangle\,.$$

Wadler (1990) showed that notation like the above is syntactic sugar for monadic expressions and described a process for translating the former into the latter. Comprehension *guards*, *e.g.,* $xy = n$, are translated using the function

$$\begin{aligned} \text{guard}_M\ \mathit{True} &= \text{unit}_M\ \bot \\ \text{guard}_M\ \mathit{False} &= \text{zero}_M \end{aligned}$$

where M is the monad and $\bot$ is *undefined.* 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

$$\lambda n \to (\text{unit}_A\, n \overset{A}{>\!\!>\!\!=} \text{unit}_A \cdot (-1) \overset{A}{>\!\!>\!\!=} \text{amb} \cdot \iota \overset{A}{>\!\!>\!\!=}$$

$$\lambda x \to (\text{unit}_A\, x \overset{A}{>\!\!>\!\!=} \text{amb} \cdot \iota \overset{A}{>\!\!>\!\!=} \text{unit}_A \cdot (\times x) \overset{A}{>\!\!>\!\!=}$$

$$\text{unit}_A \cdot (= n) \overset{A}{>\!\!>\!\!=} \text{guard}_A \overset{A}{>\!\!>\!\!=} \text{unit}_A\, x))$$

where $(\iota\, x)$ equals $\{1\,..\,x\}$.

From Comprehensions to Dataflow Graphs

Recall that our goal is to create programs comprised solely of combinators. To maximize composability, these combinators should be of a single type, yet the desugared comprehension above contains functions of many different types. However, if sets are used to represent sets, singleton sets are used to represent scalars, and non-empty and empty sets are used to represent *True* and *False*, then the type signatures

$$\begin{aligned} \to \boxed{f'} \to &:: \{a\} \to \langle\{a\}\rangle \\ \overset{\to}{\to} \boxed{g'} \to &:: \{a\} \to \{a\} \to \langle\{a\}\rangle \end{aligned}$$

are general enough to represent the types of all functions in the desugared comprehension. To prove this, we first show that *amb* can be lifted to the type, $\{a\} \to \langle\{a\}\rangle$, as follows:

$$\begin{aligned} \text{amb}'\,\{\} &= \langle\rangle \\ \text{amb}'\,\{x, y\ldots\} &= \langle\{x\},\{y\}\ldots\rangle. \end{aligned}$$

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

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

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

[4] This is analogous to the function in the cell which is performed by the molecular assemblies called proteasomes and in the organelles called lysosomes.

Lance R. Williams (2015) Programs as Polypeptides. Proceedings of the European Conference on Artificial Life 2015, pp. 150-157

and can be defined as follows

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

where $join_S$ is right fold of $(\cup)$ and

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

Bind can then be used with $unit_A$ to lift ι into a function

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

with the type, $\{a\} \to \langle\{a\}\rangle$, as demonstrated below

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

Next we define two functions of type, $\{a\} \to \langle\{a\}\rangle$, to replace *guard*. The first causes a computation to fail when its argument is empty while the second does the opposite:

$$\begin{aligned} \text{some}'\,\{\} &= \langle\rangle \\ \text{some}'\,\{x, y \ldots\} &= \langle\{x, y \ldots\}\rangle \\ \text{none}'\,\{\} &= \langle\{\}\rangle \\ \text{none}'\,\{x, y \ldots\} &= \langle\rangle. \end{aligned}$$

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, sensible lifted forms for these functions are easily defined. For example

$$\begin{aligned} \text{pred}' &= \text{unit}_A \cdot (\text{map}_S\, (-1)) \\ \text{times}'\, x'\, y' &= \text{unit}_A\, \{\, x \times y \mid x \in x',\, y \in y' \,\} \\ \text{equals}'\, x'\, y' &= \text{unit}_A\, \{\, x \mid x \in x',\, y \in y',\, x = y \,\} \end{aligned}$$

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

$$\lambda n' \to (\text{unit}_A\, n' \overset{A}{>\!\!>\!\!=} \text{pred}' \overset{A}{>\!\!>\!\!=} \iota' \overset{A}{>\!\!>\!\!=} \text{amb}' \overset{A}{>\!\!>\!\!=}$$

$$\lambda x' \to (\text{unit}_A\, x' \overset{A}{>\!\!>\!\!=} \iota' \overset{A}{>\!\!>\!\!=} \text{amb}' \overset{A}{>\!\!>\!\!=}$$

$$(\text{times}'\, x') \overset{A}{>\!\!>\!\!=} (\text{equals}'\, n') \overset{A}{>\!\!>\!\!=} \text{some}'))$$

where n' is of type $\{a\}$. This was a lot of work, but we have reaped a tangible benefit, namely, non-deterministic comprehensions can now be rendered as *dataflow graphs*. In Figure 2 (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 $(\ >\!\!>\!\!=_A)$. Junctions correspond to values of common subexpressions bound to variable names introduced by λ–expressions. Lastly, (A) is amb' and (S) is $some'$. This result is important because, without the amenity (provided by all general purpose programming languages) of being able to define and name functions, comprehension syntax quickly becomes unwieldy. For this reason, we make extensive use of dataflow graphs as a visual programming language in the remainder of this paper.

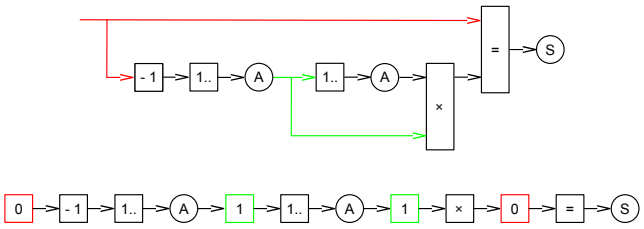

Figure 2: *Non-deterministic dataflow graph* for deciding primality (top). Dataflow graph compiled into a sequence of *non-deterministic combinators* (bottom).

From Dataflow Graphs to Combinators

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*. These would contain the environments needed to lookup the values bound to variable names introduced by λ–expressions. Happily, this turns out to be unnecessary. In this section we 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.

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

$$\to \boxed{f''} \to \ :: \quad [\{a\}] \to \langle[\{a\}]\rangle.$$

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

$$f''\ (x' : s'') = \text{map}_A\ (:\ s'')\ (f'\, x')$$

where stack s'' is of type $[\{a\}]$, map_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:

$$g''\ (x' : y' : s'') = \text{map}_A\ (:\ s'')\ (g'\, x'\, y').$$

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 non-deterministic computation has yielded a satisfactory intermediate result (whether that is something or nothing) and fail otherwise:

$$\text{some}''\ (x' : s'') = \begin{cases} \langle\rangle & \text{if } x' = \{\} \\ \text{unit}_A\ s'' & \text{otherwise} \end{cases}$$

$$\text{none}''\ (x' : s'') = \begin{cases} \text{unit}_A\ s'' & \text{if } x' = \{\} \\ \langle\,\rangle & \text{otherwise.} \end{cases}$$

Multiple functions can be applied to a single value by pushing copies of the value onto the top of the stack and then applying the functions to the copies. This preserves the value 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

$$\text{x}''_k\ (s'') = \text{unit}_A\ ((s''\ !!\ (n-k)) : s'')$$

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 for deciding primality into a sequence of combinators

$$\text{x}''_0 \overset{A}{>\!\!=\!\!>} \text{pred}'' \overset{A}{>\!\!=\!\!>} \iota'' \overset{A}{>\!\!=\!\!>} \text{amb}'' \overset{A}{>\!\!=\!\!>} \text{x}''_1 \overset{A}{>\!\!=\!\!>} \iota'' \overset{A}{>\!\!=\!\!>} \text{amb}''$$
$$\overset{A}{>\!\!=\!\!>} \text{x}''_1 \overset{A}{>\!\!=\!\!>} \text{times}'' \overset{A}{>\!\!=\!\!>} \text{x}''_0 \overset{A}{>\!\!=\!\!>} \text{equals}'' \overset{A}{>\!\!=\!\!>} \text{some}''$$

where $(>\!\!=\!\!>)$ is *Kleisli composition*

$$f\ >\!\!=\!\!>\ g = (>\!\!>\!\!=\ g) \cdot f$$

In Figure 2 (bottom) boxes are functions with type signatures matching f''. Arrows connecting pairs of boxes are instances of $(>\!\!=\!\!>_A)$. Lastly, Ⓐ is *amb*$''$ and Ⓢ is *some*$''$.

Reified Actor Comprehensions

The last two sections of the paper demonstrated that: 1) Non-deterministic 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 specifying behaviors manifested by reified actors in a virtual world. All results from prior sections apply. However, non-determinism must be combined with other 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 behaviors composed of combinators can report the time they consume. The boxes of dataflow graphs with one and two inputs now have types $\{Actor\} \to \langle\{Actor\}\rangle'$ and $\{Actor\} \to \{Actor\} \to \langle\{Actor\}\rangle'$ where $\langle\,\rangle'$ is the type constructor of monad R. Arrows connecting boxes are instances of $(>\!\!>\!\!=_R)$. Combinators now have type $[\{Actor\}] \to \langle[\{Actor\}]\rangle'$ and are composed with $(>\!\!=\!\!>_R)$.

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 $\boxed{\hat{}} \to \boxed{@}$. 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 $\boxed{\hat{}} \to \boxed{@} \to$ Ⓐ addresses a single sibling while the subgraph $\boxed{\#} \to$ Ⓝ fails if the actor has a neighbor.

Table 1: Unary generators.

Name	Abbrev.	Definition
hands	|	actor sharing hand with x
nexts	$>$	actor with directed bond from x
prevs	$<$	actor with directed bond to x
bonds	:	union of hands, nexts and prevs
neighbors	#	actors in neighborhood of x
contents	@	actors that are contained in x
parents	^	actor that contains x
members	*	members of group of x
others	+	members of group of x but not x

Table 2: Unary guards.

Name	Abbrev.	Definition
amb	A	non-deterministic choice
some	S	Fail if empty.
none	N	Fail if non-empty.

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.

Table 3: Binary relations.

Name	Abbrev.	Definition
same	$=$	set intersection
different	$!=$	set difference
similar	$\sim$	all x type equivalent to some y
dissimilar	$!\sim$	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 grey 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.

Where data dependencies determine order of execution, this order is followed. Where it would otherwise be underdetermined, two devices are introduced to specify execution

Table 4: Unary actions.

Name	Abbrev.	Definition
drop	$!\mid$	Delete hand of x.
unbond	$!>$	Delete directed bond from x.
unbond′	$!<$	Delete directed bond to x.
quit	$*-\!>$	Remove x from its group.
exit	$@-\!>$	Place x inside its parent's parent.
digest	$>!>$	Reduce x to primitive combinators.
on	$/$	Replace combinator with behavior.
off	$\backslash$	Replace behavior with combinator.

Table 5: Binary actions.

Name	Abbrev.	Definition
grab	$\mid$	Create hand between x and y.
bond	$>$	Create directed bond from x to y.
bond′	$<$	Create directed bond from y to x.
join	$-\!>*$	x joins group of y.
eat	$-\!>@$	Place x inside y.
compose	$>\!=\!>$	Replace x with $x\ >\!=\!>_R y$.
swap	$\%$	x and y swap positions and bonds.

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 dependency that determines execution order. Second, execution order can be explicitly specified using dotted *control lines*.

In addition to non-determinism and mutable threaded state, instances of monad R also possess a data logging ability that is used to instrument combinators so that behaviors comprised 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 logged time by one when they are run. Significantly, this occurs on all branches of the non-deterministic computation until a branch succeeds so that the full cost of simulating non-determinism on a (presumed) deterministic substrate by means of backtracking is accounted for. Two kinds of combinators increase 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., join*, pay an additional time penalty proportional to the product of the actor's mass and the L_1 distance moved.

Ideally, the actor model described in this paper would be reified as an ACA so that self-replicating programs consume real physical resources. Actors in an embedded group might share a single processor or might jointly occupy a 2D area of

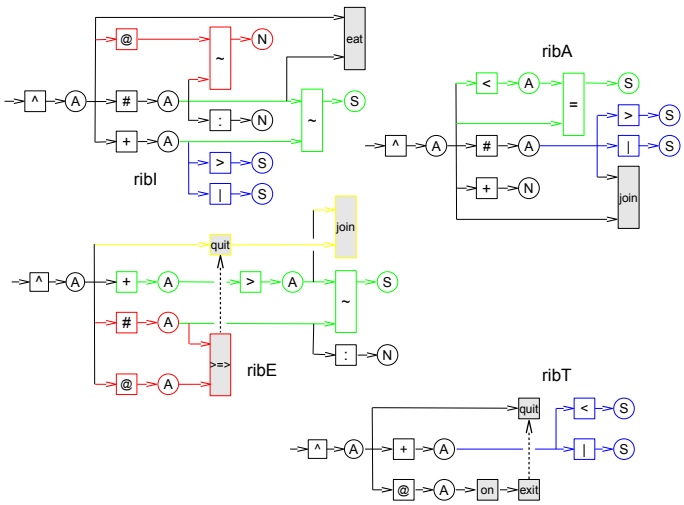

Figure 3: Behaviors defining a ribosome.

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 reified actor model as an event driven simulation using a priority queue (Gillespie, 1977). Event times are modeled as Poisson processes associated with embedded groups and event rates are consistent with the joint 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 $f_D(t) = D\,e^{-Dt/(ms)}/(m\,s)$ where m is mass, s is distance, and D is the ratio of the time needed to execute one primitive operation and the time needed to transport a unit mass a unit distance. As such, it defines the relative cost of computation and data transport in the ACA substrate.[5]

When an *action event* is at the front of the queue, a behavior is chosen at random from among all actors of type behavior in the group. After the behavior is executed, 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 behavior.

Computational Ribosomes

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. Comprehensions can be compiled into sequences of primitive combinators and reified in analogous ways. A

[5] In all of our experiments D equals 10.

plasmid is a compiled comprehension reified as a chain of actors of type combinator linked with directed bonds:

$$P = [\![c_0]\!]^- > [\![c_1]\!]^- > \cdots > [\![c_{N-1}]\!]^-$$

where $(>)$ is a directed bond and $[\![\ ]\!]$ denotes an actor that is reified at the root level. A single undirected bond (not shown) closes the chain and marks the plasmid's *origin*. While plasmids are spatially distributed chains of many actors, *enzymes* are single actors of type behavior:

$$E = [\![c_0 >\!\Rightarrow_R c_1 >\!\Rightarrow_R \cdots >\!\Rightarrow_R c_{N-1}]\!]^+.$$

Biological ribosomes are arguably the most important component of the fundamental dogma (Watson and Crick, 1953). They translate messenger RNA into polypeptides using a four stage process of *association*, *initiation*, *elongation* and *termination*. We can construct a *computational ribosome* that will translate plasmids into enzymes by defining four behaviors with analogous functions (Figure 3), reifying the behaviors as enzymes, and placing them inside an actor of type object

$$R = [\![E_{\text{ribA}}, E_{\text{ribI}}, E_{\text{ribE}}, E_{\text{ribT}}]\!]_0.$$

Behavior *ribA* first checks to see if R possesses a self-directed bond.[6] If so, *ribA* attaches R to the plasmid by adding it to the group of the initial combinator, $[\![c_0]\!]^-$. Next, *ribI* finds an actor in the neighborhood with type matching $[\![c_0]\!]^-$ and places it inside R. When R is at position n on the plasmid, *ribE* finds a neighbor with type matching $[\![c_{n+1}]\!]^-$ and composes it with the combinator inside R, *i.e.,* with $[c_0 >\!\Rightarrow_R \cdots >\!\Rightarrow_R c_n]^-$. It then advances the position of R to $n+1$. This process continues until R reaches $N-1$, at which point *ribT* promotes the combinator to a behavior, expels it, and detaches R from the plasmid.

If a ribosome and a plasmid are placed in the world with a supply of primitive combinators, the ribosome manufactures the enzyme described by the plasmid

$$R + P_b + \textstyle\sum_C m_b(c)\,[\![c]\!]^- \rightarrow R + P_b + E_b$$

where C is the set of 42 primitive combinators and $m_b(c)$ is the number of combinators of type c in P_b and E_b, *i.e.,* the plasmid and enzyme reifications of behavior b.

Now that we have a ribosome, we need something to do with it. We could (of course) use ribosomes to synthesize the enzymes of which they themselves are comprised. However, it would be more interesting if these enzymes were then used to construct additional ribosomes. To accomplish this, we need a 'machine' that will collect the finished enzymes and place them inside an object of the correct type. We call this machine a *factory*. Factories are copiers of *compositional information*, which is heritable information distinct from the *genetic information* that ribosomes translate into enzymes. A factory can be constructed by reifying the behaviors defined in Figure 4 as enzymes and placing them inside an object with a type distinct from that of ribosomes:

$$F = [\![E_{\text{facA}}, E_{\text{facB}}, E_{\text{facY}}, E_{\text{facZ}}, E_{\text{facZ}'}]\!]_1.$$

Behavior *facA* creates a directed bond with any unbonded non-empty object it finds in the factory's neighborhood. This object and its contents serve as the *model*. Behavior *facB* creates a second directed bond from the factory to an empty object with type matching the model. This object serves as the container for the *product.* Behavior *facY* moves behaviors from the neighborhood similar to those in the model into the product. Behavior *facZ* recognizes when the product contains the full set of behaviors and deletes the bond connecting it to the factory. Behavior *facZ′* does the same but also installs a self-directed bond on ribosomes that enables their association behaviors (elements unique to *facZ′* are yellow in Figure 4).

As an initial experiment, we demonstrate mutual replication of a mixed population of ribosomes and factories. Plasmids P_b encoding enzymes E_b comprising ribosomes and factories are placed in a 2D virtual world consisting of 64×64 sites together with a large surplus of ribosomes ($r = 64$) and single instances of factories with ribosome and factory models, F_R and F_F. The supply of primitive combinators and empty objects is replenished as instances are incorporated into enzymes and products. Consequently, the concentration of consumables is held constant. Plasmids and consumables required for synthesis of factory enzymes are overrepresented relative to those for ribosomal enzymes:

$$r\,R + F_R + F_F + \textstyle\sum_B P_b + 2[\![\,]\!]_0 + 3[\![\,]\!]_1 + \sum_B \sum_C m_b(c)\,[\![c]\!]^-$$
$$\rightarrow (r+1)\,R + 2\,F_R + 2\,F_F + \textstyle\sum_B P_b$$

where the multiset $B = \{\, b \mid E_b \in 2R \cup 3F \,\}$. We observe that the ribosomes synthesize the enzymes encoded by the plasmids and these are then used by the factories to construct additional ribosomes and factories. See Figure 5.

Conclusion

Fifty years after von Neumann described his automaton, it remains a paragon of non-biological life. The rules governing CAs are simple and physical, and partly for this reason, the automaton von Neumann constructed using them is uniquely impressive in its semantic closure. Yet perhaps because RASPs are (in comparison with CAs) relatively well-appointed 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. The field of programming languages has made remarkable advances in the years since von Neumann conceived his automaton. Modern functional programming

[6] Ribosomes without this bond are disabled and serve solely as *models* for *factories*, *i.e.,* as compositional information.

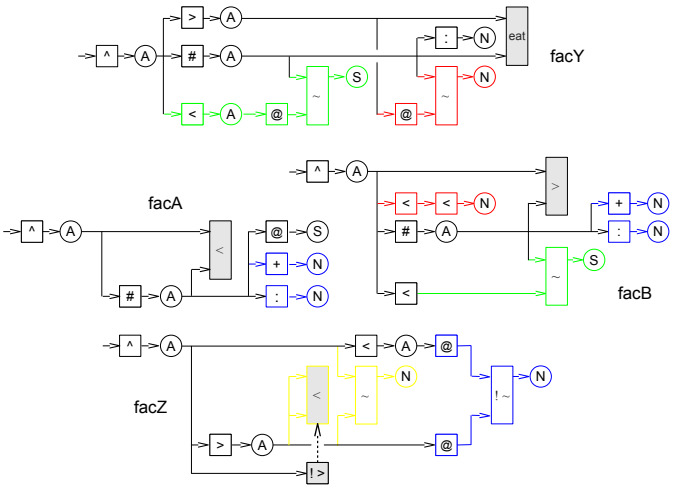

Figure 4: Behaviors defining a factory.

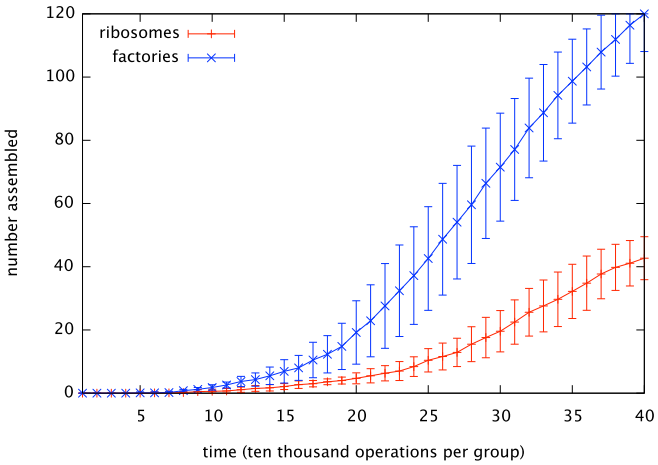

Figure 5: Average increase in numbers of ribosomes and factories (ten runs). Error bars show ± one standard deviation.

languages like Haskell bear little resemblance to the machine languages that are native to RASPs. In this paper, we have attempted to show that programs defined using seemingly exotic constructs like non-deterministic comprehensions can in fact be compiled into sequences of combinators with simple, well-defined semantics. Moreover, because they do not have address operands, these combinators can be reified in a virtual world as actors of only a few fixed types. This makes it possible to build programs that build programs from components delivered by diffusion using processes that resemble chemistry as much as computation.

Acknowledgements

Special thanks to Joe Collard. Thanks also to Dave Ackley, Stephen Harding, Barry McMullin and Darko Stefanovic.

References

Ackley, D. (2013). Bespoke physics for living technology. *Artificial Life*, 34:381–392.

Adami, C., Brown, C. T., and Kellogg, W. (1994). Evolutionary learning in the 2D artificial life system "Avida". In *Artificial Life IV*, pages 377–381. MIT Press.

Berman, P. and Simon, J. (1988). Investigations of fault-tolerant networks of computers. In *STOC*, pages 66–77.

Berry, G. and Boudol, G. (1990). The chemical abstract machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 81–94, New York, NY, USA. ACM.

Felleisen, M. (1990). On the expressive power of programming languages. In *ESOP'90*, pages 134–151. Springer.

Fontana, W. and Buss, L. W. (1999). What would be conserved if the tape were played twice? In Cowan, G. A., Pines, D., and Meltzer, D., editors, *Complexity*, pages 223–244. Perseus Books, Cambridge, MA, USA.

Gillespie, D. T. (1977). Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361.

Hutton, T. J. (2004). A functional self-reproducing cell in a two-dimensional artificial chemistry. In *Proc. of the 9th Intl. Conf. on the Simulation and Synthesis of Living Systems (ALIFE9)*, pages 444–449.

Laing, R. A. (1977). Automaton models of reproduction by self-inspection. *Journal of Theoretical Biology*, 66(1):437–456.

McCarthy, J. (1963). A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland.

Nakamura, K. (1974). Asynchronous cellular automata and their computational ability. *System Comput. Controls*, 15(5):56–66.

Nehaniv, C. L. (2004). Asynchronous automata networks can emulate any synchronous automata network. *IJAC*, 14(5-6):719–739.

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

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

Ray, T. S. (1994). An evolutionary approach to synthetic biology, Zen and the art of creating life. *Artificial Life*, 1:179–209.

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

Wadler, P. (1990). Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, NY, USA. ACM.

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

Williams, L. (2014). Self-replicating distributed virtual machines. In *Proc. of the 14th Intl. Conf. on the Simulation and Synthesis of Living Systems (ALIFE14)*, pages 711–718.