Evolving Combinators*

Matthias Fuchs
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
67653 Kaiserslautern
Germany
E-mail: `fuchs@informatik.uni-kl.de`

October 28, 1996

Abstract

One of the many abilities that distinguish a mathematician from an automated deduction system is to be able to offer appropriate expressions based on intuition and experience that are substituted for existentially quantified variables so as to simplify the problem at hand substantially. We propose to simulate this ability with a technique called genetic programming for use in automated deduction. We apply this approach to problems of combinatory logic. Our experimental results show that the approach is viable and actually produces very promising results. A comparison with the renowned theorem prover OTTER underlines the achievements.

*This work was supported by the *Deutsche Forschungsgemeinschaft (DFG)*.

1 Introduction

Automated deduction systems have gained remarkable powerfulness in recent years. Nevertheless, they still lack a number of abilities that distinguish a good mathematician. One of these abilities is to use intuition and experience to *suggest* a solution to a problem and then to solve the usually much easier task of proving that it is indeed a correct solution. Since mostly the first suggested solution is not correct, this process may continue over several stages possibly involving modification or re-design of a solution ("*trial and error*"). Although this procedure cannot deny a relation to guesswork and as such should be considered "non-mathematical", it is nevertheless an indispensable and valid tool in the repertoire of a mathematician that—as history has shown—is often the crucial key to solving intricate problems.

Therefore it appears to be profitable to equip an automated deduction system with the ability to suggest solutions and to (attempt to) prove their correctness instead of having the system attempt to construe a correct solution by means of deductive mechanisms. As appealing as this idea sounds, it immediately raises the serious question as to how pivotal concepts such as "intuition" and "experience" that are hardly accessible by a computer can be brought to bear. Since enumerating all solutions or suggesting solutions at random in most cases are impractical or insufficient alternatives, a sensible automated suggestion of solutions seems to be out of reach.

In the area of deduction, suggesting a solution essentially amounts to finding expressions to be substituted for existentially quantified variables so that the validity of the whole formula at hand is not changed. In the majority of cases, after replacing the existentially quantified variables with the "right" expressions, the problem then becomes much simpler which explains the profitableness of this kind of proceeding. Assuming that we are dealing with problems of first-order predicate logic, these expressions can be represented by first-order terms. Thus, basically a search for the "right" first-order terms must be conducted.

Genetic programming (GP, [9]) is a method for searching large spaces of programs in order to find a program that performs some given task satisfactorily. A program is commonly a LISP S-expression that essentially is a first-order term. Hence, GP can be applied to searching for the "right" terms to be substituted for existentially quantified variables. We shall demonstrate the use of GP for this purpose in the area of *combinatory logic* (CL, [1], [3]). More precisely, we shall *evolve* combinators that are to satisfy a certain condition specified by the problem of CL at hand. Our experiments have shown that this at first sight hopeless endeavor is practical and actually produces very promising results as a comparison with a powerful proving system such as OTTER shows.

The report is organized as follows: First, section 2 introduces the basics of CL and the problems addressed here. Then, section 3 outlines GP in the light of this particular application. Section 4 documents the experimental studies. Finally, section 5 concludes the report.

2 Combinatory Logic

In this section we shall present CL from the viewpoint of automated deduction (see also [19], [20], [12], [11]). We assume that the reader is familiar with the basic notions of rewriting systems.

In [1] CL is defined by equational axioms for the combinators S and K:

$$Sxyz = xz(yz) \qquad\qquad Kxy = x$$

Expressions are implicitly left-associated, i.e., $xyz \stackrel{\text{def}}{=} (xy)z$. x, y and z denote variables from the set $\mathcal{V}$ of variables. For use with an automated deduction system, such expressions are transformed into "proper" first-order terms by using a binary function symbol a ("apply"), e.g., the equation for S given above then reads as

$$a(a(a(S,x),y),z) = a(a(x,z),a(y,z)).$$

Besides combinators S and K various other combinators can be studied.

The first kind of problem we are interested in is to find out whether combinators from a given set $\mathcal{B}$ of combinators (a *basis*) can be combined so as to satisfy the equation of a combinator $\Theta \notin \mathcal{B}$. Consider, for instance, $\mathcal{B} = \{B, W\}$, where

$$Bxyz = x(yz) \quad \text{and} \quad Wxy = xyy,$$

and the question is whether there is a combination of Bs and Ws that satisfies the equation of L with $Lxy = x(yy)$. In first-order predicate logic, this question gives rise to the following goal:

$$\exists z\, \forall x,y : zxy = x(yy)\,?$$

An automated deduction system proceeds by negating and Skolemizing the goal and then adding it to the set of axioms originating from the combinators in $\mathcal{B}$. It then attempts to prove the resulting set of equations and inequations to be inconsistent, i.e., for the example the set

$$\begin{array}{rcl} a(a(a(B,x),y),z) & = & a(x,a(y,z)) \\ a(a(W,x),y) & = & a(a(x,y),y) \\ a(a(z,f_1(z)),f_2(z)) & \neq & a(f_1(z),a(f_2(z),f_2(z))) \end{array}$$

must be shown to be inconsistent by deducing $s \neq s$ (s being some term). This is achieved through continuous paramodulation or demodulation steps mainly involving the goal and the resulting derivatives of it. The search spaces created this way can become enormous and quite often exceed the power of automated deduction systems. (Furthermore, additional measures must be taken if we are not only interested in the fact that a combinator exists, but also want to know exactly what it looks like.) Were we, however, simply requested to show that BWB indeed is a combinator[1] satisfying the equation for L, the resulting problem involving the goal

$$\forall x,y : BWBxy = x(yy) \qquad \text{or} \qquad a(a(a(a(B,W),B),c_1),c_2) \neq a(c_1,a(c_2,c_2))$$

[1] Both elementary combinators such as B or W and compound combinators such as BWB are denoted as combinators. Obviously, elementary combinators are a special case of compound combinators.

merely requires a number of rewrite steps, in this case (given in short notation):

$$BWBxy = W(Bx)y = Bxyy = x(yy).$$

If the axioms can be transformed into a convergent (confluent and terminating) system of rewrite rules, a proof of the latter problem simply reduces to computing normal forms. (This is of course not possible for the axioms in the example.)

The second kind of problem we want to address here is to check for the so-called *strong fixed-point property* ([17]; see also [14]). A (compound) combinator Θ satisfies the strong fixed-point property if

$$\forall x : \Theta x = x(\Theta x).$$

Given a basis $\mathcal{B}$, the question is if there exists a combination of (elementary) combinators in $\mathcal{B}$ that satisfies the strong fixed-point property, i.e.,

$$\exists z \, \forall x : zx = x(zx)\,?$$

The existential quantification again results in an enormous growth rate of the search space.

If we are given a combinator Θ and the problem is to prove that Θ actually satisfies the strong fixed-point property, the task is usually much simpler: $\forall x : \Theta x = x(\Theta x)$ is proved by rewriting both Θx and $x(\Theta x)$ to some term t. This usually causes considerably less search effort than a use of paramodulation in the presence of an existentially quantified goal. If the axioms form (or can be completed so that they form) a set of convergent rewrite rules, there is no search at all, and $\forall x : \Theta x = x(\Theta x)$ is actually decidable by computing normal forms. In the case that there is no convergent (in particular no terminating) set of rewrite rules, the search space can nevertheless be minimized—possibly at the expense of losing completeness—by applying the axioms despite non-termination as rewrite rules with the combinator occurring in the left side of a rule (coupled with a restriction of the number of permitted rewrite steps) which complies with the basic idea of these axioms that are to be utilized for evaluating expressions involving combinators. (A complete discussion of these issues is beyond the scope of this report. See, e.g., [16] instead.)

The essential part is that for both kinds of problems it is desirable to find some other (efficient) means to come up with a combinator than search-intensive deductive mechanisms (paramodulation) in order to significantly simplify the problems. The use of GP for this purpose is the topic of the next section.

3 Genetic Programming

GP is an instance of the general *genetic algorithm* (GA) introduced in [8]. We shall therefore concisely describe the fundamentals of the GA before the specifics of GP will be addressed. (See also, e.g., [5] for a comprehensive description of the GA.)

The GA is an adaptive method based on principles known from general genetics and biological evolution. It is very useful for finding (nearly) optimal solutions of problems in many domains. Unlike other search methods, the GA maintains a set of (sub-optimal) solutions, i.e., several points in the search space. In this context, a solution is preferably called an *individual*, and the whole set is referred to as a *population* or *generation*. Usually, the size of the population is fixed. In order to explore the search space, the GA applies so-called *genetic operators* to (a subset of the) individuals of its current population. This way, new individuals can be created and hence new points in the search space can be reached. In order to keep the population size fixed, it must be determined which individuals are to be eliminated in order to make room for the new ones. For this purpose a so-called *fitness measure* is employed which rates the fitness (i.e., the ability to solve the problem at hand) of each individual of the current population. The genetic operators are applied to the fittest individuals, this way producing offspring which then replaces the least fit individuals (Darwinian principle of "*survival of the fittest*" or, in GA terminology, *elite* or *truncate selection*; other modes of selection are also possible).

So, the GA basically proceeds as follows: Starting with a randomly generated initial population, the GA repeats the cycle comprising the rating of all individuals using the fitness measure, applying the genetic operators to (a selection of) the best individuals, and replacing the worst individuals with offspring of the best, until some termination condition is satisfied (e.g., an individual with a satisfactory fitness level has been created).

In the context of GP, an individual commonly corresponds to a LISP S-expression, i.e., a program that is to solve a given task. In our particular case here, an individual is a first-order term (a combinator) composed of combinators from a basis $\mathcal{B}$ and the binary function symbol a. As a matter of fact, the purpose of the λ-calculus in general and CL in particular suggests to consider a combinator to be a (functional) program. The semantics of such a program is defined by the axioms that each specify the effects of the respective combinator in $\mathcal{B}$.

The initial population hence consists of random combinators. A random combinator is a random binary tree with internal nodes labeled by a and leaves labeled by one of the combinators in $\mathcal{B}$. Details of generating random combinators will be discussed in the context of our experiments (section 4).

Since we used exclusively *crossover* as genetic operator, we shall confine ourselves to a description of this operator. ('Reproduction' as given in [9], for instance, is basically realized by the survival of the fittest individuals.) The crossover operator selects two (not necessarily distinct) parents from the pool of $r\%$ best (surviving) individuals of the current generation. In each parent, a sub-tree is chosen at random. Two child individuals are produced by creating copies of the parents and then exchanging the chosen sub-trees. (For more details of this process see [9].)

The motivation and purpose of crossover is to isolate good "partial" solutions and to re-combine them in order to form ever fitter complete solutions. Building blocks (sub-terms in our case) of the fittest individuals persist and have the potential to contribute to an individual showing satisfactory performance. It is this information acquired in the course of the genetic search and in a way stored in the form of individuals that distinguishes GP (the GA in general) from (pure) random search. There are theoretical studies and ample empirical evidence that corroborate this fact (cf. [8], [9]).

The fitness measure Φ determines which individuals will survive and which individuals will be eliminated in order to make room for offspring of the surviving individuals. Φ is pivotal for GP (for the GA in general), exerting "genetic pressure" that guides and strongly influences the search despite omnipresent random effects that are actually indispensable to escape from local optima and to explore new regions of the search space—a distinctive property of GP and the GA in general. But the fitness measure poses serious problems for our application of GP in the area of automated deduction.

Given an individual (a combinator Θ), the only information we have available is that Θ either solves the problem at hand or it does not, or it is not known if it does so because of the undecidability in case the axioms cannot be transformed into a set of convergent rewrite rules. This kind of information is insufficient to produce an effective fitness measure. (E.g., in the initial population mostly none of the individuals solve the problem so that the same fitness would be assigned to all of them, thus making it impossible to discriminate fitter individuals from less fit ones.) In spite of the substantial lack of reliable information on how close a combinator Θ comes to solving the given problem, the fitness measure developed in the sequel nonetheless attempts to estimate the performance of a Θ in a more differentiated way than merely by qualifying a Θ as "solving", "not solving", or "unknown".

Both kinds of problems of CL introduced in section 2 reduce to checking for the validity of an all-quantified equation $s = t$ once a combinator has been substituted for the existentially quantified variable. Such an equation is valid if its sides can be rewritten to identical terms using the given axioms. If this cannot be achieved, a *term difference* δ (related to ideas presented in [4]) may be employed to estimate by how much $s = t$ "missed" validity. In the following, u and v are first-order terms construed with the help of a, $\mathcal{B}$ and $\mathcal{V}$, i.e., $u, v \in Term(\mathcal{F}, \mathcal{V})$ with $\mathcal{F} = \{a\} \cup \mathcal{B}$. Note that a combinator $\Theta \in Term(\mathcal{F})$, i.e., Θ does not contain variables.

$$\delta(u,v) = \begin{cases} 0, & u \equiv v \wedge (u, v \in \mathcal{B} \vee u, v \in \mathcal{V}) \\ \delta(u_1, v_1) + \delta(u_2, v_2), & u \equiv a(u_1, u_2) \wedge v \equiv a(v_1, v_2) \\ |\gamma(u) - \gamma(v)| + 1, & \text{otherwise.} \end{cases}$$

$\gamma(u)$ counts the number of function symbols in $u \in Term(\mathcal{F}, \mathcal{V})$:

$$\gamma(u) = \begin{cases} 1, & u \in \mathcal{B} \vee u \in \mathcal{V} \\ 1 + \gamma(u_1) + \gamma(u_2), & u \equiv a(u_1, u_2). \end{cases}$$

Hence we have $\delta(u, v) = 0$ *if and only if* $u \equiv v$. Essentially, δ identifies disagreeing sub-terms and adds up measures for their disagreement. These measures basically are differences in the number of symbols occurring in disagreeing sub-terms.

The term difference δ is the basis for our fitness measure. Naturally, δ is only one among many sensible possibilities. The simplicity of δ and our experimental results speak in favor of our definition of term difference.

However, using δ to design a fitness measure without any further modifications can cause problems. Consider $\mathcal{B} = \{B, T\}$, where $Bxyz = x(yz)$, $Txy = yx$, and the problem is to find a combinator from $\mathcal{B}$ that satisfies the equation for Q with $Qxyz = y(xz)$.

(Note that the equations for B and T can be transformed into a set of convergent rewrite rules so that there always exist unique normal forms.) When using δ to compare the outcome $x(yz)$ of $Bxyz$ with the desired expression $y(xz)$ we obtain $\delta(x(yz), y(xz)) = 2$. This is about the best result attained by a random combinator. The simplicity of B also practically guarantees that B will be among the random combinators of the initial population (see the method for generating random combinators given in section 4). However, B carries little "genetic" information and hence is not very useful for the search process of GP. Even worse, B will be very likely to cause GP to converge prematurely in that the whole population consists of Bs only. (If B is one parent, and the sub-tree of the other parent Θ chosen for crossover is Θ itself, then another B will be produced as offspring. If two Bs are selected as parents, two more Bs will be produced. The high fitness of B ensures "survival". Note that GP does not eliminate duplicates mainly for efficiency reasons.) Nonetheless we are of course interested in short combinators that solve the problem at hand so that it does not make sense to rigorously disallow short combinators.

We resolve this dilemma by introducing a "minimal structure requirement" that causes a penalty depending on how much the *length* $\mathcal{L}(\Theta)$ of a combinator Θ falls short of a given minimal length l_{min}. $\mathcal{L}(\Theta)$ simply counts the number of symbols of $\mathcal{B}$ occurring in Θ. (We use $\mathcal{L}(\Theta)$ rather than $\gamma(\Theta)$ in order to focus on the essential parts of a Θ; recall that the symbol a was merely introduced to obtain proper first-order terms and hence is only necessary for using common notation.) $\mathcal{L}$ correlates with γ in the following way:

$$\mathcal{L}(\Theta) = \frac{\gamma(\Theta) + 1}{2}$$

The correspondingly modified term difference is

$$\Delta_\Theta(u, v) = \begin{cases} \delta(u, v), & \mathcal{L}(\Theta) \geq l_{min} \\ \delta(u, v) \cdot (l_{min} - \mathcal{L}(\Theta) + 1), & \text{otherwise.} \end{cases}$$

The fitness $\Phi(\Theta)$ of a combinator Θ is now determined as follows: Let $s = t$ be the goal obtained after replacing the existential variable with Θ (i.e., all remaining variables occurring in $s = t$ are implicitly all-quantified; cp. section 2). In case the axioms form a convergent rewrite system we let $\Phi(\Theta) = \Delta_\Theta(s\!\downarrow, t\!\downarrow)$, where $u\!\downarrow$ denotes the (unique) normal form of $u \in Term(\mathcal{F}, \mathcal{V})$ with respect to the axioms. Under these conditions, we have $\Phi(\Theta) = 0$ *if and only if* Θ is a solution.

In case convergence cannot be guaranteed (for all problems considered here it means that termination fails to hold), we rewrite s and t a limited number of times and then compare the results of these rewrite steps. More precisely, let $\mathcal{R}(u) = \{u_1, \ldots, u_n\}$ satisfying $u \equiv u_1 \Rightarrow \cdots \Rightarrow u_n$, $n \in \mathbb{N}$, and $\Rightarrow$ is the rewrite relation generated by regarding the axioms as rewrite rules as described earlier. (Note that $\mathcal{R}(u)$ is unique if a fixed reduction strategy is employed. We employed leftmost/outermost reduction as the most "natural" reduction strategy for these kinds of problems.) Then the fitness of Θ is given by

$$\Phi(\Theta) = \min(\{\Delta_\Theta(s', t') \mid s' \in \mathcal{R}(s), t' \in \mathcal{R}(t)\}).$$

Here, we have that Θ is a solution if $\Phi(\Theta) = 0$. That is, we naturally miss out on some solutions.

In both cases, the fitness measure is the smaller the fitter a combinator Θ is considered to be. If $\Phi(\Theta) = 0$ then Θ is a solution. As a refinement, we consider Θ_1 to be fitter than Θ_2 even though $\Phi(\Theta_1) = \Phi(\Theta_2)$ if $\mathcal{L}(\Theta_1) < \mathcal{L}(\Theta_2)$ in order to favor shorter combinators. (Note that the possible disadvantages of short combinators discussed earlier are compensated for by l_{min}.)

Naturally, when a combinator Θ has an almost perfect fitness, say $\Phi(\Theta) = 1$, we have absolutely no guarantee that Θ bears significant resemblance to a combinator that actually solves the problem. But this is a difficulty that GP can and has to cope with in general: A LISP program that "almost" solves a given task may be significantly different from a program that actually solves the task. But the powerfulness and limitations of GP cannot be addressed with theoretical considerations. We shall therefore now present our experimental results.

4 Experiments

We conducted our experimental studies in the light of 30 problems of CL taken from the TPTP problem library ([18]) version 1.2.1. The names of the problems (as used in the TPTP) are listed in the first column of table 1. Problems marked with an asterisk have a set of axioms that can immediately be used as convergent rewrite rules. Therefore they allow for computing the fitness measure based on comparing two normal forms as opposed to the more expensive (and less reliable) comparison of several intermediate rewrite steps. These 30 problems are all the problems in the TPTP that belong to one of two problem categories introduced in section 2. They also appear under various names in the literature (e.g., [19], [20], [12], [11]). Please consult the TPTP (sub-directory `COL`) for the specification of the problems.

Recall that GP starts with a population of combinators generated at random. A combinator is a term composed of the binary function symbol a and combinators (constants) from the current basis $\mathcal{B}$. For our experiments we adopted the following method for creating random combinators: When creating a term, we place either the symbol a at the root with probability p_a or a combinator $\Theta \in \mathcal{B}$ with probability p_Θ. Here, we set $p_a = 0.5$ and $p_\Theta = 0.5/|\mathcal{B}|$ for all $\Theta \in \mathcal{B}$. If symbol a is put at the root, we proceed recursively for both sub-terms. In case a given maximal depth $d_{max} \in \mathbb{N}$ is reached, where the depth $d(u)$ of a term u is defined by

$$d(u) = \begin{cases} 0, & u \in \mathcal{B} \vee u \in \mathcal{V} \\ 1 + \max(\{d(u_1), d(u_2)\}), & u \equiv a(u_1, u_2), \end{cases}$$

we forcefully stop the recursive generation of sub-terms (on the branches of the random term that account for reaching d_{max}) by setting $p_a = 0$ and $p_\Theta = 1/|\mathcal{B}|$ for all $\Theta \in \mathcal{B}$. (In [9] further, more elaborate methods for generating random terms are proposed. It remains to be investigated how those methods affect the outcome of our experiments.)

Table 1: Comparing GP, random search, and OTTER: Of the 30 problems, GP solves 27 problems, whereas 21 problems can be solved by random search, and 14 problems can be solved by OTTER in the autonomous mode (given the respective limitations of resources; see text for more information).

	RANDOM		GP			Proving with GP			OTTER
Name	p	n_{RND}	G^*	R^*	I^*	min	max	$\times$	auto mode
COL003-1	0.000004	1,151,291	19	10	200,000	11s	427s	3	—
COL004-1	0.000001	4,605,168	19	18	360,000	6s	121s	3	—
COL006-1	0	∞	—	—	—	—	—	10	—
COL029-1	0.12466	35	0	1	1,000	6.5s	8.9s	0	< 1s
COL030-1	0.003876	1186	0	1	1,000	6.1s	8.2s	0	< 1s
COL031-1	0.000694	6634	3	1	4,000	5.1s	9.1s	0	< 1s
COL032-1	0.000596	7725	3	2	8,000	2s	6.3s	0	< 1s
COL033-1	0.001248	3688	0	5	5,000	3.3s	8.9s	0	< 1s
COL034-1	0.000004	1,151,291	14	15	225,000	31s	464s	0	1.2s
COL035-1	0.000121	38,057	0	40	40,000	3.6s	107s	0	13s
COL036-1	0.000001	4,605,168	15	49	784,000	52s	489s	3	5s
COL037-1	0	∞	26	228	6,156,000	446s	558s	8	20s
COL038-1	0	∞	22	29	667,000	25s	561s	3	—
COL039-1	0.000446	10,324	4	2	10,000	3s	148s	0	< 1s
COL041-1	0.000017	270,891	17	4	72,000	44s	222s	0	< 1s
COL042-1	0	∞	15	90	1,440,000	35s	35s	9	—
COL043-1	0.000001	4,605,168	23	56	1,344,000	292s	581s	7	—
COL044-1	0	∞	24	3	75,000	35s	211s	0	14s
COL046-1	0	∞	19	11	220,000	68s	551s	3	—
COL049-1	0.000014	328,939	0	113	113,000	13s	400s	4	< 1s
COL057-1	0	∞	20	228	4,788,000	563s	563s	9	5.3s
COL060-1*	0.000082	56,159	9	1	10,000	< 1s	5.3s	0	—
COL061-1*	0.000116	39,698	7	1	8,000	< 1s	4.8s	0	—
COL062-1*	0.000008	575,644	15	4	64,000	7.4s	29s	0	—
COL063-1*	0.000011	418,650	14	2	30,000	2.6s	26s	0	—
COL064-1*	0.000002	2,302,583	15	20	320,000	20s	109s	0	—
COL065-1*	0.000001	4,605,168	7	90	720,000	13s	311s	0	—
COL066-1	0.000014	328,939	21	3	66,000	3.3s	38s	0	—
COL067-1	0	∞	—	—	—	—	—	10	—
COL072-1	0	∞	—	—	—	—	—	10	—

For our experiments we used $d_{max} = 10$. This is the depth limit for random combinators. For combinators produced via crossover, we restricted the depth to $D_{max} = 17$. Offspring exceeding D_{max} is discarded. (Note that these depth restrictions are only motivated by practical considerations. Large combinators are very likely to be more time consuming during fitness evaluations, but the gains of admitting very large combinators might be very marginal. So, depth restrictions must be viewed as an attempt to attain a reasonable cost-benefit ratio.) Furthermore, we set the "minimal structure requirement" $l_{min} = 10$, the "survival rate" $r = 30\%$, and the number of "intermediate rewrite steps" $n = 20$ which implies that $|\mathcal{R}(u)| \leq 20$ for all $u \in Term(\mathcal{F}, \mathcal{V})$.[2] The setting of these parameters were in parts inspired by [9] and by our own preliminary (non-conclusive) experiments. These parameters were so far not subjected to systematic studies and are hence in no sense optimal, but appear to be appropriate.

Besides these control parameters and fitness-related parameters, the success of GP largely depends on the size M of the population. Usually, the bigger M is, the better are the chances of success. However, in practice we have to find a compromise between a large population and acceptable computation time: Unlike in nature, on most computers the fitness of individuals is computed sequentially which accounts for the lion's share of computation effort. (All other computations involved in GP are practically negligible.) We therefore decided on a size $M = 1000$ that is pretty common for GP.

A run of GP—which consists of creating an initial random generation (population) commonly referred to as generation 0, and then successively producing successor generations 1, 2, ... by applying truncate selection and crossover (after assessing fitness)—is terminated as soon as a generation satisfies a given *success predicate*. Here, the success predicate is satisfied if a generation contains a combinator Θ that solves the problem at hand which is equivalent to Θ having fitness $\Phi(\Theta) = 0$. (In case we are interested in finding combinators that do not exceed a given maximal length L_{max} we can use the success predicate $\Phi(\Theta) = 0 \wedge \mathcal{L}(\Theta) \leq L_{max}$.) GP also terminates if generation G_{max} is reached.

When trying to solve a problem with GP, it has proven useful to make several shorter runs (with a relatively small G_{max}) instead of one long run (cp. [9]). In order to decide on a suitable value for G_{max} we employ a method introduced in [9]. The hub of this method is to perform numerous runs of GP for a given problem. In each run the ordinal of the generation that satisfied the success predicate is recorded (unless of course the run terminated on account of exceeding a given G_{max}). The data collected this way can be utilized to (empirically) determine probabilities $P(M, i)$ to succeed no later than by generation i when using a population of size M. (All other parameters derive from context.) This method is of course only a means for *a posteriori* analyses that is of no immediate use for a novel problem, but it is useful for detecting general tendencies and for conducting empirical studies.

In order to determine $P(M, i)$ here for a given problem, we ran GP 100 times. During each of these 100 runs, as soon as a generation satisfied the success predicate, the number of this generation was recorded. Otherwise GP stopped after reaching genera-

[2] We have $|\mathcal{R}(u)| < 20$ if u can be rewritten to a normal form in less than 20 rewrite steps (using leftmost/outermost reduction).

tion 50 (thus having processed 51 generations because of generation 0). With this data, $P(M,i)$ can be determined for the problem at hand. Since $P(M,i)$ is the probability to succeed by generation i,

$$z = 1 - (1 - P(M,i))^R$$

is the probability to succeed at least once by generation i in $R \in \mathbb{N}$ (independent) runs. That is, if we run GP R times with $G_{max} = i$, we shall have at least one generation satisfying the success predicate in one of these runs with probability z. In order to assess the difficulty of a problem for GP, we want to know how many runs $R = R(z,i)$ of GP with $G_{max} = i$ we have to execute so that a given success probability z is achieved. (All other parameters are kept fixed.) $R(z,i)$ is given by

$$R(z,i) = \begin{cases} \perp, & P(M,i) = 0 \\ 1, & P(M,i) = 1 \\ \left\lceil \frac{\log(1-z)}{\log(1-P(M,i))} \right\rceil, & \text{otherwise.} \end{cases}$$

(The symbol '$\perp$' represents "undefined".) The number of individuals that have to be processed during $R(z,i)$ runs with $G_{max} = i$ and a population of size M is

$$I(M,i,z) = R(z,i) \cdot M \cdot (i+1)$$

(Note that—for our parameter setting, in particular $r = 30\%$—300 of the 1000 individuals of a generation $i > 0$ are simply copied from generation $i-1$ and do not require their fitness to be (re-) computed.)

Following [9], we set $z = 0.99$. Then, for each problem, if there is a $0 \leq i \leq 50$ so that $P(M,i) > 0$ (i.e., GP could solve the problem at least once during the 100 runs), we can identify a $0 \leq G^* \leq 50$ and an associated $R^* = R(z, G^*)$ so that

$$I^* = I(M, G^*, z) = R^* \cdot M \cdot (G^* + 1)$$

is minimal.

For each of the 30 problems of CL the values for G^*, R^* and I^* are listed in table 1 in the correspondingly labeled columns. (An entry '—' indicates that $P(M,i) = 0$ for all $0 \leq i \leq 50$. That is, GP did not succeed in any of the 100 runs.) As an example, consider problem `COL041-1` (cf. table 1). There, we have $G^* = 17$ and $R^* = 4$ which signifies that—according to our empirical data—4 runs of GP, each run not going beyond generation 17, will produce at least one successful individual with a probability of 0.99 when using a population of size 1000. In doing so, $I^* = 4 \cdot (17+1) \cdot 1000 = 72,000$ individuals will be processed (at most).

This empirical study resulting in a measure I^* of the difficulty for GP to solve the respective problem can be utilized to counter a frequent criticism of GP, namely to be just a costly disguise for pure random search. Although Koza in [9] invalidates this criticism at least by presenting ample experimental evidence that, as a matter of fact, random search is no match for GP, we nevertheless conducted a number of experiments displayed by table 1 that allow for comparing random search with GP with respect to our application of GP.

We examined random search as follows: For each of the 30 problems we (empirically) determined the probability p to encounter a combinator Θ with $\Phi(\Theta) = 0$ by generating 10^6 random combinators and using the relative frequency of combinators Θ satisfying $\Phi(\Theta) = 0$ as p. These random combinators were produced using the same algorithm that was also employed to produce the random combinators of the initial population for GP.[3] Based on p, the number $n(z)$ of random combinators that have to be generated in order to find at least one combinator Θ satisfying $\Phi(\Theta) = 0$ with probability z is (analogous to $R(z,i)$) given by

$$n(z) = \begin{cases} \perp, & p = 0 \\ 1, & p = 1 \\ \left\lceil \frac{\log(1-z)}{\log(1-p)} \right\rceil, & \text{otherwise.} \end{cases}$$

For each problem, table 1 shows p and $n_{RND} = n(0.99)$. A comparison of n_{RND} and I^* in table 1 reveals that random search can keep up with GP only for the simplest problems (e.g., `COL029-1`,...,`COL033-1`) which GP also often solves best merely by generating a random generation 0 a few times. (Note that GP processes a number of individuals that is a multiple of $M = 1000$.) For all other problems, I^* is significantly smaller than n_{RND}. Also, GP can solve some problems with a rather small I^* (e.g., `COL044-1` or `COL046-1`) where random search did not find a single combinator solving the respective problem among 10^6 random combinators.[4] All in all, 27 of the 30 problems can be solved by GP, whereas only 21 problems (a subset of the 27 problems) can be solved by random search.

Thus, the above stated criticism is also not valid for this application of GP. As a matter of fact, the results obtained with GP encouraged us to make a rather daring experiment in that we compared the run-times of a GP-based prover (`GPP`) with a powerful "standard" theorem prover, namely OTTER. `GPP` simply repeats running GP until generation $G_{max} = 25$, and it stops this iteration as soon as the success predicate is satisfied. ($G_{max} = 25$ is inspired by the experimental results of table 1.) Besides such a successful halt, `GPP` can also time out. The time-out T was set to 600 seconds.

Hence, a run of `GPP` comprises several runs of GP (using $G_{max} = 25$), and it stops on account of a satisfied success predicate or if it exceeds T. Since `GPP` is subject to random effects, we did 10 runs with `GPP` in order to obtain more reliable data. In table 1, the columns labeled 'min' and 'max' list the minimal and maximal time (CPU time in seconds) required by `GPP` to succeed regarding the 10 runs. Column '$\times$' shows how many of the 10 runs were terminated by a time-out. Under the given conditions, `GPP` solves 27 problems in at least one of the 10 runs. (Note that one of the problems that cannot be solved, namely `COL067-1`, is labeled as an "open problem" in the TPTP.)

[3]Note that we limit the depth of combinators to $d_{max} = 10$ for random search as well as in the initial population of GP, whereas combinators of depth $D_{max} = 17$ may be produced via crossover. This does not put random search at a disadvantage: All the combinators found by GP well fit within the d_{max} restriction. GP can make use of the bigger D_{max} to create "intermediate" larger combinators that play a role in the evolutionary process (cp. [9]).

[4]We picked the number 10^6 somewhat arbitrarily, but we think it is adequate for our purposes here. Generating more random combinators might help to find a random combinator where none could be found before, but this will not significantly distort the observed trend.

We employed OTTER in its "autonomous mode" ([13]) where it uses built-in strategies for recognizing problem domains and choosing appropriate parameter settings. Both OTTER and GPP are implemented in C and were run on a SPARCstation 10. The time-out $T = 600$ seconds was also imposed on OTTER resulting in an entry '—' in table 1 (last column).

The results of OTTER, which—in the autonomous mode—solves merely 14 problems that GPP can also cope with, show that these 30 problems of CL comprise several hard problems that cannot be handled reasonably by a theorem prover with standard or automated parameter settings. (This is in spite of the fact that the designers of OTTER have extensive experience with problems of CL.) Of course OTTER can prove some more of these 30 problems if the parameters are set appropriately for each problem by a user *who is experienced in working with* OTTER (cp. [11]). GPP, however, tackled all problems with the same parameter setting. We believe that an automated theorem prover (such as OTTER) is hardly capable of achieving a success rate comparable to that of GPP, and if it is, then it is not without significant (and mostly laborious and time-consuming) intervention on the parts of a user. Furthermore, even if a theorem prover is faster than GPP (at least for certain problems, as it is the case for OTTER in particular in connection with problems COL036-1, COL037-1, COL044-1, and COL057-1), we must not forget the enormous potential for massive parallelization inherent to GP. Instead of computing the fitness of individuals sequentially (as it is done in our implementation), it is literally natural to perform these computations in parallel so that the maximum of these computation times rather than their sum adds to the overall computation time. Then, overall computation time becomes less sensitive to an increasing population size (assuming a suitable hardware architecture) which entails salient speed-ups (e.g., [6]). Also note that GP requires much less memory. Memory requirement naturally depends on the population size M. With $M = 1000$, GP needs less than 1 MB, which is a value no theorem prover can work with reasonably.

We do not want to deny that GPP also has unpleasant properties. Since random effects play a (vital) role in GP, sometimes GPP succeeds very fast, sometimes it takes longer, and sometimes it even times out. But this indeterminism may also be regarded as a benefit: While a user of a standard theorem prover has to sit down and think of alternative parameter settings if his system failed when using a particular setting, for GPP it makes sense to simply "try again"—one might get lucky next time.

And just to add another argument to the role of luck in answering open questions, we indeed can now answer a question posed in [19] regarding problem COL004-1 (named '*Problem 4*' in [19]). Problem COL004-1 is concerned with the question whether there is a combinator composed of combinators from $\mathcal{B} = \{S, K\}$ that satisfies the equation for U given by $Uxy = y(xxy)$. In [19], the shortest known combinator has length 13, and the question was whether there is a shorter combinator. During our experiments we encountered combinators of length 10, 11, and 12:

$$S(S(K(S(SKS))))(SKS) \qquad S(S(K(S(SKS))))(SK(KK))$$

$$S(KS)S(S(KS)(SK))(SKS)$$

5 Discussion

The idea of applying processes governed by random influences to a theoretically well-founded field such as automated reasoning appears strange, even absurd. Yet, making an automated reasoning system actually work heavily depends on procedures that elude a rigorous theoretical study. The (practical) value of procedures such as the eminently important and omnipresent search-guiding heuristics can only be assessed via empirical studies. Dealing efficiently with the general undecidability of problems related to automated reasoning also led researchers to test approaches that seemed ludicrous, but turned out practical (e.g., [7]). Such approaches should not be precluded without a closer investigation just because they are—at the moment—not as amenable to a thorough theoretical examination as other (possibly less beneficial) approaches are.

The application of genetic programming to solving problems of combinatory logic is an attempt to model the ability of mathematicians to suggest—based on intuition and experience—appropriate expressions to be substituted for existentially quantified variables so as to significantly simplify the problem at hand. The fact that genetic programming basically realizes intuition and experience with randomness and evolution seems odd, but it suits the computer, and our experimental results clearly speak in favor of this approach. Although we only implemented a very basic version of genetic programming (for extensions and refinements see, e.g., [10], [15]), our results are very encouraging. The success rate of our approach is remarkable even though we employed one fixed parameter setting. Further experiments have to show whether modifications of these settings can bring improvements in particular with respect to the problems not handled to our complete satisfaction.

Furthermore, it might be worthwhile investigating the applicability of this approach to inductive theorem proving, where existential quantifications pose even more serious problems (e.g., [2]).

Finally, we applied genetic programming to a very limited domain of automated reasoning in order to show that genetic programming can simulate to some extent a likewise limited but nonetheless important capability of mathematicians. After replacing existentially quantified variables, the problems became "almost decidable" which is here crucial for an efficient and sensible fitness evaluation required by genetic programming. It remains to be investigated how genetic programming can be applied more universally to a broader range of problems from automated reasoning. That is, the question is whether genetic programming or the general genetic algorithm can also in the context of automated reasoning be fully used as what it actually is, namely an alternative and powerful search method.

References

[1] **Barendregt, H.P.**: *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, Amsterdam, 1981.

[2] **Biundo, S.**: *Automated Synthesis of Recursive Algorithms as a Theorem Proving Tool*, Proc. 8^{th} European Conference on AI (ECAI-88), Munich, GER, 1988, pp. 553–558.

[3] **Curry, H.B.; Feys, R.**: *Combinatory Logic*, North-Holland, Amsterdam, 1958.

[4] **Denzinger, J.; Fuchs, M.**: *Goal oriented equational theorem proving using teamwork*, Proc. 18^{th} German Conference on AI (KI-94), Saarbrücken, GER, 1994, LNAI 861, pp. 343–354.

[5] **De Jong, K.**: *Learning with Genetic Algorithms: An Overview*, Machine Learning **3**:121–138, 1988.

[6] **Dracopoulos, D.C.; Kent, S.**: *Speeding up Genetic Programming: A Parallel BSP Implementation*, Proc. 1^{st} International Conference on Genetic Programming (GP-96), Stanford University, CA, USA, 1996, p. 421.

[7] **Ertel, W.**: *Random Competition: A Simple, but Efficient Method for Parallelizing Inference Systems*, Techn. Report TUM-19050, Technical University of Munich, 1990.

[8] **Holland, J.H.**: *Adaptation in natural and artificial systems*, Ann Arbor: Univ. of Michigan Press, 2^{nd} edition, 1992.

[9] **Koza, J.R.**: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, MA, 1992.

[10] **Koza, J.R.**: *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, Cambridge, MA, 1994.

[11] **Lusk, E.; McCune, W.**: *Uniform Strategies: The CADE-11 Theorem Proving Contest*, Journal of Automated Reasoning **11**, 1993.

[12] **Lusk, E.; Wos, L.**: *Benchmark Problems in Which Equality Plays the Major Role*, CADE-11, Saratoga Springs, NY, USA, 1992, LNAI 607, pp. 781–785.

[13] **McCune, W.**: *OTTER 3.0 Reference Manual and Guide*, Techn. Report ANL-94/6, Argonne Natl. Laboratory, 1994.

[14] **McCune, W.; Wos, L.**: *A Case Study in Automated Theorem Proving: Finding Sages in Combinatory Logic*, Journal of Automated Reasoning **3**, 1987, pp. 91–107.

[15] **Niwa, T.; Iba, H.**: *Distributed Genetic Programming: Empirical Study and Analysis*, Proc. 1^{st} International Conference on Genetic Programming (GP-96), Stanford University, CA, USA, 1996, pp. 339–344.

[16] **O'Donnell, M.J.:** *Computing in Systems Described by Equations*, LNCS 58, Springer-Verlag, 1977.

[17] **Smullyan, R.:** *To Mock a Mockingbird*, A. Knopf, New York, 1985.

[18] **Sutcliffe, G.; Suttner, C.; Yemenis, T.:** *The TPTP Problem Library*, Proc. CADE-12, Nancy, FRA, 1994, LNAI 814, pp. 252–266.

[19] **Wos, L.; McCune, W.:** *Challenge Problems Focusing on Equality and Combinatory Logic: Evaluating Automated Theorem-Proving Programs*, CADE-9, Argonne, IL, USA, 1988, LNCS 310, pp. 714–729.

[20] **Wos, L.; Winker, S.; McCune, W.; Overbeek, R., Lusk, E.; Stevens, R.:** *Automated Reasoning Contributes to Mathematics and Logic*, CADE-10, Kaiserslautern, GER, 1990, LNAI 449, pp. 485–499.