Prebiotic Functional Programs: Endogenous Selection in an Artificial Chemistry

Devansh Vimal[1], Cole Mathis[1,2], Westley Weimer[3], and Stephanie Forrest[1,4,5]

[1]Biodesign Institute, Arizona State University, Tempe AZ, USA
[2]School of Complex Adaptive Systems, Arizona State University, Tempe AZ, USA
[3]Electrical Engineering and Computer Science, University of Michigan, Ann Arbor MI, USA
[4]School of Computing and Augmented Intelligence, Arizona State University, Tempe AZ, USA
[5]Santa Fe Institute, Santa Fe NM, USA
cole.mathis@asu.edu

Abstract

Artificial chemistry simulations produce many intriguing emergent behaviors, but they are often difficult to steer or control. This paper proposes a method for steering the dynamics of a classic artificial chemistry model, known as AlChemy (Algorithmic Chemistry), which is based on untyped lambda calculus. Our approach leverages features that are endogenous to AlChemy without constructing an explicit external fitness function or building learning into the dynamics. We demonstrate the approach by synthesizing non-trivial lambda functions, such as Church addition and succession, from simple primitives. The results provide insight into the possibility of endogenous selection in diverse systems such as autocatalytic chemical networks and software systems.

Introduction

Living systems emerged from unconstrained abiotic mixtures of molecules (Smith and Morowitz, 2016). The field of prebiotic chemistry aims to understand how nonspecific and reactive abiotic compounds self-organize into the restricted subsets of chemical space that are characteristic of life (Muchowska et al., 2020). Understanding and predicting chemical reactivity is a notoriously difficult problem, and even the best available models are only capable of simulating picoseconds to nanosecond time scales in detail (Zhang et al., 2024). Artificial chemistry models represent a coarse-grained approach that do not model the full detail of chemistry but still enable exploration of self-organization in reactive mixtures (Banzhaf and Yamamoto, 2015). These models, like other artificial life (ALife) models, can exhibit emergent phenomena that are non-trivial consequences of the microscopic dynamics. However, they are typically difficult to control or steer, as the relationship between model features and model behavior is, itself, complex. This has contributed to significant debate as to whether the self-organized entities that emerge from these models can serve as a substrate for evolution and selection in a rigorous sense (Vasas et al., 2010, 2012; Danger et al., 2020).

In contrast to artificial chemistry, evolutionary computation (EC) models rely on an externally defined, often static, fitness function to guide behavior, and they typically lack open-ended evolution Miikkulainen and Forrest (2021). Although EC has been used to discover rules for ALife systems that lead to desired emergent behaviors, e.g. (Mitchell et al., 1994; Hecker and Moses, 2015), and some ALife systems build adaptive mechanisms in to the system, e.g. (Ray, 1993; Lenski et al., 1999), ALife systems are typically not easily steerable once the underlying dynamics have been defined. Here, we demonstrate how to guide a classic artificial chemistry model called AlChemy (Fontana and Buss, 1994b,a, 1996) without altering its internal dynamics. Our mechanism uses features endogenous to the dynamics of the model while allowing high-level specification of desired emergent outcomes.

AlChemy (short for Algorithmic Chemistry) is an artificial chemistry model that models key features of chemistry as a formal computational system known as the *lambda calculus*. In brief, AlChemy represents molecules as *lambda expressions*, which can be thought of as simple computer programs. Pairs of these programs interact by using one program as input to another (called a collision), yielding a new program through their combination. This process is analogous to chemical reactions among molecules. We hypothesize that this approach can be extended to synthesize algorithmically useful programs via an implicit steering mechanism, which endogenously guides the bottom-up dynamics of the system to select target 'molecules' (i.e., lambda expressions). We achieve this by adding carefully-designed lambda functions to the AlChemy soup, which function as reagents for the simulation. These functions (1) encode the desired behavior of the target molecules as unit tests (Runeson, 2006), and (2) output copies of successful functions to the simulation. Thus, the dynamics of AlChemy are guided endogenously by the properties of these functions.

A key contribution of the paper is the design of an effective *amplifier* function. Its presence in the simulation steers the dynamics such that (1) molecules that pass the tests (i.e., exhibit desired behavior) are amplified, while (2) the tests are persist in the population until desired molecules emerge. This allows the system to select for the desired functional properties without specifying the precise program

('molecule') ahead of time. Formally, an amplifier function is a second-order function that, when applied to a candidate lambda function, returns duplicates of the candidate function if it passes the test, but returns itself (the amplifier) if the test fails. This structure has the effect of amplifying the abundance of functions that pass the embedded test (in the case of success) or preserving the test for future interactions (in the case of failure). Critically, an amplifier is itself a lambda expression, and can thus be added to the AlChemy state without further manipulation of the simulation.

This approach can be thought of as a simple form of program synthesis, in which the programmer specifies desired behavior via unit tests, and the AlChemy system produces a program that passes those tests. It can also be thought of as an automated approach to chemical synthesis that monitors the system in real time, and when a key observable is detected (the unit tests are passed) the solution is purified to increase the concentration of the desired product (Asche et al., 2021). Alternatively, one can think of this approach as a version of Dynamic Combinatorial Chemistry (DCC) for functional programs. Although the reactions here are not thermodynamically controlled or strictly reversible, the design principle is the same: one designs an experiment that will amplify a desired target rather than designing the target itself (Lehn and Eliseev, 2001). We demonstrate the effectiveness of our approach with test functions that successfully select and amplify arithmetic functions from an initial population of four primitive lambda calculus expressions.

Background and Related Work

AlChemy AlChemy was introduced by Fontana and Buss in 1992 to study how higher-level organizations might spontaneously emerge from basic molecular interactions. They refer to this as the *existence problem* of population genetics (Fontana and Buss, 1994a,b). AlChemy abstracts the details of chemistry and uses lambda expressions to represent molecules. Lambda expressions are computational functions written in the notation of the lambda calculus. Function application represents molecular interaction, and lambda reduction (see below) models chemical reactions. All functions are reduced to normal form using a process that is well-defined in the lambda calculus. However, this reduction process is not guaranteed to terminate. To avoid runaway reductions, AlChemy terminates the reaction after a fixed number of reduction operations or a maximum memory allocation. Here we use a threshold of 8000 reduction steps and a limit of 1000 vertices on the syntax tree represented by the expression. The dynamics of an AlChemy simulation proceed as follows: 1) Generate a set of N randomly generated lambda expressions, called a *soup*; 2) Select two expressions at random from the soup, call them A and B; 3) Apply A to B, and attempt to reduce to normal form within a constant number of reductions; 4) If (3) succeeds, add the newly generated expression to the soup and remove a different, randomly chosen, expression from the soup, and otherwise discard the irreducible expression (this guarantees a constant size population); 5) Iterate steps 2–4 T times. Each function application and attempted reduction is called a collision. Time is measured by the number of collisions that have occurred.

Fontana & Buss showed that in the long time limit ($T \gg N$), these dynamics can produce a set of mutually dependent lambda expressions, such that the application of any two expressions in the set yields another expression in the set (Fontana and Buss, 1994a). These sets are analogous to autocatalytic sets of molecules. Fontana & Buss identified a hierarchy of these collectively reproducing sets, which included selfish replicators, small hypercycles, and complex ecosystems (Fontana and Buss, 1994b). We recently revisited the AlChemy simulation, focusing on the stability and diversity of the emergent organizations (autocatalytic networks) using the original code base (Mathis et al., 2024). Here, we show how to steer the AlChemy dynamics to produce useful programs, using a testing approach that relies on the functional nature of AlChemy.

The Lambda Calculus The lambda calculus is a formal system of computation (Church, 1985). It specifies interactions between pure functions, i.e., functions that have no internal state. We summarize it here; see (Barendregt et al., 1984) for a rigorous treatment. The lambda calculus operates on structures called lambda expressions defined over a finite alphabet Σ. Expressions take one of three forms:

1. Single variables, x, chosen from Σ.
2. Lambda abstractions of the form $\lambda x.\ E$, where x is a variable, and E is a lambda expression. This form describes a function with argument x, binding all other x-variables within its body E, which are not already bound to another abstraction.
3. Applications of the form $(E_1\ E_2)$, where E_1 and E_2 are expressions. This form describes the composition of objects. If E_1 is a function, then E_2 is passed to E_1 as an argument.

The lambda calculus also defines two transformation operations, β-reduction and α-renaming, and a deterministic strategy for applying these operations to any lambda expression. Both operations are rewriting rules that rewrite the given lambda expression into another lambda expression. They can be iteratively applied until the lambda expression is in a form such that no rewrites are possible. This is called a *normal form*. Once a lambda expression is in normal form, it has terminated its computation. Because the lambda calculus is Turing complete, there exist lambda expressions that never reduce to a normal form—that is, the deterministic reduction process may never terminate.

The β-reduction operation takes as input expressions of the form $(\lambda x.\ E_1)\ E_2$. The input expression is rewritten by substituting each copy of x in E_1 with a copy of E_2. The

Function	Lambda Expression
`true`	$\lambda a.\ \lambda b.\ a$
`false`	$\lambda a.\ \lambda b.\ b$
`and`	$\lambda a.\ \lambda b.\ a\ b\ a$
`or`	$\lambda a.\ \lambda b.\ a\ a\ b$
`not`	$\lambda a.\ a$ `false true`
`zero`	$\lambda a.\ \lambda b.\ b$
`one`	$\lambda a.\ \lambda b.\ a\ b$
`two`	$\lambda a.\ \lambda b.\ a\ (a\ b)$
The integer n	$\lambda a.\ \lambda b.\ (a\ (a\ \ldots\ (a\ b)))$, where a is applied to b n times
`S`	$\lambda x.\ \lambda y.\ \lambda z.\ x\ z\ (y\ z)$
`K`	$\lambda x.\ \lambda y.\ x$
`I` (identity function)	$\lambda x.\ x$
`P`	$\lambda s.\ \lambda a.\ \lambda b.\ a\ s\ b$
`scc`	$\lambda n.\ \lambda a.\ \lambda b.\ a\ (n\ a\ b)$
`add`	$\lambda n.\ \lambda m.\ n$ `scc` m
`add2`	$\lambda n.$ `scc (scc` n`)`

Table 1: Key lambda expressions used in this work.

λx and E_2 in the original expression are then removed. This operation is analogous to a function call in a modern programming language: $\lambda x.E_1$ is the function, and E_2 is the argument passed to the function. The α-renaming operation rewrites all variables within the scope of a lambda expression, substituting for each another variable chosen from Σ, unless a variable is already bound to another variable, in which case it is unchanged. This operation is analogous to renaming a variable in an arithmetic expression. For example, the x variable in the expression $\lambda x.\ \lambda y.\ x$ can be α-renamed to z, producing $\lambda z.\ \lambda y.\ z$.

If an expression is not in normal form, then the leftmost, outermost reducible expression is β-reduced first. {During a reduction, the substituting expression may have a variable that conflicts with an existing bound variable within the same scope (for example, the first y within the reducible expression $(\lambda x.y\ x)\ y$). In this case, we apply α-renaming to the variable in the substituted expression to preserve the semantic meaning of the expression before and after reduction. Two expressions are said to be *exactly isomorphic* if they have the same syntactic structure and the same semantic structure under α-renaming.

Lambda Notation Common functions in the lambda calculus used in this work are shown in Table 1. A lambda expression with no free (i.e., externally-defined) variables is said to be in closed form and is called a *combinator*; all of the expressions in Table 1 are combinators. The boolean literals `true` ($\lambda a.\ \lambda b.\ a$) and `false` ($\lambda a.\ \lambda b.\ b$) are functions that, by convention, take as input two arguments and return their first and second argument respectively. Propositional connectives (i.e., `and`, `or`, `not`) have the standard semantics. Numbers are encoded using the standard Church numeral representation (Barendregt et al., 1984). A Church numeral n is a function of two arguments. It encodes n applications of its first argument to its second argument.

Certain combinators are defined in the literature. The S, K and I (identity function) combinators, taken together, are universal, i.e., they form a Turing-complete system, and any computation can be expressed as combinations of them. We also describe the P combinator, which permutes its arguments. Our initial AlChemy population (*soup*) consists only of copies of S, K, I combinators, except when we are amplifying addition, where we also include the P combinator in the soup. In this paper, we focus on evolving three functions that manipulate numbers. Each function takes in one or more Church numerals and returns a single Church numeral:

1. *Successor* (`scc`), a function that increments its argument by 1: $\lambda n.\ \lambda a.\ \lambda b.\ a\ (n\ a\ b)$
2. *Add-two* (`add2`), a function that increments its argument by 2, by applying `scc` twice: $\lambda n.$ `scc (scc` n`)`
3. *Addition* (`add`), a function that takes in two arguments, n and m, and applies `scc` to m exactly n times thus producing the sum $n + m$: $\lambda n.\ \lambda m.\ n$ `scc` m.

We focus on these functions because they encode nontrivial computation, have testable semantics, and show how the discovery of one function (`succ`) can serve as a stepping stone to `add`. For example, just as addition can be implemented as n applications of successor m, so multiplication can be accomplished by applying `add n` m times to 0: `mul` = $\lambda n.\ \lambda m.\ m$ (`add` n) 0.

Related Work One approach to designing ALife models with desired properties is to use EC to evolve the low-level rules, as in Mitchell and Crutchfield's famous work on evolving cellular automata (CA) rules (Mitchell et al., 1994). This approach relies on an explicit fitness function (e.g., how many cells are in a given state) after some number of iterations of the CA, and the discovered rules determine the CA's dynamics. Other approaches evolve a rule set or adjust weights while the system runs, e.g., learning classifier systems (Holland et al., 1986), Echo (Hraber et al., 1997), and countless reinforcement learning algorithms dating back to (Samuel, 1959; Sutton and Barto, 2005). In contrast, the dynamics of AlChemy do not explicitly include a learning step, relying instead only on deterministic interactions between lambda expressions. Similarly Hillis's classic work on co-evolving parasites and sorting networks also relies on embedding an evolutionary dynamic directly in the system (Hillis, 1990), by introducing the idea of two competing populations, each evolving against the other in what is known as *competitive co-evolution* Angeline and Pollack

(1993). Although its goal is quite different and the context is an EC with external fitness and genetic operators, *lexicase selection* Spector (2012) resembles the amplifier function presented below. The algorithm selects parents for the next generation by assessing performance on a randomly chosen subset of test cases that define the problem domain.

Program synthesis discovers (synthesizes) programs that satisfy a stated specification (Gulwani et al., 2017). Traditional program synthesis often uses logical approaches (such as constraint solving or deduction) or search-based methods. These have led to the successful synthesis of non-trivial program kernels (e.g., dynamic programming algorithms) from formal specifications (Srivastava et al., 2010). Program synthesis has long been goal in software engineering (Manna and Waldinger, 1971), and it has enjoyed renewed interest with the advent of large language models (Austin et al., 2021). By contrast, our approach focuses on the use of amplifier test functions, which more closely mimics familiar unit tests (Runeson, 2006).

Several other ALife approaches synthesize or evolve program code. Tierra and Avida explored the endogenous generation of programs, but differ from ours by relying on competition among individual programs and built-in self-reproduction (Lenski et al., 1999) making these systems more similar to artificial organisms, in contrast to the more distributed artificial chemistry system used here, where autocatalytic reproduction emerges spontaneously from the dynamics of the system and the reagents that we add to the system (Mathis et al., 2024). Similarly, Agüera y Arcas et al. (2024) studies the rise of self-replicators in minimal programming languages but uses a Turing Machine based model rather than the functional approach we take here. Finally, Genetic Programming and more recent variants use EC to generate software in the context of an external fitness function (Koza, 1999; Weimer et al., 2010).

Implementation

One contribution of this paper is the reimplementation of Fontana and Buss's AlChemy system in a modern programming language (Rust)[1]. The original implementation was written in an early version of C and has certain incompatibilities with modern compilers. Our implementation supports more replicates of runs and larger simulations, e.g., the runs in this paper contain up to 10^6 expressions running for up to 10^8 collision steps. Small-scale experiments (100 soups or fewer) in this work were run on an AMD Ryzen 7840U CPU with 32 GiB of memory, and large-scale experiments were run on the ASU *Sol* supercomputer (Jennewein et al., 2023). Run parameters were determined from preliminary experiments and the results reported in (Mathis et al., 2024).

[1] `https://github.com/AgentElement/functional-supercollider`

Amplifier Functions

The cornerstone of our approach are *amplifier test functions*: generic second-order functions that duplicate functions that pass a series of unit tests. Given a unit test, the amplifier function wraps it such that application of the amplifier to a function f will return copies of f if f passes the test. Otherwise, if the test fails, the amplifier will return itself. We refer to the number of copies of f as the *amplification factor*. In the success case, the amplifier test amplifies the number of good functions in the population. In the failure case, by returning itself, the amplifier test maintains its own population.

We illustrate the amplifier design with an example. Suppose our goal is to produce a soup that contains a significant proportion of addition functions, e.g., enough for easy detection. The programmer creates a unit test for addition, called `test`, such as `(= (f 2 3) 5)`. An amplifier test function `atf` for that `test` is:

$$\texttt{atf} = \lambda f.\,(\texttt{test}\ \ f)\ f\ \texttt{atf}$$

This amplifier test function returns itself (`atf`) if $(\texttt{test}\ \ f)$ is false, and returns its argument (f) if $(\texttt{test}\ \ f)$ is true. In this example, only one copy of f is returned (the amplification factor is 1). [2]. Note that the amplifier test functions do *not* generate the desirable function. Rather, they amplify desirable functions when they appear in the soup and collide with the amplifier function.

In AlChemy, amplifier functions are mostly inert because they are large—often the largest expressions in the soup. When used as arguments to other functions, they frequently fail to reduce to normal form. The pragmatic reduction of AlChemy (reduce up to T steps, and if the expression is still reducible or exceeds a certain memory limit, then fail the reaction) means that most collisions fail if the test function is used as the argument. As a time-saving optimization, we can eagerly fail collisions that have amplifier test functions as arguments if we know they will not reduce in T steps.

Amplifier test functions with high amplification factors can amplify very rare spontaneous events. For example, the spontaneous evolution of addition from a soup of only S, K, I and P combinators is a rare event. In our experiments, an amplification factor of 100 suffices to ensure that a successful addition function remains in the soup available for more collisions with test functions. However, high amplification factors can lead to the proliferation and eventual domination of unwanted *trickster functions*: functions that pass the test cases without actually behaving like the desired function (see Section).

[2] As an explicit example, we obtain the amplifier test function $\texttt{atf} = \lambda f.\,(=\,(f\ 2\ 3)\ 5)\ f\ \texttt{atf}$, upon substituting the expression $(=\,(f\ 2\ 3)\ 5)$ for `test`

Experiment: Amplifier Functions

The first experiment demonstrates that adding amplifier functions to soups does amplify desirable functions, as expected. The probability of an amplifier function colliding with a desired function is proportional to the concentrations of amplifier functions and target functions in the soup. If there are too few amplifier functions, then they are unlikely to randomly collide with the desired function. Thus, the initial emergence of desirable functions is fragile because good functions can collide with other functions, transforming them into undesirable functions, just as desired molecular products might react with other molecules to form undesirable products.

In this initial experiment, the target function is not expected to emerge spontaneously. Instead we initialize the soup with small quantities of the target function (between 0.002% and 10%) and the experiment asks whether the amplifier test functions actually amplify a target in the face of competing background reactions. For these experiments, the successor function is the target.

We measure the effectiveness of the amplifier tests by measuring the quantity of functions in the soup that are exactly isomorphic to a pre-written implementation of the desired function. For example, to measure the effectiveness of the successor test function, we measure the concentration of functions in the soup that are exactly isomorphic to the expression $\lambda n.\ \lambda a.\ \lambda b.\ a\ (n\ a\ b)$. Because this is not the only possible implementation of successor, this measurement is conservative and thus provides a lower bound on the effectiveness of amplification. Since it is undecidable to determine if two lambda expressions exhibit identical behavior on all inputs, an approximation of this kind is necessary.

Figure 1 plots the results as a function of both the initial number of successor functions and the percentage of test functions. The horizontal axis plots the prevalence of test functions (as a fraction of the population) as the soup evolves and the vertical axis shows how many target functions (successor) were artificially introduced into the soup (as a fraction of the population). The color indicates the percentage of soups with at least 20% successor functions (out of 100 soups run). We found that introducing as few as 5% amplifier functions substantially amplified the presence of the successor function in the soup. On the other hand, when the amplifier test functions were not included, no runs achieved a population of successor functions that exceeded 20%.

The amplification of good functions is largely independent of the fraction of test functions introduced into the soup. Desirable functions are more likely to be amplified by the test functions than undesirable functions, even if there are only a small number of them present in the system. If the initial population contains more than 0.16% successor functions and at least 10% amplifier test functions, then after 10^6 collisions, the population of successor functions exceeds

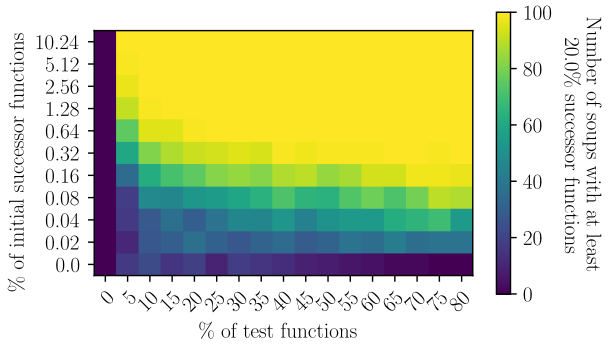

Figure 1: Amplification of the successor function under varying initial conditions. For each pixel in the heatmap, 100 soups containing 5000 expressions are run for 10^6 collisions. We measure the number of soups containing at least 20% successor functions at the end of their runs. Each soup is initialized with a small fraction of successor functions (y-axis), and a large fraction of test functions (x-axis). Bright yellow indicates that all soups have a large quantity of successor functions, dark blue indicates that none do.

20% in one half of the runs. However, without the amplifier test functions there is no amplification and no soup developed a population of successor functions above 20%. Thus, once a target function appears in the soup, the amplifier test succeeds and produces enough copies to facilitate easy detection.

Experiments: Targeted Selection

Having established that a target function can be amplified, we next investigate whether the dynamics of AlChemy can generate target functions endogenously. We focus on generating and amplifying target functions from soups that do not initially contain the target. Some of these functions are building blocks that can be used to synthesize the final result, e.g., the successor function is a building block for the addition function. In the following, we study this building block and two other target functions: addition and add-two. The successor function is a key building block for many arithmetic functions, simple add-two is an example of a function that very rarely emerges spontaneously, and addition is a non-trivial extension of successor that almost never emerges spontaneously.

In all three experiments, we measure the population of target and amplifier functions every 1000 collisions. We perturb the soup with new amplifier functions every 10^5 collisions to replenish their supply. This is necessary because the dynamics of AlChemy randomly remove a function from the soup after each successful collision, thus diluting amplifier functions as the simulation proceeds. Each run lasts 10^6 collisions, and initially contains between 5000 and 6000 expressions, depending on the number of unit tests. These pa-

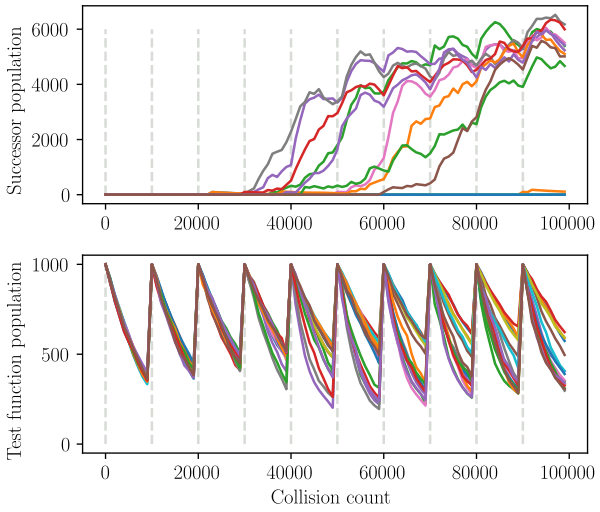

Figure 2: Sixteen representative runs testing for and amplifying **successor**. (Top) Population of `scc` in the soup over time. Different colors correspond to different experimental runs with the same initial conditions. (Bottom) The population of test functions over time. Every 10^5 collisions (gray dashed lines), we introduce new amplifier functions to replenish those lost by random deletion and replacement. Each run is seeded with universal combinators and amplifier functions only.

rameters were chosen to match our available computational resources.

Successor In the first small-scale experiment, we show that successor emerges reliably from a seed mixture of S, K and I expressions in equal quantities, plus amplifiers for successor. The amplifier functions each take a function as an argument, apply the function to a fixed integer $0 \leq n \leq 20$, and check the result for equality with its successor: $n' = n + 1$. These amplifiers take the form $\lambda f.\ (=\ \ (f\ n)\ n')$. The $(=\ \ (f\ n)\ n')$ term acts as a test case. Figure 2, shows the populations of `scc` and amplifiers over time for sixteen runs. Each run begins with an initial population of 6000 functions (5000 seeds and 1000 amplifiers).

Add-two Next, in a larger experiment of 1000 runs, we show that add-two emerges reliably. The initial condition for this experiment is a soup that contains a seed mixture of S, K, and I expressions in equal quantities plus amplifiers for add-two. The amplifiers for add-two take a function as an argument, apply this function to a fixed integer $0 \leq n \leq 20$, and check the result for equality with $n' = n + 2$. In the top panel of Figure 3, we plot the population of `add2` over time for 1000 runs. Each run begins with an initial population of 5000 functions (15% are amplifier functions, the remainder are seed functions).

Addition Finally, we investigate the emergence of addition in an experiment of 1000 runs. The initial condition is a seed mixture of S, K, I and P expressions in equal quantities, plus an equal quantity of two amplifiers, one for addition and one for successor. The amplifiers for addition each take a function as argument, apply this function to a fixed pair of integers $0 \leq n, m \leq 20$, and check the result for equality with their sum $\sigma = n + m$. The tests take the form $\lambda f.\ (=\ \ (f\ n\ m)\ \sigma)$. In the bottom panel of Figure 3, we plot the population of `add` over time for 1000 runs. Each run begins with an initial population of 5000 functions (15% amplifier functions, and the rest seed functions).

In principle, the P combinator is not strictly necessary for the evolution of addition, as the S, K, and I expressions are universal. However, in another experiment with 1000 soups seeded only with S, K, and I expressions, we observed the spontaneous emergence of the addition function only once (not shown). By random chance, the function was not amplified and diluted out of the soup. Using the P combinator increases the likelihood of generating addition because it produces addition when composed with `scc`. We hypothesize that the P combinator would not be needed in larger-scale versions of these experiments, an experiment we leave for future work. Including P accounts for the fact that many more soups yielded addition in this experiment than add-two in the previous experiment.

Sensitivity We also evaluate three different initial populations, and test each population for the spontaneous emergence of the add and successor function, with and without tests. This set of experiments is summarized in Table **??**. We find that random inputs never generate good functions, unless the amplifier test is added and even then, rarely and only for the successor function.

If we seed the soup with expressions that are randomly generated by the binary tree method of (Mathis et al., 2024) instead of using S, K, I then the probability of evolving any integer function is too small to reliably measure. Across over 1000 runs, the successor function evolved fewer than five times from random expressions. This suggests that randomly-generated expressions may either derail the spontaneous emergence of successor, or produce more tricksters that dominate the soup.

Trickster Functions

The amplifier functions sometimes amplify a function that displays partially correct behavior but does not match the developer's intent. Because the function passes at least one test, and because each amplifier function encodes exactly one unit test, it may be amplified and persist in the simulation. We call such functions *trickster functions*. Of course,

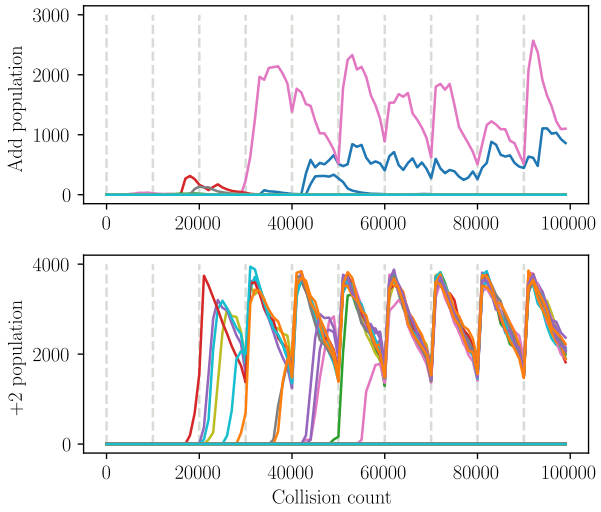

Figure 3: 1000 representative runs targeting **addition** and **add-two**. (Top) Population of `add` in the soup over time. (Bottom) Population of `add2` in the soup over time. Each set of runs was seeded with combinators and amplifier functions only. Different colors correspond to different experimental runs with identical initial conditions. The population of test functions over time is plotted in each panel. Every 10^5 collisions (gray dashed lines), amplifier functions are added to replenish those lost by random deletion and replacement. Each run is seeded with universal combinators, `P`, and amplifier functions.

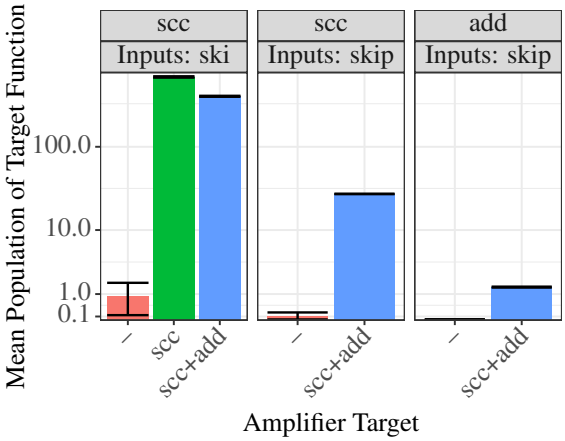

Figure 4: Discovery of numeric functions under different initial conditions. Each panel shows a unique combination of initial inputs and target functions. The horizontal axis indicates which test functions were included in the amplifier set. The *Inputs* row specifies the functions seeded into the soup at the start (either random functions, `SKI`, or `SKIP`). Bar height represents the time-averaged population of the function indicated at the top of the panel. Values are averaged across 10^3 independent runs. "–" denotes cases where no amplifier functions were added. Cases with *random* input functions are not shown as those yielded mean populations of less than 10^{-5}.

the complete set of unit tests provided to the system might also be inadequate to test exhaustively for the desired functionality in the target function. Writing a complete test suite is a classic problem in software engineering and programming languages. While testing can exhaustively check behavior for functions with small finite domains (e.g., `and` or `or`), in the general case of unbounded inputs or data structures, no finite test suite will suffice, as is the case for the functions explored here.

We developed three heuristics for mitigating trickster functions:

1. When the test applies k arguments to the candidate function, pass only those with at least k leading abstractions (i.e., have arity k or greater).
2. When the test applies k arguments to the candidate function, pass only functions with bodies that reference those first k arguments (i.e., fail functions that do not actually use their arguments).
3. Discard functions that are structurally identical to `true` or `false` wrapped in extra lambda abstractions.

These heuristics are not universally applicable—for example, #3 would not be appropriate if we were trying to find a function of three arguments that ignores those three arguments and always returns zero—but in practice, we found them to be efficient, and applied them to all experiments reported here.

Discussion and Conclusion

The amplifier test function amplifies functions that might otherwise be fragile and unlikely to persist in the population, enabling implicit selection for functionality in AlChemy. Such functions can be building blocks for synthesizing a desired final result or the final result itself. Most functions that do not replicate themselves are fragile, including those required for many theoretical models, including Turing Machines and Peano Arithmetic) (Mathis et al., 2024). For example, addition function returns a number and not a copy of the addition function, while a sorting function returns a sorted list and not a copy of itself. Indeed, small functions that exactly replicate themselves (called *quines* in the literature), are notoriously rare and hard to construct.

The amplifier design introduces implicit selective pressure into the bottom-up artificial life simulation dynamics in a manner that is analogous to the approach in *dynamic combinatorial chemistry* (DCC) (Lehn and Eliseev, 2001). In DCC, as in our system, a dynamic library of molecules is generated that rapidly inter-convert. Key functionality can be selected out of this system by introducing a protein (analogous to our amplifier) that selectively binds to desirable

functional groups (analogous to our unit tests). This binding shifts the equilibrium of the internal dynamics toward the formation of the bound molecule, leading to amplification.

Limitations We propose an *amplification* solution, not a *generation* solution. Indeed, a key difficulty in amplifying the 'add' function was the fact that it rarely emerged spontaneously from our initial conditions. Additionally, AlChemy, as implemented here, uses the untyped lambda calculus. Some synthesis tasks are likely to be more efficient in the typed lambda calculus. Such an extension would limit collisions to occur only between compatible elements, and are comparable to orthogonal components in synthetic approaches, e.g., with Click Chemistry (Paioti et al., 2024).

The emergence and selection of trickster functions in our system is unsurprising from a software engineering perspective, but it provides an interesting correspondence to synthetic chemistry. The key challenge in synthetic chemistry is identifying reagents and reaction conditions that enable the chemist to produce some desired products (and hopefully many different ones) while preventing the system from producing undesirable side-products. This is difficult because reactivity is fundamentally determined by the tendency (or not) of electrons to transition between orbitals. Making some orbitals accessible while preventing all others requires conditions that are exquisitely fine-tuned (Szabo and Ostlund, 1996). In the case of our amplifier functions, we need to fine tune the tests using ever more diverse and precise test suites, just as chemists need to fine tune reaction conditions. We expect the spontaneous emergence of more complex functions in AlChemy to be rare for the same reason that we expect total synthesis of complex organics to be difficult: the desired function itself will be more rare, and the proportion of functions that are nearly identical to the desired function will grow as the complexity of the function grows (Marshall et al., 2021). Identifying approaches to overcome this challenge could provide insights for both program synthesis and prebiotic chemistry.

Future directions Although the amplifier described here was constructed using only the native lambda calculus representation of AlChemy, the design may generalize to other representations. In particular, we are interested in using a typed representation in an AlChemy-like system based on a functional programming language, such as Haskell, OCaml or TypeScript. This would allow us to investigate whether a bottom-up system such as AlChemy can generate useful functions, and compare those implementations to large human coded standard libraries. For example, if the system were initialized with existing functions, would other interesting functions emerge? Ultimately, such an approach could provide a new approach to program synthesis, an active area of research in computer science (Gulwani et al., 2017).

Our experiments focused on functions that emerge spontaneously in a constant size soup through combination and reduction processes in AlChemy. An interesting question is how the size of the soup interacts with the amplifier functions, and how the length of the simulation affects results. We expect that larger system sizes would enable more distinct expressions to exist in the system at any given time, possibly accelerating the rate of discovery. However, the rate of amplification might change with system size, suggesting an interplay between efficiency and diversity. We suspect the broad features of this trade-off may be amenable to a mean-field analysis.

The AlChemy system is an exciting approach to artificial chemistry because it retains some essential aspects of real chemical systems (Mathis et al., 2024; Fontana and Buss, 1996). Using the approach outlined here we can now explore how the system responds to selective pressures that can be imposed through an endogenous mechanism. Our approach may enable future artificial chemistry approaches to more closely mirror the experimental procedures of modern prebiotic chemistry (Surman et al., 2019). We plan to further investigate how AlChemy can be used to make predictions about long-term chemical experiments and guide experimental design by developing the correspondence between AlChemy and real reaction platforms (Asche et al., 2021).

Conclusion This work addresses a long-standing limitation of conventional evolutionary algorithms, which typically rely on externally defined objective functions that return a numerical fitness value, or small set of values, for every individual in the population. This limitation prevents evolutionary computation from achieving the generative open-ended evolution that we observe in the natural world Miikkulainen and Forrest (2021). Similarly, prebiotic chemical experiments often exhibit diverse forms of self-organization, but open-ended evolution of those forms is limited because they lack genetic systems (Szathmáry, 2006). This work provides insights into how more elaborate molecules might be produced through the step-wise selection of simpler molecules, if those simpler molecules can be amplified or made to persist longer against background reactions (Asche et al., 2021; Surman et al., 2019) Ultimately, we aspire to develop systems that combine the self-organizing features of artificial chemistry, with EC to drive the emergence of novel functionality in chemical and software systems.

Acknowledgments

We thank Kirtus Leyba, Pemma Reiter, Joe Renzullo, and the anonymous reviewers for their careful corrections and thoughtful suggestions. SF gratefully acknowledges the partial support of NSF (CCF CCF2211750), DARPA

(N6600120C4020), ARPA-H (SP4701-23-C-0074), and the Santa Fe Institute.

References

Agüera y Arcas, B., Alakuijala, J., Evans, J., Laurie, B., Mordvintsev, A., Niklasson, E., Randazzo, E., and Versari, L. (2024). Computational life: How well-formed, self-replicating programs emerge from simple interaction.

Angeline, P. J. and Pollack, J. B. (1993)). Competitive environments evolve better solutions for complex tasks. In *Proceedings of the Fifth International Conference on Genetic Algorithms*.

Asche, S., Cooper, G. J., Keenan, G., Mathis, C., and Cronin, L. (2021). A robotic prebiotic chemist probes long term reactions of complexifying mixtures. *Nature communications*, 12(1):3547.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. (2021). Program synthesis with large language models.

Banzhaf, W. and Yamamoto, L. (2015). *Artificial chemistries*. MIT Press.

Barendregt, H. P. et al. (1984). *The lambda calculus*, volume 3. North-Holland Amsterdam.

Church, A. (1985). *The calculi of lambda-conversion*. Number 6. Princeton University Press.

Danger, G., d'Hendecourt, L. L. S., and Pascal, R. (2020). On the conditions for mimicking natural selection in chemical systems. *Nature Reviews Chemistry*, 4(2):102–109.

Fontana, W. and Buss, L. W. (1994a). What would be conserved if" the tape were played twice"? *Proceedings of the National Academy of Sciences*, 91(2):757–761.

Fontana, W. and Buss, L. W. (1994b). "the arrival of the fittest": Toward a theory of biological organization. *Bulletin of Mathematical Biology*, 56:1–64.

Fontana, W. and Buss, L. W. (1996). The barrier of objects: From dynamical systems to bounded organizations.

Gulwani, S., Polozov, O., and Singh, R. (2017). Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119.

Hecker, J. P. and Moses, M. E. (2015). Beyond pheromones: evolving error-tolerant, flexible, and scalable ant-inspired robot swarms. *Swarm Intelligence*, 9:43–70.

Hillis, W. D. (1990). Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D: Nonlinear Phenomena*, 42(1-3):228–234.

Holland, J. H., Holyoak, K. J., Nisbett, R. E., and Thagard, P. R. (1986). *Induction: processes of inference, learning, and discovery*. MIT Press, Cambridge, MA, USA.

Hraber, P. T., Jones, T. L., and Forrest, S. (1997). The ecology of echo. *Artificial Life*, 3:165–190.

Jennewein, D. M., Lee, J., Kurtz, C., Dizon, W., Shaeffer, I., Chapman, A., Chiquete, A., Burks, J., Carlson, A., Mason, N., et al. (2023). The sol supercomputer at arizona state university. In *Practice and Experience in Advanced Research Computing 2023: Computing for the Common Good*, pages 296–301.

Koza, J. R. (1999). *Genetic programming III: Darwinian invention and problem solving*, volume 3. Morgan Kaufmann.

Lehn, J.-M. and Eliseev, A. V. (2001). Dynamic combinatorial chemistry. *Science*, 291(5512):2331–2332.

Lenski, R. E., Ofria, C., Collier, T. C., and Adami, C. (1999). Genome complexity, robustness and genetic interactions in digital organisms. *Nature*, 400(6745):661–664.

Manna, Z. and Waldinger, R. J. (1971). Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165.

Marshall, S. M., Mathis, C., Carrick, E., Keenan, G., Cooper, G. J., Graham, H., Craven, M., Gromski, P. S., Moore, D. G., Walker, S. I., et al. (2021). Identifying molecules as biosignatures with assembly theory and mass spectrometry. *Nature communications*, 12(1):3033.

Mathis, C., Patel, D., Weimer, W., and Forrest, S. (2024). Self-organization in computation and chemistry: Return to alchemy. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 34(9).

Miikkulainen, R. and Forrest, S. (2021). A biological perspective on evolutionary computation. *Nature Machine Intelligence*, 3(1):9–15.

Mitchell, M., Crutchfield, J. P., and Hraber, P. T. (1994). Evolving cellular automata to perform computations: mechanisms and impediments. *Physica D: Nonlinear Phenomena*, 75(1):361–391.

Muchowska, K. B., Varma, S. J., and Moran, J. (2020). Nonenzymatic metabolic reactions and life's origins. *Chemical Reviews*, 120(15):7708–7744.

Paioti, P. H., Lounsbury, K. E., Romiti, F., Formica, M., Bauer, V., Zandonella, C., Hackey, M. E., Del Pozo, J., and Hoveyda, A. H. (2024). Click processes orthogonal to cuaac and sufex forge selectively modifiable fluorescent linkers. *Nature Chemistry*, 16(3):426–436.

Ray, T. S. (1993). An evolutionary approach to synthetic biology: Zen and the art of creating life. *Artificial Life*, 1(1_2):179–209.

Runeson, P. (2006). A survey of unit testing practices. *IEEE software*, 23(4):22–29.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229.

Smith, E. and Morowitz, H. J. (2016). *The origin and nature of life on earth: the emergence of the fourth geosphere*. Cambridge University Press.

Spector, L. (2012). Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 401–408.

Srivastava, S., Gulwani, S., and Foster, J. S. (2010). From program verification to program synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 313–326, New York, NY, USA. Association for Computing Machinery.

Surman, A. J., Rodriguez-Garcia, M., Abul-Haija, Y. M., Cooper, G. J., Gromski, P. S., Turk-MacLeod, R., Mullin, M., Mathis, C., Walker, S. I., and Cronin, L. (2019). Environmental control programs the emergence of distinct functional ensembles from unconstrained chemical reactions. *Proceedings of the National Academy of Sciences*, 116(12):5387–5392.

Sutton, R. S. and Barto, A. G. (2005). Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks*, 16:285–286.

Szabo, A. and Ostlund, N. S. (1996). *Modern quantum chemistry: introduction to advanced electronic structure theory*. Courier Corporation.

Szathmáry, E. (2006). The origin of replicators and reproducers. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 361(1474):1761–1776.

Vasas, V., Fernando, C., Santos, M., Kauffman, S., and Szathmáry, E. (2012). Evolution before genes. *Biology direct*, 7:1–14.

Vasas, V., Szathmáry, E., and Santos, M. (2010). Lack of evolvability in self-sustaining autocatalytic networks constraints metabolism-first scenarios for the origin of life. *Proceedings of the National Academy of Sciences*, 107(4):1470–1475.

Weimer, W., Forrest, S., Le Goues, C., and Nguyen, T. (2010). Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116.

Zhang, S., Makoś, M. Z., Jadrich, R. B., Kraka, E., Barros, K., Nebgen, B. T., Tretiak, S., Isayev, O., Lubbers, N., Messerly, R. A., et al. (2024). Exploring the frontiers of condensed-phase chemistry with a general reactive machine learning potential. *Nature Chemistry*, 16(5):727–734.