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

Tomáš Křen[1] and Roman Neruda[2]

[1] Faculty of Mathematics and Physics
Charles University in Prague
Malostranské náměstí 25, 11000 Prague, Czech Republic
tomas.kren@mff.cuni.cz
[2] Institute of Computer Science
Academy of Sciences of the Czech Republic
Pod Vodárenskou věží 2, 18207, Prague, Czech Republic
roman@cs.cas.cz

Abstract. In this paper, a generalization of genetic programming for simply typed lambda calculus is presented. We propose three population initialization methods depending on different search strategies. The first strategy corresponds to standard ramped half-and-half initialization, the second one corresponds to exhaustive systematic search, and the third one represents a novel geometric strategy. Three well-known benchmark experiments support that the geometric strategy outperforms the standard generating method in success rate, time consumption and average individual size.

1 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].

The key issue in the lambda calculus approach to enrich GP with types is the method of individual generation. During the expansion phase the set 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 *geometrical search strategy* 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 3 introduces necessary notions. Main original results about search strategies in individual generating are described in section 4. Section 5 presents results of our method on three well-known tasks, and the paper is concluded by section 6.

2 Related work

In [4] Yu presents a GP system utilizing polymorphic higher-order functions[3] and lambda abstractions. Important point of interest in this work is use of `foldr` 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.

In [6] Briggs and O'Neill present technique utilizing typed GP with combinators. 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 polymorphic type system then 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.

In [7] by Binard and Felty even stronger type system (*System F*) is used. 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 in [8].

[3] Higher-order function is a function taking another function as input parameter.

3 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.*

$$\begin{aligned} x \in V \cup C &\Rightarrow x \in \Lambda \\ M, N \in \Lambda &\Rightarrow (M\ N) \in \Lambda \qquad \textit{(Function application)} \\ x \in V, M \in \Lambda &\Rightarrow (\lambda\, x\, .\, M) \in \Lambda \qquad \textit{(λ-abstraction)} \end{aligned}$$

Function application and λ-*abstraction* are concepts well known from common programming languages. For example in JavaScript $(M\ N)$ translates to expression M`(`N`)` and $(\lambda\, x\, .\, M)$ translates to expression `function(`x`){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]. $M_1\ M_2\ M_3\ \ldots\ M_n$ is an abbreviation for $(\ldots((M_1\ M_2)\ M_3)\ \ldots\ M_n)$ and $\lambda\, x_1 x_2 \ldots x_n\, .\, M$ for $(\lambda\, x_1\, .\, (\lambda\, x_2\, .\, \ldots (\lambda\, x_n\, .\, M)\ldots))$.

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.*

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

Type $\sigma \to \tau$ is type for functions taking as input something of a type σ and returning as output something of a type τ. $\tau_1 \to \tau_2 \to \ldots \to \tau_n$ is an abbreviation for $\tau_1 \to (\tau_2 \to (\ldots \to (\tau_{n-1} \to \tau_n)\ldots))$. 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 $(x, \sigma) \in \Gamma$.

[4] Apart 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.

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.*

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

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.*

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

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

4 Our approach

4.1 Introduction

Our approach to λ-term gerating is based on technique briefly described in [8], which generates well-typed λ-terms in their long normal form. We use this technique to perform systematic exhaustive enumeration of λ-terms in their long normal form in order from smallest to largest. We use well known *A* algorithm* [9] 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" 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 statete 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.

4.2 Algorithm

The inputs for the term generating algorithm are following.

1. Desired type τ of generated terms.
2. Context Γ representing 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 size[5]. 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).

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).

[5] A* 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 silly 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 is should by pretty effective to compute this value precisely and store them for later.

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 rules[6].

1. If $L = \langle \rho_1 \to \ldots \to \rho_n \to \alpha; \Gamma \rangle$, where α is atomic type and $n \geq 1$, then L is replaced by $(\lambda\, x_1 \ldots x_n \,.\, \langle \alpha; \Gamma, x_1 \colon \rho_1, \ldots, x_n \colon \rho_n \rangle)$. Thus this expansion results in exactly one successor.
2. If $L = \langle \alpha; \Gamma \rangle$ where α is *atomic* type, then for each $f : (\tau_1 \to \ldots \to \tau_m \to \alpha) \in \Gamma$ the unfinished leaf L is replaced by $(\ f\ \langle \tau_1; \Gamma \rangle\ \ldots\ \langle \tau_m; \Gamma \rangle\ )$. Thus this expansion results in many (possibly zero or one) successors.

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 $\langle \tau; \Gamma \rangle$ is inserted to the queue, search strategy S again initializes its internal state and the process continues.

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 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 $\{2, \ldots, D_{init}\}$, 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

[6] For 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.

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 ore 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".

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*[10] 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 long normal form into $\beta\eta$-normal form. Another performence enhancing transformation is use of "applicative" tree representation (coming directly from inductive definition of λ-terms).

5 Experiments

We made tree experiments comparing performance of standard *ramped half-and-half strategy* with our *geometric strategy* (with default parameter $q = 0.75$). 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 ability of the system to produce a correct solution, computational cost estimates and average size of an individual throughout generations. For the first 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 $z = 99\%$).

5.1 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.

$$\tau = \mathbb{R} \to \mathbb{R}$$
$$\Gamma = \{(+) : \mathbb{R} \to \mathbb{R} \to \mathbb{R}, (-) : \mathbb{R} \to \mathbb{R} \to \mathbb{R}, (*) : \mathbb{R} \to \mathbb{R} \to \mathbb{R}, rdiv : \mathbb{R} \to \mathbb{R} \to \mathbb{R},$$
$$sin : \mathbb{R} \to \mathbb{R}, cos : \mathbb{R} \to \mathbb{R}, exp : \mathbb{R} \to \mathbb{R}, rlog : \mathbb{R} \to \mathbb{R}\}$$

where $$rdiv(p, q) = \begin{cases} 1 & \text{if } q = 0 \\ p/q & \text{otherwise} \end{cases} \qquad rlog(x) = \begin{cases} 0 & \text{if } x = 0 \\ log(|x|) & \text{otherwise.} \end{cases}$$

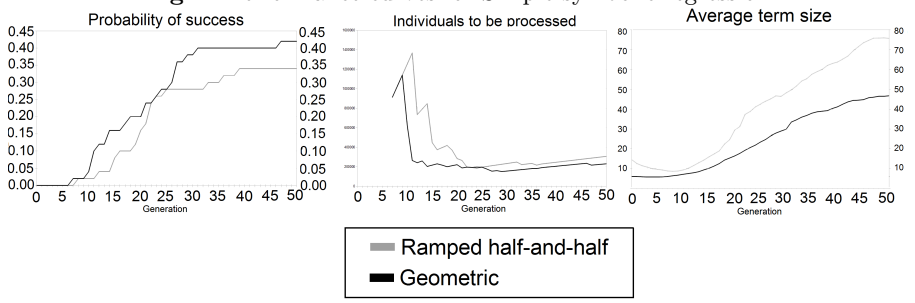
Fitness function is computed as follows.

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

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.

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 size for generation 50 was 76. The experiment took 46 minutes. Our *geometric strategy* scored 21/50 (42%) success rate. Minimal $I(M, i, z)$

Fig. 1. Performance curves for Simple symbolic regression.



was in generation 29 with 150,000 individuals to be processed. The average individual size for generation 50 was 47. The experiment took 26 minutes. Figure 1 shows the progress of those metrics throughout generations. So for the *Simple symbolic regression* our geometric strategy is slightly more successful then ramped half-and-half strategy in all observed metrics.

5.2 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.

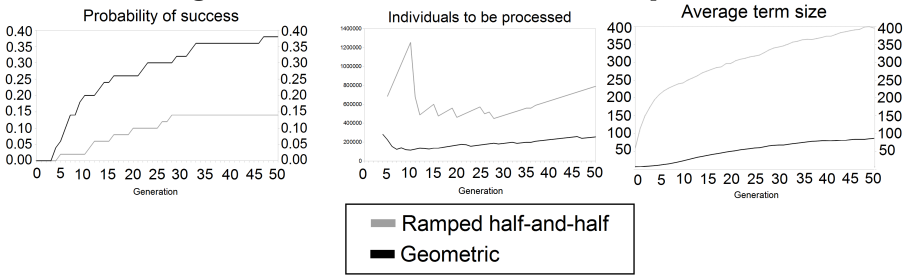
$$
\begin{aligned}
\tau &= AntAction \\
\Gamma &= \{\ \ l : AntAction, r : AntAction, m : AntAction, \\
&\quad\ ifa : AntAction \to AntAction \to AntAction, \\
&\quad\ \ p2 : AntAction \to AntAction \to AntAction, \\
&\quad\ \ p3 : AntAction \to AntAction \to AntAction \to AntAction\}
\end{aligned}
$$

Action l turns the ant left. Action r turns the ant right. 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. Action $p2\ x\ y$ first performs action x and after it action y. Action $p3\ x\ y\ z$ first performs action x, after that action y and finally z. 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 steps[7].

[7] In [1] this limit is said to be 400 time steps. But there is also mentioned following solution, which is described as correct solution: `(ifa m (p3 l (p2 (ifa m r) (p2`

Fig. 2. Performance curves for Artificial ant problem.



Standard *ramped half-and-half strategy* scored 7/50 (14%) success rate. Minimal $I(M, i, z)$ was in generation 28 with 449,500 individuals to be processed. The average individual size for generation 50 was circa 400. 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 circa 90. The experiment took 107 minutes. This is a big improvement in all watched factors. Figure 2 shows it in greater detail.

5.3 Even parity problem

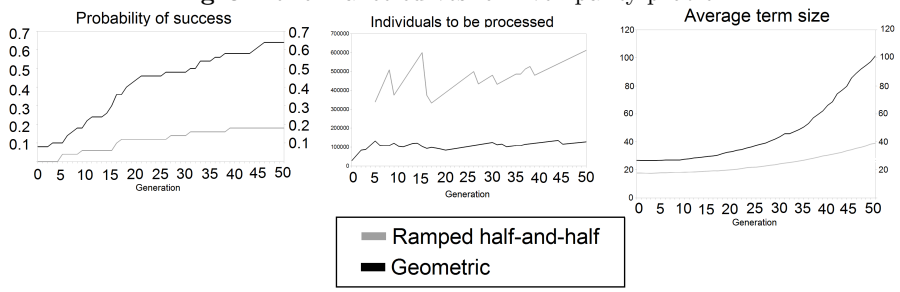
Previous two problems were instances of standard GP, i.e. their building symbols obeyed the closure requirement. This third experiment will break the closure requirement, thus ideal for testing typed GP techniques. The even-parity function is a boolean function taking as inputs N boolean values and returning *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].

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

The type $[Bool]$ stands for *list of Bools* and for purpose of this problem is considered atomic. Unlike in [4], we use specific instance of polymorphic function

`r (p2 l r)))(p2 (ifa m l) m)))`. 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. 3. Performance curves for Even parity problem.



`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.

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 39. 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 circa 100. The experiment took 33 minutes. Figure 3 shows it in greater detail. Once again this is a big improvement against standard method. One may get the impression that geometric strategy suffers from bloat here, but considering that it performs significantly better and that term size around 100 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 then 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 False 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 $I(M, i, z) = 1,440,000$.

6 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.

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.

References

1. Koza, J.R.: Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge, MA, USA (1992)
2. Koza, J.R.: Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers, Norwell, MA, USA (2003)
3. Montana, D.: Strongly typed genetic programming. Evolutionary computation **3**(2) (1995) 199–230
4. Yu, T.: Hierarchical processing for evolving recursive and modular programs using higher-order functions and lambda abstraction. Genetic Programming and Evolvable Machines **2**(4) (2001) 345–380
5. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programming. Lulu Enterprises, UK Ltd (2008)
6. Briggs, F., O'Neill, M.: Functional genetic programming and exhaustive program search with combinator expressions. International Journal of Knowledge-Based and Intelligent Engineering Systems **12**(1) (2008) 47–68
7. Binard, F., Felty, A.: Genetic programming with polymorphic types and higher-order functions. In: Proceedings of the 10th annual conference on Genetic and evolutionary computation, ACM (2008) 1187–1194
8. Barendregt, H., Dekkers, W., Statman, R.: Lambda Calculus With Types. Cambridge University Press (2010)
9. Russell, S.J., Norvig, P.: Artificial Intelligence - A Modern Approach (3rd Ed.). Prentice Hall (2010)
10. Peyton Jones, S.L.: The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1987)