[něco jako?] Generating lambda term individuals using forgetful A*

Tomáš Křen and Roman Neruda

Matfyz ...

Abstract. ***(Aby tu něco bylo, pak se udělá lepší (tohle je trochu upravenej abstract z diplomky))*** In this paper, generalization of the standard genetic programming (GP) for simply typed lambda calculus is presented. We use population initialization method parameterized by simple search strategy. First described strategy corresponds to standard ramped half-and-half method, second one corresponds to exhaustive systematic search and third one is a novel geometric strategy, which outperforms standard method in success rate, time consumption and average individual size in two experiments. Other performance enhancements based on theory of lambda calculus are proposed and supported by experiment. Abstraction elimination is utilized to enable use of simple tree-swapping crossover.

1 Introduction

1.1 gp a typy jsou dobrý / motivace

...

1.2 příběh

(...)

Our approach aims to play with the full arsenal given by simply typed lambda calculus, thus we begin our reasoning with an exhaustive systematic search in mind. Our second goal is to construct a system generalizing Standard GP [1]. In order to satisfy both these goals the designed system should be parameterized by some simple piece of code that makes the difference between exhaustive systematic search and standard but quiet arbitrary ramped half-and-half generating method.

Those two design goals also differentiate our system from the three state of the art systems for typed GP known to us.

(...)

poznamky naky
- geometrická strategie generování termů jakožto hybrid systematického a náhodného

sytylu — to že jsme si vytičili ty dva cíle který plníme tou jednoduchou strategií, tak máme možnost nalízt jednoduchou strategii která je na pomezí obou cílů - taková strategie je právě ta naše geometrická a ukazuje se že se bohulibě chová právě k strašáku GP – ***bloatu****.*
- teoretické LC konstrukty s výhodou použity - @-stromy a eta-redukce
- křížení by eliminace

1.3 obsah kapitol článku v 1 větě

...

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

Let us describe programming language, in which we generate individual programs — 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*. A λ-term as described above corresponds to a program expression with no type information included. Now we will describe *types* (or *type terms*). After putting those two pieces (*λ-terms* and *types*) together we will get system called *simply typed λ-calculus*.

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.

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.*

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.

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

4 Our approach

4.1 Introduction

TODO

- úvod ke kapitole - systematicky, A*, filtrace, znova dám do fronty začátek

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 actual number of those leafs 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 collection 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 $(\lambda\, x_1 \ldots x_n \,.\, \langle \sigma; \Gamma, x_1 \colon \rho_1, \ldots, x_n \colon \rho_n \rangle)$. 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.*

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

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.

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

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.

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.

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

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[1] 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. John R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, 1992.
2. 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
3. Riccardo Poli, William B. Langdon, Nicholas F. McPhee *A Field Guide to Genetic Programming.* Lulu Enterprises, UK Ltd, 2008.
4. 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.
5. D. J. Montana. *Strongly typed genetic programming.* Evolutionary Computation, 3(2): 199–230, 1995.
6. 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.
7. 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.
8. 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.

[1] 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.

9. H. P. Barendregt, *The Lambda Calculus: its Syntax and Semantics*, revised ed., North-Holland, 1984.
10. H. Barendregt , S. Abramsky , D. M. Gabbay , T. S. E. Maibaum. *Lambda Calculi with Types.* Handbook of Logic in Computer Science, 1992.
11. Henk Barendregt, Wil Dekkers, Richard Statman, *Lambda Calculus With Types.* Cambridge University Press, 2010.
12. Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.
13. Stuart J. Russell, Peter Norvig, *Artificial Intelligence: A Modern Approach.* Pearson Education, 2003.