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

Tomáš Křen Roman Neruda

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.**

I. Introduction

GENETIC programming (GP) represents an efficient method for automatic generating of programs by means of evolutionary techniques [2], [**?**]. Early attempts to enhance the GP approach with the concept of types include the seminal work [6] 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 [5].

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 [4] 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 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 **??**.

II. Related work

In [5] Yu presents a GP system utilizing polymorphic higher-order functions[1] 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 [9] 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 [**?**] 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[2].

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 [12].

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

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.

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

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*[2]. $M_1\ M_2\ M_3\ \ldots\ M_n$ 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.

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

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 $(x, \sigma) \in \Gamma$.

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

$$x : \sigma \in \Gamma \ \Rightarrow\ \Gamma \vdash x\colon \sigma$$
$$\Gamma \vdash M\colon \sigma \to \tau\ ,\ \Gamma \vdash N\colon \sigma \ \Rightarrow\ \Gamma \vdash (M\ N)\colon \tau$$
$$\Gamma, x : \sigma \vdash M\colon \tau \ \Rightarrow\ \Gamma \vdash (\lambda x\,.\,M)\colon \sigma \to \tau$$

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

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.

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

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 gerating is based on technique briefly described in [12], 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* [**?**] 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

[2]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.

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

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[3]. 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).

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[4].

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 $(\lambda\, x_1 \ldots x_n \,.\, \langle \alpha; \Gamma, x_1 : \rho_1, \ldots, x_n : \rho_n \rangle)$. 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.

1) 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*.

2) Ramped half-and-half strategy: is generalization of the standard ramped half-and-half method described in [2]. 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 $depth$ 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.

3) 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

[3]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.

[4]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.

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*[13] 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).

V. Experiments

VI. Conclusions

Dodělat závěr (...)

References

[1] TODO. Todo, *Dát sem místo toho ten bibtex*. MIT Press, Cambridge, MA, 2014.

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

[3] Koza, J.R., Keane, M., Streeter, M., Mydlowec, W.,Yu, J., Lanza, G. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence.* Springer, 2005. ISBN 978-0-387-26417-2

[4] Riccardo Poli, William B. Langdon, Nicholas F. McPhee *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.

[5] T. Yu. *Hierachical processing for evolving recursive and modular programs using higher order functions and lambda abstractions*. Genetic Programming and Evolvable Machines, 2(4):345–380, December 2001. ISSN 1389-2576.

[6] D. J. Montana. *Strongly typed genetic programming.* Evolutionary Computation, 3(2): 199–230, 1995.

[7] T. D. Haynes, D. A. Schoenefeld, and R. L. Wainwright. *Type inheritance in strongly typed genetic programming.* In P. J. Angeline and K. E. Kinnear, Jr., editors, Advances in Genetic Programming 2, chapter 18, pages 359–376. MIT Press, Cambridge, MA, USA, 1996. ISBN 0-262-01158-1.

[8] J. R. Olsson. *Inductive functional programming using incremental program transformation and Execution of logic programs by iterative-deepening A* SLD-tree search.* Dr scient thesis, University of Oslo, Norway, 1994.

[9] Forrest Briggs, Melissa O'Neill. *Functional Genetic Programming and Exhaustive Program Search with Combinator Expressions.* International Journal of Knowledge-based and Intelligent Engineering Systems, Volume 12 Issue 1, Pages 47-68, January 2008.

[10] H. P. Barendregt, *The Lambda Calculus: its Syntax and Semantics*, revised ed., North-Holland, 1984.

[11] H. Barendregt , S. Abramsky , D. M. Gabbay , T. S. E. Maibaum. *Lambda Calculi with Types.* Handbook of Logic in Computer Science, 1992.

[12] Henk Barendregt, Wil Dekkers, Richard Statman, *Lambda Calculus With Types*. Cambridge University Press, 2010.

[13] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[14] Stuart J. Russell, Peter Norvig, *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.