1 Introduction

The Darwinian principle of adaptation through replication, heritable variation, and selection is not limited to a population of biological entities. It is applicable to any object that can be copied and varied, and for which at least some of the variants are distinguishable (by an arbitrary criterion called "fitness"). Genetic Programming (GP) applies the logical structure of the Darwinian scheme to computer programs, searching for those that compute a desired function. In the realm of computation the "phenotype" corresponds to the *behavior* of a program (i.e., the graph of a function in the set-theoretic sense) and the "genotype" corresponds to *that which behaves* (i.e., a syntactical construct expressing a function).

Genetic Programming [Koza, 1992] has indeed become a powerful machine learning method. Its success has been demonstrated in a variety of applications, such as solving symbolic regression problems [Koza, 1992], discovering game-playing strategies [Koza, 1992, Angeline, 1993], inducing decision trees [Koza, 1992], generating controllers (e.g., for robots) [Koza, 1992, Reynolds, 1994, Spencer, 1994, Handley, 1994], cracking and evolving randomizers [Koza, 1992, Jannink, 1994].

A question, however, remains: When is the Darwinian process effective? The search for an answer is still a frontier in biology. The question is hard, because replication and variation apply to the carrier of behavior ("genotype"), while selection applies to the behavior ("phenotype"), and, at least in biology, we don't sufficiently understand the mapping between the two.

GP applications typically start with a cleverly crafted representation of a problem domain from which suitable high-level primitives are inferred. It may be argued that in such cases GP plays a minor role in finding the solution compared to the contribution of the user [Abbott, 1993, Taylor, 1993]. The point relevant here is that the problem-specific components of applications make a rigorous and general theoretical exploration of GP nearly impossible [O'Reilly and Oppacher, 1994].

An analysis is needed of the intrinsic constraints and opportunities of GP deriving from the *specific language* that maps syntactical constructs (genotypes) into functional behaviors (phenotypes). Such an analysis could begin by turning to an abstract universe of functions that is (i) transparent, (ii) sufficiently formal to encourage mathematical analysis, and (iii) canonical (i.e., it should capture a programming language paradigm). One such universe is λ-calculus, invented by Alonzo Church [Church, 1932, Church, 1933] to study the properties of functions. As is well known, λ-calculus is the syntactically unsugared core of functional programming languages. The notion of λ-definability is equivalent to Turing's notion of computability and the Herbrand-Gödel notion of general recursiveness. An exploration of the characteristic features of the *"λ-calculus landscape"*, that is, the program-to-function mapping in λ-calculus, would greatly help in framing what is possible and what is not with GP within the paradigm of functional programming.

This, then, is the motivation for our use of λ-calculus. The present paper by no means characterizes the features of the λ-calculus landscape. That is a long-term challenge. Our contribution merely consists in illustrating GP within this semantically elegant and minimalist framework, raising a few questions, and providing support for the claim that the study of the λ-calculus landscape holds promise for a more rigorous and systematic understanding of GP.

As an example we attempt to evolve the predecessor function. Church had just about convinced himself that there is no λ-definition of the predecessor function, when Kleene found a representation for it [Kleene, 1981, Revesz, 1988]. It was an important result, because otherwise a computable function would exist that is not λ-definable. Although the predecessor function seems difficult to find, there is no particular practical benefit in finding it, since it is already known.

Section 2 briefly introduces the λ-calculus. Section 3 introduces genetic operators that naturally fit the syntactical machinery of λ-calculus. In Section 4 the predecessor function is evolved as an example. In Sections 5 and 6 the influence of different parameter settings is discussed. Section 7 concludes, and raises a few issues.

2 λ-calculus

In λ-calculus functions are represented as λ-expressions. The simplest λ-expression is a variable (without type or sort) of which there is an infinite supply. To build more complex λ-expressions there are only two constructions: application and abstraction.

The syntax of a λ-expression (following [Revesz, 1988]) is:

$$\begin{aligned} \langle \lambda\text{-}\mathit{expression} \rangle &::= \langle \mathit{variable} \rangle | \langle \mathit{abstraction} \rangle | \langle \mathit{application} \rangle && (1) \\ \langle \mathit{abstraction} \rangle &::= \lambda \langle \mathit{variable} \rangle . \langle \lambda\text{-}\mathit{expression} \rangle && (2) \\ \langle \mathit{application} \rangle &::= (\langle \lambda\text{-}\mathit{expression} \rangle) \langle \lambda\text{-}\mathit{expression} \rangle && (3) \end{aligned}$$

Abstraction introduces a formal parameter $\langle \mathit{variable} \rangle$ (e.g., x) and turns a given $\langle \lambda\text{-}\mathit{expression} \rangle$ into a unary function. For example, x is a λ-expression by virtue of (1), and abstracting x via (2) yields $\lambda x.x$ which is the identity function I, defined in usual notation as $I(x) = x$. The x after the dot in $\lambda x.\mathbf{x}$ corresponds to the body of the function (i.e., the right hand side in the defining equation $I(x) = \mathbf{x}$). In $\lambda \mathbf{x}.x$ the λ preceding the $\mathbf{x}$ declares it as a formal parameter, corresponding to the left hand side in the equation $I(\mathbf{x}) = x$.

Of course, we could also abstract some other variable, say, y to get $\lambda y.x$. In an expression of the form $\lambda \langle \mathit{variable} \rangle . \langle \lambda\text{-}\mathit{expression} \rangle$ all occurrences of the $\langle \mathit{variable} \rangle$ in $\langle \lambda\text{-}\mathit{expression} \rangle$ are called "bound". A variable that is not bound is termed "free". Names of bound variables don't matter, and we identify expressions that differ only in the names of bound variables.

$\langle \mathit{application} \rangle$ is intended to express the application of an operator (here enclosed in parentheses) to an operand. There is no syntactical distinction between operator and operand, both are arbitrary λ-expressions.

λ-calculus is meant to be a theory of the evaluation of functions. This is achieved by defining a reduction relation between expressions that captures the notion of "substitution":

$$(\lambda x.P)Q \Rightarrow [Q/x]P \tag{4}$$

where P and Q are λ-expressions and x is a variable. $[Q/x]P$ means the substitution of Q for all occurrences of x in P. (We assume unique names for bound variables, distinct from names of free variables.) The situation in equation (4) corresponds to the substitution of the actual parameter Q for the formal parameter x in the body P of a function $\lambda x.P$. The reduced form corresponds to the result of the evaluation. For example, $I(y) \stackrel{\text{def}}{=} (\lambda x.x)y \Rightarrow [y/x]x = y$.

An expression that contains no (sub)-expression to which scheme (4) applies is called a "normal form", and the process of rewriting an expression into normal form is a "normalization". Not every expression does have a normal form. One may encounter an infinite sequence of reductions, corresponding to a non-terminating computation.

Everything computable can be defined with just the syntax (1-3) and the rule (4). Recall, however, that there is no syntactical distinction between functions and arguments. Natural numbers, for example, are functions too, and they can be represented in a variety of ways. We will use the Church numerals here, and henceforth refer to them simply as numerals:

$$\mathbf{0} \stackrel{\text{def}}{=} \lambda f.\lambda x.x$$

$$\mathbf{1} \overset{\text{def}}{=\!=} \lambda f.\lambda x.(f)x$$

$$\mathbf{2} \overset{\text{def}}{=\!=} \lambda f.\lambda x.(f)(f)x$$

$$\mathbf{3} \overset{\text{def}}{=\!=} \lambda f.\lambda x.(f)(f)(f)x$$

In general, the numeral representing the number **n** iterates its first argument to its second argument n times. In the following we use the short-hand notation

$$\mathbf{n} \overset{\text{def}}{=\!=} \lambda f.\lambda x.\underbrace{(f)\ldots(f)}_{n \text{ times}}x \overset{\text{def}}{=\!=} \lambda f.\lambda x.(f)^n x. \tag{5}$$

Note that in (untyped) λ-calculus every expression can act via (4) as a map sending *any* expression into some expression (that may or may not possess a normal form). In recursion theory, however, the notion of function is a map from non-negative integers to non-negative integers. Thus, only the behavior of λ-expressions restricted to a representation of numbers (i.e., numerals) is considered. In other words, arithmetic functions are λ-expressions whose reduced form is a numeral when they are applied to one (or several) numeral(s). λ-expressions that are arithmetic functions don't contain free variables. Such expressions are termed "closed expressions", also known as combinators.

As an example take the successor function, which increments its argument by one: **succ** $\overset{\text{def}}{=\!=} \lambda n.\lambda g.\lambda y.((n)g)(g)y$. Normalizing the application of **succ** to the numeral **2** yields:

$$\begin{aligned}(\mathbf{succ})\mathbf{2} \quad \overset{\text{def}}{=\!=} \quad & \underbrace{(\lambda n.\lambda g.\lambda y.((n)g)(g)y)}_{\mathbf{succ}}\underbrace{\lambda f.\lambda x.(f)(f)x}_{\mathbf{2}} \Rightarrow \\ & \lambda g.\lambda y.((\lambda f.\lambda x.(f)(f)x)g)(g)y \Rightarrow\end{aligned}$$

$$\lambda g.\lambda y.(\lambda x.(g)(g)x)(g)y \Rightarrow \lambda g.\lambda y.(g)(g)(g)y \equiv \mathbf{3}.$$

A more detailed introduction of λ-calculus can be found, for example, in [Revesz, 1988, Hankin, 1994, Barendregt, 1984].

3 Genetic programming in λ-calculus

In GP the task is to find a program with a prespecified behavior in a given language. A *fitness function* is defined to grade the actual behavior of a program with respect to the desired target behavior.

In the present case GP is not operating on conventional programs but on λ-expressions[1]. Since we want to evolve arithmetic functions, we restrict the space of λ-expressions to closed expressions, and we will use simple genetic operators that preserve syntactical legality and closure.

GP starts with a population of randomly generated closed expressions. An expression is chosen for reproduction with a probability proportional to its fitness. The reproduction event produces either an exact copy, a mutant, or a recombinant with another randomly chosen expression. Selection pressure results from constraining a population to a constant number of expressions: each time an expression has been reproduced, another one, chosen randomly, is removed.

[1]Genetic Programming originally operated on LISP programs which are basically λ-expressions cast in user friendly syntax.

3.1 Mutation

According to the grammar (1-3), a mutation should naturally consist in introducing or removing the two expression constructors ⟨*abstraction*⟩ and ⟨*application*⟩. Our mutation operators are inspired by [O'Reilly and Oppacher, 1994], and are motivated by minimizing the syntactical change of a λ-expression upon mutation while preserving closure of the expression.

An abstraction of a *new* variable can be inserted before any (sub-)expression. Similarly, any unused abstraction (i.e., an abstraction that does not bind a variable) can be deleted. For example, $\lambda x_0.x_0$ and $\lambda x_0.\lambda x_1.x_0$ can be transformed into one another by insertion and deletion of $\lambda x_1.$.

The insertion of an application requires two steps. First, a (sub-)expression is chosen randomly and is determined (randomly) to become the operator or the operand in the application to be created. Second, the missing operand or operator expression has to be generated. This happens according to the following scheme: let V be the set of variables that are bound at the point where the missing expression has to be inserted. The missing expression is either a randomly chosen variable in V or, if $V = \emptyset$, the identity function $\lambda x.x$. For example, $\lambda x_0.x_0$ can be mutated into $\lambda x_0.(x_0)x_0$ or $(\lambda x_0.x_0)\lambda x.x$ by insertion of an application. Similarly, an application can be removed by erasing either the operator or the operand expression, but only if the expression to be erased does not contain an application (i.e., if it is of the form $\lambda x_1.\lambda x_2.\cdots\lambda x_n.x$, where x may or may not be one of the x_i) For example, $\lambda x_0.\underline{(}x_0\underline{)\lambda x_1.x_0}$ can be mutated into $\lambda x_0.x_0$ by deleting the underlined portion of an application.

To summarize, the most basic scheme allows deletions or insertions of the following underlined constructs in the context of any closed expression: (i) $\underline{\lambda\langle x\rangle.}\bullet$, where $\bullet$ stands for an expression, and $\langle x\rangle$ is a variable, (ii) $\underline{(\langle simple\rangle)}\bullet$, and (iii) $\underline{(}\bullet\underline{)\langle simple\rangle}$, where $\langle simple\rangle$ is the identity function or a bound variable[2]. These moves are independent from one another. In our specific case we deviate from the basic moves in two minor ways: we insert a $\lambda x.x$ only if there's no bound variable to insert (this minimizes syntactical change due to insertions), and we also allow the deletion of compound expressions of the form $\lambda x_1.\lambda x_2.\cdots\lambda x_n.x$ (this speeds up things by bundling a number of primitive deletions).

Every combinator (closed expression) can be transformed into any other combinator by a finite number of mutations: in the worst case first delete all applications and abstractions of one combinator until $\lambda x.x$ is left, then proceed by inserting abstractions and applications to yield the wanted combinator.

3.2 Crossover

Crossover of parse trees is usually the exchange of subtrees [Koza, 1992]. In λ-expressions this would correspond to the exchange of sub-expressions. However, the exchange of arbitrary sub-expressions of combinators may lead to free variables in the offspring expressions, thus violating conservation of closure. Therefore, we allow only sub-expressions that are combinators to be chosen for

[2]The substitution of a bound variable by another is possible, but not realized here.

exchange. A combinator is an "encapsulated" unit whose action is independent from the context it is in, and thus could be regarded as a "natural" building block.

3.3 Selection

In a population of n λ-expressions an expression, $expr_i$, is chosen to reproduce with probability

$$p_r(expr_i) = f(expr_i) / \sum_{k=1}^{n} f(expr_k) \tag{6}$$

where $f(expr_i)$ is the fitness of $expr_i$.

The offspring of a λ-expression is added to the population. To induce a selection pressure, we maintain a constant population size by removing another expression from the population [Moran, 1958]. We use two slightly different schemes for removal The so-called *non-elitist* method chooses the expression to be removed with an expression-independent probability

$$p_d(expr_i) = 1/n. \tag{7}$$

The so-called *elitist* method does the same, except that the λ-expression with the currently best fitness is prevented from being removed.

3.4 Fitness function

Here we are interested in the behavior of a λ-expression *expr* when applied to the representation of the number i, $numeral_i$:

$$(expr)numeral_i \Rightarrow result_i, \tag{8}$$

where $result_i$ is the resulting normal form expression.

The *fitness function* grades the expression's behavior, $result_i$ with respect to the desired target behavior, $target_i$. This can be done only for a finite number of (fitness) cases. The fitness function must assign some viability to expressions that are not solutions. This poses a difficulty, as there are two ways in which a λ-expression can "fail" when applied to a numeral: (i) $result_i$ is a numeral, but does not match the desired $target_i$, or (ii) $result_i$ is not a numeral. In fact, case (ii) is typical. A simple, problem-independent way to deal with this situation is to reward a λ-expression to the extent that its actual $result_i$ is syntactically "close" to some numeral. We do by using a regular expression to find the biggest numeral expression num_i that is contained in $result_i$. Then we "punish" the expression *expr* in proportion to the syntactical "junk" that $result_i$ contains in addition to the pattern num_i (see below). Next we use the arithmetic difference between the number represented by num_i and the desired one represented by $target_i$ to assess how much the case i is satisfied. The fitness contributed by the case i, $case$, is simply the product of these two factors. Specifically,

$$case(result_i, target_i) \;=\; \underbrace{\frac{primitives(num_i)}{primitives(result_i)}}_{syntactical\ distance} \cdot \underbrace{\frac{1}{|num_i - target_i| + \epsilon}}_{arithmetic\ distance} \qquad (9)$$

$primitives(\langle \lambda\text{-}expression \rangle)$ denotes the sum of the number of applications, abstractions, and variable occurrences in $\langle \lambda\text{-}expression \rangle$. A numeral num_i representing the number k consists of two abstractions, k applications, and $k+1$ occurrences of variables. Therefore, $primitives(num_i) = 2 \cdot k + 3$. A small constant ϵ avoids division by zero when $num_i = target_i$. The maximum fitness per case is $1/\epsilon$, when $result_i = num_i = target_i$. If the result contains no numeral at all, the fitness case contributes nothing to the overall fitness. The overall fitness for expression $expr$ becomes:

$$f(expr) = \sum_{i=0}^{C} case(result_i, target_i) \qquad (10)$$

where $C + 1$ is the number of fitness cases and $result_i$ is obtained according to (8).

The functions (9) and (10) are applicable to any arithmetic target function, i.e., pairs of natural numbers $(i, target_i), i = 0, 1, \ldots$. The only aspect in which the fitness function does more than that, is in keeping expressions viable that are not functions mapping numerals to numerals, while inducing a selection pressure towards arithmetic functions in general.

4 Example: predecessor function

As mentioned in the introduction, we chose the predecessor function as an example for a λ-expression to be evolved by GP. The predecessor function has a historical meaning in λ-calculus and seems to be difficult to find [Kleene, 1981]. Its behavior is defined as

$$pred(n) = \begin{cases} n-1 & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases}$$

We look for a λ-expression that has the same behavior when applied to Church numerals.

A simplified version of Kleene's representation for the predecessor function can be found in [Revesz, 1988]:

$$\lambda x_1.(((x_1)\lambda x_2.\lambda x_3.((x_3)\lambda x_4.\lambda x_5.(x_4)(((x_2)\lambda x_6.\lambda x_7.x_6)x_4)x_5)(x_2)\lambda x_8.\lambda x_9.x_8)$$
$$\lambda x_{10}.((x_{10})\lambda x_{11}.\lambda x_{12}.x_{12})\lambda x_{13}.\lambda x_{14}.x_{14})\lambda x_{15}.\lambda x_{16}.x_{16}$$

The trick of this expression (not easily intelligible when seeing it as it stands) consists in introducing ordered pairs. Application of this expression to the numeral n generates $n+1$ ordered pairs iteratively, such that the zeroth pair is $[0,0]$ and the i-th pair $(0 < i \leq n)$ is $[i, i-1]$. It is easy to obtain $[i+1, i]$ from $[i, i-1]$. Finally, the second element of the pair $[n, n-1]$ is projected out as the result of the predecessor function.

GP was able to find predecessor functions. Among almost 300 runs, four of them were successful. Many more runs were successful, if the requirement was dropped that *pred*(0) be 0. We will return to this point later.

One of the successful runs turned out 805 different predecessor functions (in normal form) until it was stopped. All the ones we examined are based on the same principle, which is different than Kleene's. We will demonstrate this principle on the shortest predecessor function found (most of them were much longer, with the longest having 116 abstractions).

The expression reads:

$$\lambda x_1.((x_1)\underbrace{\lambda x_2.((x_2)\lambda x_3.x_3)\lambda x_4.\lambda x_5.((x_2)x_4)(x_4)x_5)}_{S}\underbrace{\lambda x_6.(x_1)\lambda x_7.\lambda x_8.\lambda x_9.x_9}_{A},$$

which we may write in abbreviated form as

$$\mathbf{pred} \overset{\text{def}}{=} \lambda x_1.((x_1)S)A$$

Applying this expression to a numeral $\mathbf{n}$, see (5), gives

$$(\mathbf{pred})\mathbf{n} = (\lambda x_1.((x_1)S)A)\lambda f.\lambda x.(f)^n x.$$

The first reduction step produces $((\lambda f.\lambda x.(f)^n x)S)A_n$ with $A_n \equiv \lambda x_6.(\lambda f.\lambda x.(f)^n x)\lambda x_7.\lambda x_8.\lambda x_9.x_9$. Note that A depends on n. The next step gives $(\lambda x.(S)^n x)A_n$, yielding finally

$$(\mathbf{pred})\mathbf{n} = (S)^n A_n.$$

When **pred** is applied to the numeral $\mathbf{n}$, its subexpression S is iterated n times on A_n. When $n = 0$, S is not applied to $A_0 \equiv \lambda x_6.(\lambda f.\lambda x.x)\lambda x_7.\lambda x_8.\lambda x_9.x_9$ at all, and A_0 normalizes to $\lambda x_6.\lambda x.x$ which is the Church numeral for 0 (modulo names of bound variables). Therefore, $(\mathbf{pred})\mathbf{0} = \mathbf{0}$.

In the case of $n > 0$, $A_{n>0} \equiv \lambda x_6.(\lambda f.\lambda x.(f)^n x)\lambda x_7.\lambda x_8.\lambda x_9.x_9$, whose normal form is $A' \overset{\mathrm{def}}{=} \lambda x_6.\lambda x.\lambda x_8.\lambda x_9.x_9$ independently of $n > 0$. When S is applied to A' the result is the numeral representing 0. A similar analysis (left to the reader) reveals that S, when applied to a Church numeral, acts exactly like a successor function! The mechanism of this predecessor function is quite elegant: it applies $n > 0$ times an expression S to an expression A' that is not a numeral. The first application is, therefore, not a successor action, but it happens to return the numeral for zero. The next $n-1$ applications of S simply increment zero to $n-1$, yielding the predecessor of n:

$$(\mathbf{pred})\mathbf{n} = (S)^n A' = \underbrace{(S)...(S)}_{n\ times} A' = \underbrace{(S)...(S)}_{n-1\ times}(S)A' = \underbrace{(S)...(S)}_{n-1\ times}\mathbf{0} \equiv \mathbf{n-1}.$$

Throughout our experiments we used 9 fitness cases (numerals 0 to 8) to test the arithmetic action of a candidate λ-expression, and to evaluate its fitness. Fitness was computed as described above with $\epsilon = 0.1$ (maximum fitness is therefore 90). A population size of 1000 was used. Usually 10^5 to $2 \cdot 10^5$ expressions were generated during a run. Other parameters, such as the mutation rate, were varied in different experiments. In the following we briefly compare these settings.

5 Parameter settings

For each combination of parameter settings 9 runs were performed. This is sufficient for a qualitative assessment, but obviously not for a statistical study.

We briefly summarize our findings.

1. Elitist vs. non-elitist selection. With non-elitist removal (Section 3.3) the temporarily best expression will get lost sometimes. Under these conditions GP never found a solution. In particular, the highest fitness score did either not improve at all or only marginally, even when $1–5 \cdot 10^6$ expressions were generated in the process. In almost all runs the best λ-expression present in the initial population is $\lambda x_1.\lambda x_2.\lambda x_3.x_3$. It corresponds to the constant function $f(x) = 0$, and, thus, solves two fitness cases correctly, as well as returning numerals for the others. During most runs better λ-expressions were generated, but they were lost before they could proliferate sufficiently, despite their probability of being chosen for reproduction (proportional to fitness) is higher than their probability of being removed (independent of fitness).

2. Normal form. When a λ-expression is generated (either randomly at the outset, or by mutation, or by crossover) it is usually not in normal form. A λ-expression can be admitted to the population either "as it is", or it can be normalized. While nothing changes semantically ("phenotypically") during normalization (that is, both the original expression and its normal form compute the same function), the effects of a genetic operation can be considerably widened at the syntactical level. This in turn will affect future variation.

The populations whose expressions were in normal form performed slightly better. An advantage of dealing with normal forms could be the enormous reduction of the search space, because one is moving between equivalence classes of expressions rather than individual expressions. This seems to outweigh the loss of redundancy, which is thought to buffer the disruptive action of crossover [Koza, 1992].

3. Number of λ-expression generated. After 10^5 λ-expression were generated the runs had usually converged, in the sense that no significant further improvement in fitness occurred.

4. Mutation and crossover rates. Only four out of 300 runs found a predecessor function. In one of these four runs the mutation rate was 0.01 and the crossover rate was 0.6 (i.e. 1% of the λ-expressions were generated using mutation, 60% of the λ-expressions were generated using crossover the remaining 39% were duplicated by copying). These rates correspond to those used by Koza [Koza, 1992]. The other successful runs occurred with mutation and crossover rates of 0.2 and 0.4, respectively, as well as 0.3 and 0.3, and 0.4 and 0.6. However, the other eight unsuccessful runs performed with these same settings were not significantly better than most runs performed with other settings.

Experiments with mutation and recombination rates varying from 0 to 0.4 and 0 to 0.8, respectively, were performed, using elitist selection, keeping the expressions in normal form, and generating $2 \cdot 10^5$ λ-expressions. The only clear conclusion from these experiments is that runs where either the mutation or the crossover rate were zero performed significantly worse than runs were both were operative.

These explorations confirm the well-known difficulty of determining "good" parameter settings in Genetic Programming [Kinnear, 1994, p.14].

6 Discussion

With the exception of the four successful runs, almost none found an expression that computed anything close to the predecessor function. In 56% of the runs no expression was found that returned the correct value for at least three out of the nine fitness cases. In only 6% of the runs an expression solved at least four fitness cases. Why were most runs so unsuccessful?

1. Fitness proportional selection without scaling is problematic. Early in a run most λ-expressions have a very low fitness and the few better ones will quickly dominate the population, leading to premature convergence. Later in the run the average fitness is close to the optimal fitness and fitness proportionate selection degenerates to random selection [Goldberg, 1989]. This may be counteracted by playing with scaled fitness or tournament selection [Koza, 1992]. However, our point was not to fine-tune the system.

2. The requirement that $pred(0) = 0$. The nature of the natural numbers requires that the predecessor of zero be zero. Our success rate would have been more impressive without this requirement. Dropping it, resulted in 11% of the runs finding a “predecessor” function, compared to the meager 1.3% that ended with a genuine predecessor complying with $pred(0) = 0$.

The reason for this behavior is not that the “predecessor” functionality without the zero case is so much simpler to realize. As a matter of fact, the distance between a predecessor expression (with $pred(0) = 0$) and a “predecessor” (with $pred(0) = something$) was at times just one mutation. GP had to generate only a few hundred λ-expressions to find a correct predecessor function, when the initial population was seeded with a “predecessor” failing on zero. The point is that when $pred(0)$ had to be zero, the constant function $f(x) = 0$ was a trap (returning always a numeral and satisfying two fitness cases). The population converged to one of the many realizations of this local maximum.

7 Concluding remarks

In this paper we have shown that an exceedingly simple version of GP is able to find an elusive function like the predecessor function. Moreover, once a predecessor function was found, hundreds of expressions with neutral functionality were produced. The mapping from expressions to behaviors on numerals is indeed many-to-one, and seems to possess some clustered structure. It was an interesting aside that the evolved predecessor function works with a different mechanism than Kleene’s version. Yet both mechanisms make use of a successor function as a component. We have not systematically tried to evolve other arithmetic functions, but we can report that GP was successful in finding, for example, the addition operation. We emphasize once more that the GP component used was minimal, implementing only the essential Darwinian scheme, and that it was problem-independent in the restricted sense of being usable for any arithmetic function. The example presented indicates that a computational analysis of the "λ-calculus landscape" is feasible.

What would such an analysis be good for? The issue at the outset was to understand what GP can do. GP's power may derive not so much from the Darwinian search strategy, but rather from the medium in which it operates, i.e., the specific way in which functional action ("semantics") is expressed by syntactical structure in a programming language. What is "easy" for an unsugared GP may simply reflect the characteristic properties and constraints of a syntax and its operational semantics. A programming language is a device mapping a space of symbolic expressions (here λ-expressions) into a space of possible behaviors (here recursive functions). The set of possible expressions or programs is structured by a neighborhood relation. Two programs are neighbors if one is transformed into the other by a single "basic" mutation reflecting the grammatical structure of the language. In λ-calculus, basic mutations are insertions or deletions of the two term constructors "application" and "abstraction" (see Section 3.1). The domain of an untyped λ-expression comprises the set of all possible λ-expressions. Our interest, however, is restricted to the behavior on numerals. On this portion of the domain many expressions will coincide in their behavior; the mapping from λ-expressions to functions (on numerals) is many-to-one. Call a behavior A "accessible" (in one step) from behavior B, if the programs that realize A are *typically* neighbors of the programs that realize B. From this viewpoint, understanding GP means understanding this "accessibility relation".

This raises a number of issues. Given a particular function, how "dense" in program-space are the expressions realizing it? Is there a notion of a "frequently realized" function, as opposed to a "rare" one? What characterizes the most frequent ones? How are their programs distributed in program space? Are programs with identical behavior accessible from one another by a few basic mutations? Do they form networks on which a population could evolve neutrally [Kimura, 1983], thereby exploring program space while not losing the currently best function?

We bluntly take this perspective from a quite different case concerning specific biopolymers: RNA sequences (think genotypes) and their folding into structures (think phenotypes) which determine chemical behavior [Schuster et al., 1994]. In that context it was discovered (i) that there exists a well-defined notion of "frequent" structure, (ii) that almost all frequent structures are realized within a small (compared to sequence length) neighborhood of any random sequence, (iii) and that sequences folding into the same structure form extended connected networks that percolate through sequence space. The implications of these findings for the evolutionary adaptation of populations of RNA molecules are immediate [Huynen et al., 1996]: the process of adaptation by mutation and selection is not suited for finding prespecified "rare" structures in a systematic way. It will find, however, without great difficulty any prespecified "frequent" structure, no matter where the process starts in sequence space.

Although nucleotide sequences and their folded structures are different objects than programs and their behaviors, the RNA example serves to illustrate (in a biologically relevant case) that what the Darwinian process can *characteristically* achieve is tightly constrained by the statistical regularities of the genotype to phenotype mapping. By analogy, a starting point for a theoretical analysis of what GP can do, may be provided by a similar statistical study of a canonical functional landscape, the "λ-calculus landscape". The advantage of this landscape is its definitional transparency and the large body of mathematical theory available on λ-calculus.

References

[Abbott, 1993] Abbott, R. (1993). mailing list genetic-programming@cs.stanford.edu.

[Angeline, 1993] Angeline, P. J. (1993). *Evolutionary Algorithms and Emergent Intelligence*. PhD thesis, The Ohio State University.

[Barendregt, 1984] Barendregt, H. G. (1984). *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. North-

Holland, Amsterdam, second revised edition.

[Church, 1932] Church, A. (1932). A set of postulates for the foundation of logic. *Annals of Math.*, 2s. 33:346–366.

[Church, 1933] Church, A. (1933). A set of postulates for the foundation of logic (second paper). *Annals of Math.*, 2s. 34:839–864.

[Goldberg, 1989] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley.

[Handley, 1994] Handley, S. G. (1994). The automatic generation of plans for a mobile robot via genetic programming with automatically defined functions. In Kinnear, K. E., editor, *Advances in Genetic Programming.* MIT Press, Cambridge, MA.

[Hankin, 1994] Hankin, C. (1994). *Lambda Calculi. A Guide for Computer Scientists.* Clarendon Press, Oxford.

[Huynen et al., 1996] Huynen, M., Stadler, P. F., and Fontana, W. (1996). Smoothness within ruggedness: The role of neutrality in adaptation. *Proc. Natl. Acad. Sci. USA*, 93:397–401.

[Jannink, 1994] Jannink, J. (1994). Cracking and co-evolving randomizers. In Kinnear, K. E., editor, *Advances in Genetic Programming.* MIT Press, Cambridge, MA.

[Kimura, 1983] Kimura, M. (1983). *The Neutral Theory of Molecular Evolution.* Cambridge University Press, Cambridge.

[Kinnear, 1994] Kinnear, K. E., editor (1994). *Advances in Genetic Programming.* MIT Press, Cambridge, MA.

[Kleene, 1981] Kleene, S. C. (1981). Origins of recursive function theory. *Annals of the History of Computing*, 3(1):52 – 67.

[Koza, 1992] Koza, J. R. (1992). *Genetic Programming.* MIT Press, Cambridge, MA.

[Moran, 1958] Moran, P. A. P. (1958). Random processes in genetics. *Proc.Camb.Phil.Soc.*, 54:60–71.

[O'Reilly and Oppacher, 1994] O'Reilly, U.-M. and Oppacher, F. (1994). Program search with a hierarchical variable length representation: Genetic programming, simulated annealing, hill climbing. In Davidor, Y., Schwefel, H., and Manner, R., editors, *Parallel Problem Solving from Nature - PPSN III, Int. Conf. on Evolutionary Computation, Jerusalem, Israel.* Springer Verlag.

[Revesz, 1988] Revesz, G. E. (1988). *Lambda-Calculus, Combinators, and Functional Programming.* Cambridge University Press.

[Reynolds, 1994] Reynolds, C. W. (1994). Evolution of obstacle avoidance behavior: Using noise to promote robust solutions. In Kinnear, K. E., editor, *Advances in Genetic Programming.* MIT Press, Cambridge, MA.

[Schuster et al., 1994] Schuster, P., Fontana, W., Stadler, P. F., and Hofacker, I. (1994). From sequences to shapes and back: A case study in RNA secondary structures. *Proc. Roy. Soc. (London) B*, 255:279–284.

[Spencer, 1994] Spencer, G. (1994). Automatic generation of programs for crawling and walking. In Kinnear, K. E., editor, *Advances in Genetic Programming*. MIT Press, Cambridge, MA.

[Taylor, 1993]

Taylor, S. (1993). mailing list genetic-programming@cs.stanford.edu Message-ID: <199309090140.AA04904@xedoc.xedoc.com.au>.