

Generating Lambda Term Individuals in Typed Genetic Programming Using Forgetful A*

Tomáš Křen^{*†}

tomkren@gmail.com

^{*}Charles University

Faculty of Mathematics and Physics

Malostranské náměstí 25, Prague

Czech Republic

Roman Neruda[†]

roman@cs.cas.cz

[†]Institute of Computer Science

Academy of Sciences of the Czech Republic

Pod Vodárenskou věží 2, Prague

Czech Republic

Abstract—Tree based genetic programming (GP) traditionally uses simple S-expressions to represent programs, however more expressive representations, such as lambda calculus, can exhibit better results while being better suited for typed GP. In this paper we present population initialization methods within a framework of GP over simply typed lambda calculus that can be also used in the standard GP approach. Initializations can be parameterized by different search strategies, leading to wide spectrum of methods corresponding to standard ramped half-and-half initialization on one hand, or exhaustive systematic search on the other. A novel geometric strategy is proposed that balances those two approaches. Experiments on well known benchmark problems show that the geometric strategy outperforms the standard generating method in success rate, best fitness value, time consumption and average individual size.

I. INTRODUCTION

Genetic programming (GP) represents an efficient method for automatic generating of programs by means of evolutionary techniques [1], [2]. Early attempts to enhance the GP approach with the concept of types include the seminal work [3] where the ideas from Ada programming language were used to define a so-called strongly typed GP. Use of types naturally opens door to enriching S-expressions, the traditional GP representation of individuals, with concepts from lambda calculus, which is simple yet powerful functional mathematical and programming language extensively used in type theory. Such attempts has shown to be successful [4].

In this paper we present a tree generating method suitable both for the standard GP[1] and typed GP over lambda calculus. We chose to describe the algorithm in the means of the typed lambda calculus since the standard S-expression representation can be understood as a special case of this representation. However, for evaluation of our approach we focus more on problems from standard GP. We chose this evaluation approach because the result of the GP system depends profoundly on the crossover operation, which remains an open problem for the typed GP over lambda calculus.

During the expansion phase of the term generating procedure the space of unfinished terms can be browsed with respect to various search strategies. Our approach to this problem aims to utilize the full arsenal given by the simply typed lambda calculus. Thus, the natural idea is to employ an exhaustive systematic search. On the other hand, if we were to mimic the standard GP approach, a quite arbitrary yet common

and successful ramped half-and-half generating heuristic [5] should probably be used. These two search methods in fact represent boundaries between which we will try to position our parameterized solution that allows us to take advantage of both strategies. This design goal also differentiate our approach from the three state of the art proposals for typed GP known to us that are discussed in the following section. Our proposed *geometric search strategy*¹ described in this paper is such a successful hybrid mixture of random and systematic exhaustive search. Experiments show that it is also very efficient dealing with one of the traditional GP scarecrows - the bloat problem.

The rest of the paper is organized as follows: The next section briefly discusses related work in the field of typed GP, while section III introduces necessary notions. Main original results about search strategies in individual generating are described in section IV. Section V presents results of our method on three well-known tasks, and the paper is concluded by section VI.

II. RELATED WORK

Yu presents a GP system utilizing polymorphic higher-order functions² and lambda abstractions [4]. Important point of interest in this work is use of `foldr`³ function as a tool for *implicit recursion*, i.e. recursion without explicit recursive calls. The terminal set for constructing lambda abstraction subtrees is limited to use only constants and variables of that particular lambda abstraction, i.e., outer variables are not allowed to be used as terminals in this work. This is significant difference from our approach since we permit all well-typed normalized λ -terms. From this difference also comes different crossover operation. We focus more on term generating process; their term generation is performed in a similar way as the standard one, whereas our term generation also tries to utilize techniques of systematic enumeration.

Briggs and O'Neill present technique utilizing typed GP with combinators [7]. The difference between approach presented in this work and our approach is that in this work terms are generated straight from *library* of combinators and no lambda abstractions are used. They are using more general

¹Not to be confused with geometric crossover introduced by [6].

²Higher-order function takes another function as an input parameter.

³ $\text{foldr } f \ z \ [] = z$
 $\text{foldr } f \ z \ (x:xs) = f \ x \ (\text{foldr } f \ z \ xs)$

polymorphic type system than us – the Hindley–Milner type system. They also discuss the properties of exhaustive enumeration of terms and compare it with GP search. They also present interesting concept of *Generalized genetic operator* based on term generation.

Binard and Felty use even stronger type system (*System F*) [8]. But with increasing power of the type system comes increasing difficulty of term generation. For this reason evolution in this work takes interesting and nonstandard shape (fitness is associated with *genes* which are evolved together with *species* which together participate in creation of individuals). This differs from our approach, which tries to be generalization of the standard GP[1].

In contrast with above mentioned works our approach uses very simple type system (simply typed lambda calculus) and concentrates on process of generation able to generate all possible well-typed normalized lambda terms. In order to do so we use technique based on *inhabitation machines* described by Barendregt [9].

III. PRELIMINARIES

In this section, several notions necessary to build a typed GP based on lambda calculus are introduced. First, let us describe a programming language, in which the GP algorithm generates individual programs — the so called λ -terms.

Definition 1: Let V be infinite countable set of *variable names*. Let C be set of *constant names*, $V \cap C = \emptyset$. Then Λ is set of λ -terms defined inductively as follows.

$$\begin{aligned} x \in V \cup C &\Rightarrow x \in \Lambda && \text{(Variable or constant)} \\ M, N \in \Lambda &\Rightarrow (M \ N) \in \Lambda && \text{(Function application)} \\ x \in V, M \in \Lambda &\Rightarrow (\lambda x. M) \in \Lambda && (\lambda\text{-abstraction}) \end{aligned}$$

Function application and *λ -abstraction* are concepts well known from common programming languages.

For example in JavaScript term $(M \ N)$ translates to expression $M(N)$ and $(\lambda x. M)$ translates to expression $\text{function}(x)\{\text{return } M;\}$. In other words, the function application corresponds to the act of supplying a function with an argument, and the λ -abstraction is equivalent to *anonymous function*⁴. $M_1 \ M_2 \ M_3 \ \dots \ M_n$ is an abbreviation for $(\dots((M_1 \ M_2) \ M_3) \ \dots \ M_n)$ and $\lambda x_1 x_2 \dots x_n. M$ for $(\lambda x_1. (\lambda x_2. \dots (\lambda x_n. M) \dots))$.

A λ -term as described above corresponds to a program expression with no type information included. Now we will describe *types* (or *type terms*).

Definition 2: Let A be set of *atomic type names*. Then \mathbb{T} is set of *types* inductively defined as follows.

$$\begin{aligned} \alpha \in A &\Rightarrow \alpha \in \mathbb{T} \\ \sigma, \tau \in \mathbb{T} &\Rightarrow (\sigma \rightarrow \tau) \in \mathbb{T} \end{aligned}$$

Type $\sigma \rightarrow \tau$ is type for functions taking as input something of a type σ and returning as output something of a type τ .

$\tau_1 \rightarrow \tau_2 \rightarrow \dots \rightarrow \tau_n$ is an abbreviation for $\tau_1 \rightarrow (\tau_2 \rightarrow (\dots \rightarrow (\tau_{n-1} \rightarrow \tau_n) \dots))$. Such "chain of arrows" types simulate types of functions with multiple (here $n - 1$) inputs. This technique is called *currying*. One can grasp the trick by considering $(f \ M_1 \ \dots \ M_n)$ as shorthand for $f(M_1, \dots, M_n)$ and by observing that functions with such types have terms of the form $\lambda x_1 x_2 \dots x_n. M$.

The system called *simply typed λ -calculus* is now easily obtained by combining the previously defined λ -terms and types together.

Definition 3:

- 1) Let Λ be set of λ -terms. Let \mathbb{T} be set of *types*. A *statement* $M : \sigma$ is a pair $(M, \sigma) \in \Lambda \times \mathbb{T}$. Statement $M : \sigma$ is vocalized as "*M has type σ* ". The term M is called the *subject* of the statement $M : \sigma$.
- 2) A *declaration* is a statement $x : \sigma$ where $x \in V \cup C$.
- 3) A *context* is set of declarations with distinct variables as subjects.

Context is a basic type theoretic concept suitable as a typed alternative for terminal and function set in standard GP. Notation $\Gamma, x : \sigma$ denotes $\Gamma \cup \{(x : \sigma)\}$ such that Γ does not contain any declaration with x as subject. We also write $x : \sigma \in \Gamma$ instead of $(x, \sigma) \in \Gamma$.

Definition 4: A statement $M : \sigma$ is *derivable from* a context Γ (notation $\Gamma \vdash M : \sigma$) if it can be produced by the following rules.

$$\begin{aligned} x : \sigma \in \Gamma &\Rightarrow \Gamma \vdash x : \sigma \\ \Gamma \vdash M : \sigma \rightarrow \tau, \Gamma \vdash N : \sigma &\Rightarrow \Gamma \vdash (M \ N) : \tau \\ \Gamma, x : \sigma \vdash M : \tau &\Rightarrow \Gamma \vdash (\lambda x. M) : \sigma \rightarrow \tau \end{aligned}$$

Our goal in term generation is to produce terms M for a given pair $\langle \tau; \Gamma \rangle$ such that for each M is $\Gamma \vdash M : \tau$.

Definition 5: Let V be infinite countable set of *variable names*. Let C be set of *constant names*, $V \cap C = \emptyset$. Let \mathbb{T} be set of types. Let \mathbb{C} be set of all contexts on $(V \cup C, \mathbb{T})$. Then Λ' is set of *unfinished λ -terms* defined inductively as follows.

$$\begin{aligned} \tau \in \mathbb{T}, \Gamma \in \mathbb{C} &\Rightarrow \langle \tau; \Gamma \rangle \in \Lambda' && \text{(Unfinished leaf)} \\ x \in V \cup C &\Rightarrow x \in \Lambda' && \text{(Variable or constant)} \\ M, N \in \Lambda' &\Rightarrow (M \ N) \in \Lambda' && \text{(Function application)} \\ x \in V, M \in \Lambda' &\Rightarrow (\lambda x. M) \in \Lambda' && (\lambda\text{-abstraction}) \end{aligned}$$

Unfinished leaf $\langle \tau; \Gamma \rangle$ stands for yet not specified λ -term of the type τ build from symbols of Γ .

IV. OUR APPROACH

A. Introduction

Our approach to λ -term generating is based on technique briefly described in [9], which generates well-typed λ -terms in their long normal form. We use this technique to perform a systematic exhaustive enumeration of λ -terms in their long normal form in order from the smallest to the largest. We use well known *A* algorithm* [10] for this task. *A** is used to search in a given state space for a goal state. It finds the optimal solution (in our case the smallest term) and uses "advising"

⁴Apart from JavaScript, anonymous functions are common e.g. in Python and Ruby, they were recently introduced to C++, and they are expected to be supported in Java 8.

heuristic function. It maintains a priority queue to organize states yet to be explored. Initially this queue contains only the initial state.

Our state space to search in is the space of unfinished λ -terms. The initial state is the unfinished term $\langle \tau; \Gamma \rangle$, where τ is the desired type of terms to be generated and Γ is the context representing the set of building symbols to be used in construction of terms (it corresponds to the set $T \cup F$ in standard GP enriched with types). The process of determining successors of a state described below is designed so it constructs well-typed λ -terms and omits no λ -term in its long normal form. A state is considered a goal state if it contains no unfinished leaf, i.e., it is a finished λ -term.

Our generating method is based on simple modification of the standard A*, which we call *forgetful A**. This modification consist in additional parameter for the A* algorithm – the *search strategy*. It is a simple filtration function (along with initialization procedure) that is given the set of all successors of the state that is being examined and returns a subset of this input. This subset is added to the priority queue to be further explored. In this way the search space may be reduced as the filtration function may *forget* some successors. If the queue becomes empty before the desired number of λ -terms is generated, then the initial state is inserted to the queue and the process continues. For the standard A* this would be meaningless, but since our A* is forgetful this kind of restart makes sense.

A* keeps a priority queue of states during the generation process, on the other hand the *ramped half-and-half method*, the standard GP algorithm for generating individuals, keeps only one individual which is gradually constructed. This behavior is easily achieved by use of suitable search strategy that returns subset consisting of only one successor. The systematic search is obtained by search strategy that returns whole input set. Our novel *geometric strategy* can be understood as point somewhere between those two extremes.

B. Algorithm

Let us look on the proposed generating procedure in a grater detail sufficient as guide for implementing it. The inputs for the term generating algorithm are following:

- 1) Desired type τ of generated terms.
- 2) Context Γ representing the set of building symbols.
- 3) Number n of terms to be generated.
- 4) Search strategy S .

Essential data structure of our algorithm is priority queue of unfinished terms. Priority of an unfinished term is given by its size⁵. At the beginning, the queue contains only one unfinished term; $\langle \tau; \Gamma \rangle$. The search strategy S also initializes its internal state (if it has one).

⁵A* heuristic function is hidden in method of computing size of unfinished leafs $\langle \tau; \Gamma \rangle$. Our algorithm uses trivial estimate $|\langle \tau; \Gamma \rangle| = 1$ which is trivially admissible. This heuristic is not as naive as it might seem since it is quite usual to have x such that $x : \tau \in \Gamma$. Since the true value of $|\langle \tau; \Gamma \rangle|$ depends only on τ and types in Γ (the *signature*), no matter how many variables/constants of each type there are, it should by pretty effective to compute this value precisely and store them for later.

At each step, the term M with the smallest size is pulled from the queue. According to the number of unfinished leafs in M one of the following actions is performed:

- 1) If the term M has no unfinished leaf (i.e., it is a finished term satisfying $\Gamma \vdash M : \tau$), then it is added to the result set of generated terms.
- 2) Otherwise, *successors* of the unfinished term M are filtered out by *search strategy* S and those successors that outlast the filtration are inserted into the queue.

Successors of an unfinished term M are obtained by *expansion* of the *DFS-first* unfinished leaf L (i.e., the leftmost unfinished leaf of M).

Expansion of the selected unfinished leaf L leads to creation of one or many (possibly zero) successors. In this process, L is replaced by a new subterm defined by the following rules⁶:

- 1) *Function rule*:
If $L = \langle \rho_1 \rightarrow \dots \rightarrow \rho_n \rightarrow \alpha; \Gamma \rangle$, where α is atomic type and $n \geq 1$, then L is replaced by $(\lambda x_1 \dots x_n. \langle \alpha; \Gamma, x_1 : \rho_1, \dots, x_n : \rho_n \rangle)$. Thus this expansion results in exactly one successor.
In other words, a *function* type is replaced by a new anonymous function, and for each of its inputs is added a fresh variable into the context of its body – which results in possibility of using those new variables inside the body of the anonymous function. Notice that this rule is never used in problems satisfying the closure requirement.
- 2) *Atomic rule*:
If $L = \langle \alpha; \Gamma \rangle$ where α is *atomic* type, then for each $f : (\tau_1 \rightarrow \dots \rightarrow \tau_m \rightarrow \alpha) \in \Gamma$, $m \geq 0$ the unfinished leaf L is replaced by $(f \langle \tau_1; \Gamma \rangle \dots \langle \tau_m; \Gamma \rangle)$. Thus this expansion results in many (possibly zero or one) successors.
In other words, an *atomic* type is replaced by a constant symbol (for $m = 0$) or function symbol (for $m > 0$) from the local context. Unfinished leafs $\langle \tau_i; \Gamma \rangle$ correspond to the argument subtrees yet to be generated – precisely as in S-expression generation. Notice that for problems satisfying closure requirement only this rule is applicable; therefore standard S-expressions are generated for such a problem.

Algorithm 2 summarizes this key phase of generating procedure by pseudocode.

Now that we have all possible successors of M , we are about to apply the *search strategy* S . A search strategy is a procedure which takes as input a set of unfinished terms and returns a subset of the input set. Therefore, search strategy acts as a filter reducing the search space.

If the queue becomes empty before the desired number n of terms is generated, then the initial unfinished term $\langle \tau; \Gamma \rangle$

⁶For the sake of simplicity it is presented as two separate rules. Since the first rule results in exactly one successor it is smarter to combine those two rules into one resulting in that unfinished leafs have only atomic types. At the beginning we transform the initial type into atomic by first rule (if necessary). After that only second rule is applied, but if it results in creation of some unatomic $\langle \tau_i; \Gamma \rangle$, then first rules are applied, but during the same successor creation This step eliminates all unatomic unfinished leafs.

Algorithm 1: Individual generating procedure.

```
function generate(Type  $\tau$ , Context  $\Gamma$ , Int  $n$ , Strategy  $S$ )
  results  $\leftarrow$  empty set
  while |results| <  $n$  do
     $S.init()$ 
    open  $\leftarrow$  empty priority queue
    start  $\leftarrow \langle \tau; \Gamma \rangle$ 
    start.G  $\leftarrow 0$ 
    start.F  $\leftarrow 1$ 
    open.insert(start)
    while  $\neg$  open.isEmpty() do
      term  $\leftarrow$  open.popTermWithLowestF()
      if numUnfinLeafs(term) = 0 then
        results.insert(term)
        if |results| =  $n$  then
          return results
      else
        nexts  $\leftarrow$  expand(term)
        depth  $\leftarrow$  term.depthOfExpandedLeaf()
        nexts'  $\leftarrow S.filter(nexts, depth)$ 
        for next  $\in$  nexts' do
          heur  $\leftarrow$  numUnfinLeafs(next)
          next.G  $\leftarrow$  term.G + 1
          next.F  $\leftarrow$  next.G + heur
          open.insert(next)
    return results
```

Algorithm 2: Expansion (successor) procedure.

```
function expand(UnfinishedTerm term)
  nexts  $\leftarrow$  empty set
   $\langle \sigma; \Gamma \rangle \leftarrow$  DFS-first unfinished leaf of term
  switch  $\sigma$  do
    case  $\rho_1 \rightarrow \dots \rightarrow \rho_n \rightarrow \alpha : n \geq 1, \alpha$  atomic
       $t \leftarrow (\lambda x_1 \dots x_n. \langle \alpha; \Gamma, x_1 : \rho_1 \dots x_n : \rho_n \rangle)$ 
      next  $\leftarrow$  replace  $\langle \sigma; \Gamma \rangle$  in term with  $t$ 
      nexts.insert(next)
    case  $\alpha : \alpha$  atomic
      for  $f : \tau_1 \rightarrow \dots \rightarrow \tau_m \rightarrow \alpha \in \Gamma, m \geq 0$  do
         $t \leftarrow (f \langle \tau_1; \Gamma \rangle \dots \langle \tau_m; \Gamma \rangle)$ 
        next  $\leftarrow$  replace  $\langle \sigma; \Gamma \rangle$  in term with  $t$ 
        nexts.insert(next)
  return nexts
```

Algorithm 3: Geometric strategy.

```
function Geometric.filter(UnfinTerms nexts, Int depth)
  nexts'  $\leftarrow$  empty set
  for next  $\in$  nexts do
    with probability  $q^{depth}$  do nexts'.insert(next)
  return nexts'
```

is inserted to the queue, search strategy S again initializes its internal state and the process continues. The whole generating procedure is summarized by *Algorithm 1*.

Let us now discuss three such search strategies.

Systematic strategy: If we use trivial strategy that returns all the inputs, then the algorithm systematically generates first n smallest lambda terms in their *long normal form*.

Ramped half-and-half strategy: This is generalization of the standard ramped half-and-half method described in [1]. If applied to context satisfying closure requirement (all constants/variables are of the same type, all functions are operations on this type), it will behave in the same way as the standard method. The internal state of this strategy consists of two variables. It is the only one strategy described here that uses an internal state.

- 1) *isFull* - A boolean value, determining whether *full* or *grow* method will be performed.
- 2) d - An integer value from $\{2, \dots, D_{init}\}$, where D_{init} is predefined maximal depth (e.g. 6).

This strategy returns precisely one randomly (uniformly) selected element from the *selection subset* of input set (or zero elements if the input set is empty). The *selection subset* to select from is determined by *depth*, d and *isFull*. The *depth* parameter is the depth (in the term tree) of the unfinished leaf that was expanded. Those elements of input set whose newly added subtree contains one or more unfinished leafs are regarded as *non-terminals*, whereas those whose newly added subtree contains no unfinished leaf are regarded as *terminals*. If *depth* = 0, then the subset to select from is set of all *non-terminals* of the input set. If *depth* = d , then the subset to select from is set of all *terminals* of the input set. In other cases of *depth* it depends on value of *isFull*. If *isFull* = *true*, then the subset to select from is set of all *non-terminals* of the input set. If *isFull* = *false*, then the subset to select from is the whole input set.

Geometric strategy: We can see those two previous strategies as two extremes on the spectrum of possible strategies. *Systematic strategy* filters no successor state thus performing exhaustive search resulting in discovery of n smallest terms in one run. On the other hand, *ramped half-and-half strategy* filters all but one successor states resulting in degradation of the priority queue into "fancy variable". *Geometric strategy* is simple yet fairly effective term generating strategy somewhere in the middle of this spectrum. It is parameterized by parameter $q \in (0, 1)$, its default well-performing value is $q = 0.75$. For each element of the input set it is probabilistically decided whether it will be returned or omitted. A probability p of returning is same for all elements, but depends on the *depth*, which is defined in the same way as in previous strategy. It is computed as follows.

$$p = q^{depth}$$

This formula is motivated by idea that it is important to explore all possible root symbols, but as the *depth* increases it becomes less "dangerous" to omit an exploration branch. We can see this by considering that this strategy results in somehow forgetful A* search. With each omission we make the search space smaller. But with increasing depth these omissions have smaller impact on the search space, i.e., they cut out lesser portion of the search space. Another slightly esoteric argument supporting this formula is that "root parts"

of a program usually stand for more crucial parts with radical impact on global behavior of a program, whereas "leaf parts" of a program usually stand for less important local parts (e.g. constants). This strategy also plays nicely with the idea that "too big trees should be killed". *Algorithm 3* summarizes the filtration procedure performed by the geometric strategy.

Furthermore, our system utilizes generalization of the standard tree-swapping crossover operator. Since it is beyond the scope of this paper we mention it only briefly. Two main concerns with swapping typed subtrees are types and free variables. Well-typed offspring is obtained by swapping only subtrees of the same type. Only subtrees with corresponding counterpart in the second parent are randomly chosen from. More interesting problem lies in free variables, which may cause trouble if swapped somewhere where it is suddenly not bounded. In order to circumvent this difficulty we utilize technique called *abstraction elimination*[11] that transforms an arbitrary λ -term into λ -term that contains no lambda abstractions and no bound variables. After the initial population is generated, it is transformed by abstraction elimination. Another possible transformation taking place after initialization is η -normalization shortening rather long normal form into shorter $\beta\eta$ -normal form. Another performance enhancing transformation is option of using "applicative" tree representation (coming directly from inductive definition of λ -terms) instead of more traditional S-expression representation. Favorable properties of applicative tree representation are also reported in [12].

A remarkable benefit of parameterizing the generating method by a search strategy taking the form of a simple filtration function is that such a function can be expressed by functional language in a very economical way. Take for instance a search strategy that behaves same as the *geometric* except that when it comes to situation where all the elements are filtered out it acts as the *ramped half-and-half*. Such *strategy composition* is easily describable as a higher order function. Thus comes to mind the possibility of using our system to come up with new search strategies on its own.

V. EXPERIMENTS

We made three experiments comparing performance of standard *ramped half-and-half strategy* with our *geometric strategy* (with default parameter $q = 0.75$).

Despite the fact that the geometric strategy was designed for typed GP over lambda trees it works well for standard GP problems satisfying the closure requirement. Furthermore, for such problems our approach generates standard S-expressions only. Another reason for including standard GP benchmarks in our experiments is the fact that the results are seriously affected by a crossover operator. Design of a good crossover for typed GP remains an open problem. Our crossover is quite complicated and beyond the scope of this paper, but for problems satisfying the closure requirement it behaves exactly the same as the standard GP crossover.

Three well known benchmark problems (Simple symbolic regression, Artificial ant and Even parity problem) were chosen for our experiments. First two satisfy the closure requirement, whereas the third one does not.

TABLE I. STANDARD GP PARAMETERS.

| | |
|-----------------------------|-------------------------------------|
| Population size | 500 |
| Number of generations | 51 |
| Probability of crossover | 90% |
| Probability of reproduction | 10% |
| Probability of mutation | 0% |
| Max depth (initial) | 6 |
| Max depth (crossover) | 17 |
| Representation | S-expressions |
| Crossover | Standard tree swapping (90-10) |
| Selection | Fitness-proportionate |
| Generating method | Ramped half-and-half / geometric s. |

In order to ensure that the experiments are replicable we chose to use the standard GP system described in [1], with all the parameters set to their default values (which are summarized in table I), only with the following exceptions:

- 1) In the second and third experiment the preservation of the best individual into the next generation is performed.
- 2) In the third experiment we use our crossover operation briefly mentioned above together with use of "applicative" tree representation.

For problems satisfying closure requirement our generating method generates S-expressions. Therefore it can be directly plugged into the standard GP system. Each experiment consisted of 50 independent runs of GP algorithm. Each run had maximally 51 generations and population size was 500 individuals. In those experiments we analyze the fitness of the best individual, the average size of an individual, the ability of the system to produce a correct solution and computational cost estimates throughout generations. For the last two metrics we use the popular measurement methods within GP field — the *performance curves* described in [1] — $P(M, i)$ (cumulative probability of success) and $I(M, i, z)$ (the total number of individuals that must be processed to yield a correct solution with probability $z = 99\%$). Difference of those two generating methods is statistically analyzed by the Welch t-test by comparing the fitness values of the best individuals of the run. Graphs showing mean values of the fitness of the best individual and the average size of an individual throughout generations contain error bars representing the standard error of the mean (SEM).

A. Simple Symbolic Regression

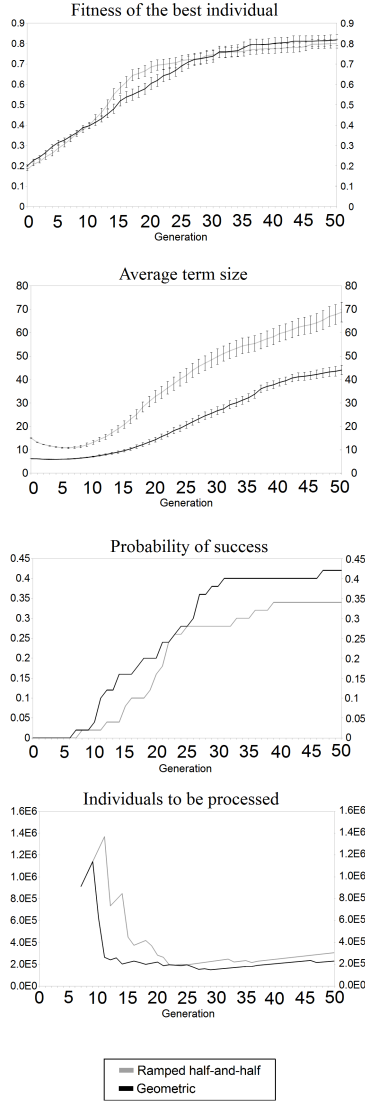
Simple Symbolic Regression is a problem described in [1]. Objective of this problem is to find a function $f(x)$ that fits a sample of twenty given points. The target function is function $f_t(x) = x^4 + x^3 + x^2 + x$. Desired type of generated programs τ and building blocks context Γ are following.

$$\begin{aligned} \tau &= \mathbb{R} \rightarrow \mathbb{R} \\ \Gamma &= \{ (+) : \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R}, (-) : \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R}, \\ &\quad (*) : \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R}, rdiv : \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R}, \\ &\quad sin : \mathbb{R} \rightarrow \mathbb{R}, cos : \mathbb{R} \rightarrow \mathbb{R}, \\ &\quad exp : \mathbb{R} \rightarrow \mathbb{R}, rlog : \mathbb{R} \rightarrow \mathbb{R} \} \end{aligned}$$

where

$$rdiv(p, q) = \begin{cases} 1 & \text{if } q = 0 \\ p/q & \text{otherwise} \end{cases}$$

Fig. 1. Graphs for Simple symbolic regression.



$$rlog(x) = \begin{cases} 0 & \text{if } x = 0 \\ \log(|x|) & \text{otherwise.} \end{cases}$$

Fitness function is computed as follows.

$$fitness(f) = \frac{1}{1 + \sum_{i=1}^{20} |f(x_i) - y_i|}$$

where (x_i, y_i) are 20 data samples from $[-1, 1]$, such that $y_i = f_t(x_i)$. An individual f such that $|f(x_i) - y_i| < 0.01$ for all data samples is considered as a correct individual.

Figure 1 shows results of this experiment. Table II summarizes statistical analysis of the fitness values of the best individuals of the run ($p\text{-value} = 0.6038$). By conventional criteria, this difference is considered to be not statistically significant. Standard *ramped half-and-half strategy* scored 17/50 (34%) success rate. Minimal $I(M, i, z)$ was in generation 23 with 192,000 individuals to be processed. The average individual

TABLE II. STATISTICAL ANALYSIS - SIMPLE SYMBOLIC REGRESSION.

| | Ramped half-and-half | Geometric |
|---------|----------------------|---------------------------------------|
| Mean | 0.7999030182 | 0.8174067512 |
| SD | 0.1521369287 | 0.1826109800 |
| SEM | 0.0215154108 | 0.0258250925 |
| N | 50 | 50 |
| t-value | 0.5207 | $\alpha = 0.05$ |
| p-value | 0.6038 | Not statistically significant. |

TABLE III. STATISTICAL ANALYSIS - ARTIFICIAL ANT PROBLEM.

| | Ramped half-and-half | Geometric |
|---------|----------------------|-------------------------------------|
| Mean | 66.20 | 79.78 |
| SD | 12.33 | 8.77 |
| SEM | 1.74 | 1.24 |
| N | 50 | 50 |
| t-value | 6.3479 | $\alpha = 0.05$ |
| p-value | <0.0001 | Extremely stat. significant. |

size for generation 50 was 68.8. The experiment took 46 minutes. Our *geometric strategy* scored 21/50 (42%) success rate. Minimal $I(M, i, z)$ was in generation 29 with 150,000 individuals to be processed. The average individual size for generation 50 was 44.1. The experiment took 26 minutes. So for the *Simple symbolic regression* our geometric strategy is slightly more successful than ramped half-and-half strategy in these observed metrics.

B. Artificial Ant

Artificial Ant is another problem described in [1]. Objective of this problem is to find a control program for an artificial ant so that it can find all food located on "Santa Fe" trail. The Santa Fe trail lies on toroidal square grid. The ant is in the upper left corner, facing right. The ant is able to move forward, turn left, and sense if a food piece is ahead of him.

$$\tau = AntAct$$

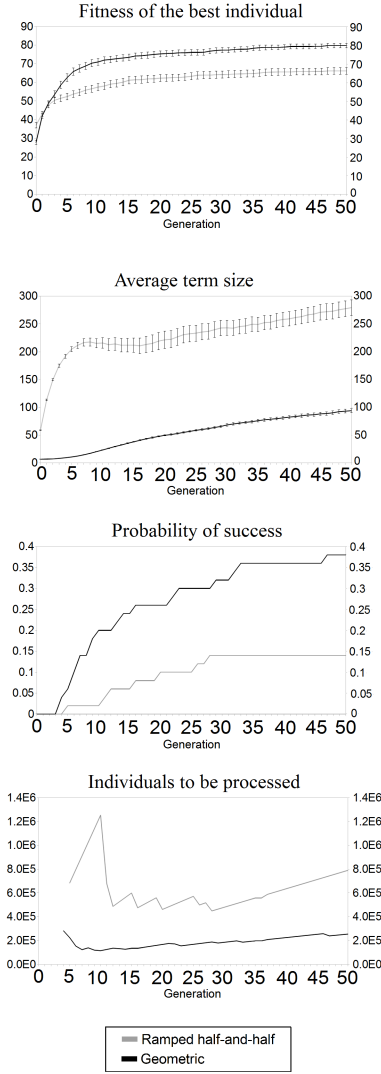
$$\Gamma = \{ l : AntAct, r : AntAct, m : AntAct, \\ ifa : AntAct \rightarrow AntAct \rightarrow AntAct, \\ p2 : AntAct \rightarrow AntAct \rightarrow AntAct, \\ p3 : AntAct \rightarrow AntAct \rightarrow AntAct \rightarrow AntAct \}$$

Actions l and r turn the ant left and right, respectively. Action m moves the ant forward. Action $ifa\ x\ y$ (*if-food-ahead*) performs action x if a food piece is ahead of the ant, otherwise it performs action y . Actions $p2$ and $p3$ perform two and three consecutive actions, respectively. Actions l, r and m each take one time step to execute. Ants action is performed over and over again until it reaches predefined maximal number of steps. Fitness value is equal to number of eaten food pieces. An individual such that eats all 89 pieces of food is considered as a correct solution. This limit is set to be 600 time steps⁷.

Figure 2 shows results of this experiment. Table III summarizes statistical analysis of the fitness values of the best individuals of the run ($p\text{-value} < 0.0001$). According to this analysis our strategy performed better; by conventional criteria, this difference is considered to be extremely statistically significant. Standard *ramped half-and-half strategy* scored 7/50 (14%) success rate. Minimal $I(M, i, z)$ was in generation

⁷In [1] this limit is said to be 400 time steps. But there is also mentioned one solution, which is described as correct solution. This program needs 545 time steps; if it is given only 400 time steps, then it eats only 79 pieces of food. Thus we use 600 time steps.

Fig. 2. Graphs for Artificial ant problem.



28 with 449,500 individuals to be processed. The average individual size for generation 50 was 279.5. The experiment took 265 minutes. Our *geometric strategy* scored 19/50 (38%) success rate. Minimal $I(M, i, z)$ was in generation 10 with 115,500 individuals to be processed. The average individual size for generation 50 was 94.1. The experiment took 107 minutes. This is a big improvement in all watched factors.

C. Even Parity Problem

Previous two problems were instances of standard GP, i.e. their building symbols obeyed the closure requirement. This third experiment will break the closure requirement, thus ideal for testing typed GP techniques. The even-parity function is a boolean function taking as inputs N boolean values and returning *True* if an even number of inputs are *True*. For odd number it returns *False*. This problem has been used by many researchers as a benchmark for GP. We compare our results with that obtained by Yu in [4], where is presented approach to evolve recursive and modular programs by use of higher-order functions and λ -abstractions. We use very similar set of

building symbols as in [4].

$$\begin{aligned} \tau &= [Bool] \rightarrow Bool \\ \Gamma &= \{and : Bool \rightarrow Bool \rightarrow Bool, \\ &\quad or : Bool \rightarrow Bool \rightarrow Bool, \\ &\quad nand : Bool \rightarrow Bool \rightarrow Bool, \\ &\quad nor : Bool \rightarrow Bool \rightarrow Bool, \\ &\quad foldr : (Bool \rightarrow Bool \rightarrow Bool) \\ &\quad \quad \rightarrow Bool \rightarrow [Bool] \rightarrow Bool, \\ &\quad head' : [Bool] \rightarrow Bool, \\ &\quad tail' : [Bool] \rightarrow [Bool]\} \end{aligned}$$

The type $[Bool]$ stands for *list of Booleans* and for purpose of this problem is considered atomic. Unlike in [4], we use specific instance of polymorphic function *foldr*. And modifications of functions *head* (returning the first element of the list) and *tail* (returning the list without the first element) are used; making them total by returning default value *False* and $[]$, respectively. We use the same fitness function as in [4]. The fitness function examines the individual by giving it all possible boolean lists of length 2 and 3.

Fig. 3. Graphs for Even parity problem.

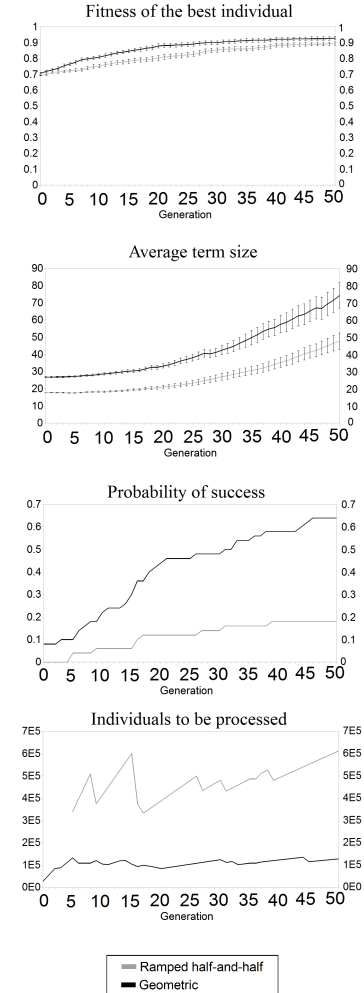


TABLE IV. STATISTICAL ANALYSIS - EVEN PARITY PROBLEM.

| | Ramped half-and-half | Geometric |
|---------|----------------------|----------------------------|
| Mean | 0.895000 | 0.930000 |
| SD | 0.075170 | 0.078490 |
| SEM | 0.010631 | 0.011100 |
| N | 50 | 50 |
| t-value | 2.2772 | $\alpha = 0.05$ |
| p-value | 0.0250 | Statistically significant. |

Figure 3 shows results of this experiment. Table IV summarizes statistical analysis of the fitness values of the best individuals of the run (p -value = 0.0250). According to this analysis our strategy performed better; by conventional criteria, this difference is considered to be statistically significant.

Standard *ramped half-and-half strategy* scored 9/50 (18%) success rate. Minimal $I(M, i, z)$ was in generation 17 with 333,000 individuals to be processed. The average individual size for generation 50 was 47.7. The experiment took 28 minutes. Our *geometric strategy* scored 32/50 (64%) success rate. Minimal $I(M, i, z)$ was in generation 0 with 28,000 individuals to be processed. The average individual size for generation 50 was 74.3. The experiment took 33 minutes.

Once again this is a big improvement against standard method.

One may get the impression that geometric strategy suffers from bloat here, but as is stated in [5]: “*We should not equate growth with bloat and we should define bloat as program growth without (significant) return in terms of fitness.*” Considering that it performs significantly better and that term size around 75 is similar as for the previous two problems, it seems to us that ramped half-and-half maybe has difficulties with constructing more complex terms, and therefore is “staying small”.

Another remarkable observation is that 4 times out of 50 (8% of all runs) the 100% correct solution was present in the generation 0. This is pretty good result for uninformed search. However, our results are less successful than those presented in [4] (with slightly different set of building symbols); they scored 40/50 (80%) success rate and $I(M, i, z)$ of 17,500 in generation 4. But by observation of typical solution which is often some modification of `foldr xor True inputList` one sees that important task for GP here is to create `xor` function. This is advantage for the more specialized term representation used in [4] (restricting use of outer variables). The difference in crossover operators may also be involved in the difference, but such analysis is beyond the scope of this paper. Finally, our result at least outperforms other results mentioned in [4]: *Generic genetic programming* scored 17/60 (28%); min $I(M, i, z) = 220,000$. *GP with ADFs* scored 10/29 (34%); min $I(M, i, z) = 1,440,000$.

VI. CONCLUSIONS

In this work, we have proposed a generalization of genetic programming for simply typed lambda calculus. The main focus of this paper is on individual generation algorithms. The main idea is to explore the spectrum of available approaches bounded on one end by the traditional ramped half-and-half strategy, while the second bound represents an exhaustive systematic search. Thus, three population initialization methods have been designed, depending on different search strategies.

The first strategy corresponds to the above mentioned traditional genetic programming initialization, the second one corresponds to exhaustive search, and the third one represents a novel geometric strategy which outperforms standard genetic programming in success rate, resource consumption and average individual size on the presented experiments.

Experiments on three well known benchmark problems show that the geometric strategy outperforms the standard ramped half-and-half generating method in success rate, best fitness value, time consumption and average individual size.

The relevance of these algorithms does not effect only initialization, but it can be used during mutation operator in a straightforward way.

Our future research will focus on exploring other possible search strategies positioned in the middle ground identified in this paper. Since the search strategy can be easily described by a rather simple filtration algorithm, it might be interesting to allow the typed GP system to evolve the search strategy it utilizes in a kind of meta-evolution way.

ACKNOWLEDGEMENTS

Tomáš Křen has been partially supported by the project GA ČR P202/10/1333 and by the SVV project number 260 104. Roman Neruda has been partially supported by the The Ministry of Education of the Czech Republic project COST LD 13002.

REFERENCES

- [1] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. Cambridge, MA, USA: MIT Press, 1992.
- [2] —, *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Norwell, MA, USA: Kluwer Academic Publishers, 2003.
- [3] D. Montana, “Strongly typed genetic programming,” *Evolutionary computation*, vol. 3, no. 2, pp. 199–230, 1995.
- [4] T. Yu, “Hierarchical processing for evolving recursive and modular programs using higher-order functions and lambda abstraction,” *Genetic Programming and Evolvable Machines*, vol. 2, no. 4, pp. 345–380, 2001.
- [5] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.
- [6] A. Moraglio, “One-point geometric crossover,” in *Parallel Problem Solving from Nature, PPSN XI*. Springer Berlin Heidelberg, 2010, pp. 83–93.
- [7] F. Briggs and M. O’Neill, “Functional genetic programming and exhaustive program search with combinator expressions,” *International Journal of Knowledge-Based and Intelligent Engineering Systems*, vol. 12, no. 1, pp. 47–68, 2008. [Online]. Available: <http://iospress.metapress.com/content/u614j13p67w66370/>
- [8] F. Binard and A. Felty, “Genetic programming with polymorphic types and higher-order functions,” in *Proceedings of the 10th annual conference on Genetic and evolutionary computation*. ACM, 2008, pp. 1187–1194.
- [9] H. Barendregt, W. Dekkers, and R. Statman, *Lambda Calculus With Types*. Cambridge University Press, 2010.
- [10] S. J. Russell and P. Norvig, *Artificial Intelligence - A Modern Approach (3rd Ed.)*. Prentice Hall, 2010.
- [11] S. L. Peyton Jones, *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987.
- [12] T. Yu and C. Clack, “PolyGP: A polymorphic genetic programming system in haskell,” *Genetic Programming*, vol. 98, 1998.