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 which outperforms standard genetic programming in success rate, time consumption and average individual size on two experiments. Other performance enhancements based on theory of lambda calculus are proposed and supported by experiments, including abstraction elimination that enables us to utilize a simple tree-swapping crossover.

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.

TODO

- ... něco k tomu, že se to teda dělá i skrz lambda kalkul

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 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

2.1 Yu

Yu - (články: evenParity, polyGP **[doplnit do citací]** co sem ted našel, ten o burzách co mám v kindlu) Odlišnosti: (1) - generování se nedělá systematicky: pokud strom dojde do místa kde funkci nemá dát jaký parametr, tak místo tý funkce dá nějaký terminál.(uvádí 85% uspěšnost) (2) - Ty křížení ma trochu jinak než v tom článku o even parity, musim zjistit jak má udělaný že se jí nedostane např prom #3 někam, kde neni definovaná když dělá přesun podstromu, v tom starym to ale bylo myslim založený na tom, že nedovolovala vnější proměnný uvnitř lambda termu - což platí i nadále. čili vtom to určitě nehraje full deck. Naopak se víc zaměřuje na polymorfizmus a další věci.

2.2 kombinatori

kombinátoři
- vůbec nepoužívaj Lambda Abstrakce
- univerzální genetickej operator
- taky řešej systematický prohledávání

2.3 kanadani

kanadani
- o dost silnější typovej systém
- System F, i jim dynamicky vznikaj typy
- moc silný typoví systém, takže generování už je dost složitý, to se otiskuje v silně nestandardním algoritmu

2.4 barendregt

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.*

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

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*[3].

TODO

- asi říct o zkratkách notací

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.

TODO

- asi říct o zkratce notace

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 $M : \sigma$.*

[3] 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.

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.

TODO

- říct že $T \cup F$ v Std GP je u nás Γ
- popsat $\Gamma, x : \tau$ notaci a to že to automaticky implikuje že x neni obsažena v Γ, tzn je nová...
- že píšem $x : \sigma \in \Gamma$ místo $(x, \sigma) \in \Gamma$

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.*

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

...Our goal 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.*

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

TODO

- Vysvětlit k čemu je to dobrý. Prostě říct že to $\langle \tau ; \Gamma \rangle$ představuje zatim nespecifikovaný term který je ale typu τ a mohu se v něm objevit stavební symboly z Γ.

4 Our approach

4.1 Introduction

Our approach to λ-term gerating is based on technique briefly described in [11], 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* [13] for this task. A* is used to search in a given state space for a goal state. 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.

TODO

- Kde nejlíp zmínit citaci diplomky? Něco jako, popisujem to tu velmi ve ztručnosti, more elaborate description may be found in [diplomka]?
- definovat long normal form? (asi nejde uplně krátce bohužel)

4.2 Generating 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. 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 $\Gamma \vdash M\colon \tau$), then it is added to the result set[4] 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.

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 $(\ f\ (\tau_1, \Gamma)\ \ldots\ (\tau_m, \Gamma)\ )$. 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.

TODO

- Okomentovat jak se počítá velikost a při tý příležitosti říct že A* heuristika se skrejvá v tom jak velký určíme unfinished leafs, že my používáme 1, ale že si de představit o dost sofistikovanější verze.

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.*

[4] TODO: říct že tim že je to set se řeší problém toho, aby se tam nevyskytovali některý vygenerovaný víckrát, pokud však toto nechcem můžem použít jinou collection , případně to vůbec nezminovbat když by to kradlo moc místa

Ramped half-and-half strategy 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 - A 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 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.

TODO

- napsat o tom že ji používa koza [citace] a že je to standard, pokud je použita na gamu co splnuje closure podmínku, tak generuje přesně stejně jako
- ...This means that the queue always contains only one (or zero) state.

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".

4.3 Discussion and further improvements

4.4 η-normalization

(Otázka zda vůbec zminovat?)

- pač je to generovany v lnf, neboli $\beta\eta^{-1}$-nf kde η^{-1} je η-expanze tak je chytrý transformovat to do $\beta\eta$-nf. To stačí opakovanou η-redukcí, protože beta normálnost se neporuší (asi ve zkratce uvést proč, pač je to celkem přímočarý)

4.5 Crossover

Zmínil bych jí tu ale jen hodně rychle..., nutno zmínit pokud budem uvádět even parity problém

5 Experiments

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 σ and building blocks context Γ are following.

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

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

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.

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.

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

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[5] time steps.

5.3 Even parity problem

...

6 Conclusions

TODO

- do future work bych napsal, že díky jednoduchosti toho co strategie dělá je to idealní kandidát na optimalizaci pomocí samotnýho GP

References

1. Koza, J.R.: Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge, MA, USA (1992)

[5] 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 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.

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. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programming. Lulu Enterprises, UK Ltd (2008)
5. 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
6. Haynes, T.D., Schoenefeld, D.A., Wainwright, R.L.: Type inheritance in strongly typed genetic programming. In Angeline, P.J., Kinnear, Jr., K.E., eds.: Advances in Genetic Programming 2. MIT Press, Cambridge, MA, USA (1996) 359–376
7. Olsson, J.R.: 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)
8. 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
9. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics (Studies in Logic and the Foundations of Mathematics). Revised edition edn. Elsevier Science (1984)
10. Barendregt, H.: Lambda calculi with types. In Abramsky, S., Gabbay, D.M., Maibaum, T.S.E., eds.: Handbook of Logic in Computer Science, Oxford University Press (1992) 117–309
11. Barendregt, H., Dekkers, W., Statman, R.: Lambda Calculus With Types. Cambridge University Press (2010)
12. 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)
13. Russell, S.J., Norvig, P.: Artificial Intelligence - A Modern Approach (3rd Ed.). Prentice Hall (2010)