Typed Functional Genetic Programming

Tomáš Křen

Roman Neruda

***Abstract*— 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.**

I. Introduction

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.

[Kurzívou jsou neformální poznámky]

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.*

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

kombinátoři

- vůbec nepoužívaj Lambda Abstrakce

- univerzální genetickej operator

- taky řešej systematický prohledávání

A eště tak nak všeobecně

- 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

II. Generating method

- Unifished term je term který může obsahovat jako list uzel tvaru (τ, Γ), takzvaný unfinished leaf (UL).

- Pracujeme s prioritní frontou rozdělaných lambda termů.

- Priorita je dána počtem UL v termu.

- Na počátku je do fronty vložen jediný UT = (τ, Γ)

- Z fronty vyndám term s nejnižším #UL, pokud má #UL = 0, pak je to vygenerovaný jedinec, pokud ne tak vezmu jeho DFS-první UL a expanduju ho. Výsledkem expandování je několik (possibly 0) unfin. termů, které jsou až na expandovaný uzel totožné s původním UT, daný UT je nahrazen novým pod-UT.

- Expandování probíhá následovně:

(a) je to $(\sigma \rightarrow \tau, \Gamma)$ pak tento list nahradim novým podstromem $\lambda\, x\, .\, (\tau; \Gamma, x : a)$

(b) je to (α, Γ) pak pro každý

$f : (\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \alpha) \in \Gamma$

nahradíme tento list novým podstromem

$(\ f\ (\tau_1, \Gamma)\ \ldots\ (\tau_n, \Gamma)\ )$

- Prohledávací strategie má pak funkci filtru následníků.

- Pokud se fronta vyprázdní ještě před vygenerováním požadovaného počtu termů tak zbytek dogenerujeme stejným zpusobem, tzn do fronty přihodíme (τ, Γ) a jedem dál

- Systematická strategie nezahodí nic (Pro ní generování kolapsuje do A algoritmu).*

- Ramped-half-and-half zahodí vždy vše krom jediného.

- Geometrická strategie nezahodí s pravděpodobností q na hloubka UL

η-normalization ...

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

III. Crossover

[Odtud dál jsou zkrácené pasáže z diplomky,co mi přišli celkem ok, ale ještě budou určitě editovány

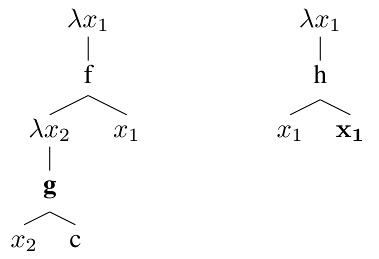
(zkracování, doplňování, vhodnější formulace,...)]

Design goal behind our approach to crossover operation is to try to generalize standard tree swapping crossover. The crossover operation in standard GP is performed by swapping randomly selected subtrees in each parent S-expression.

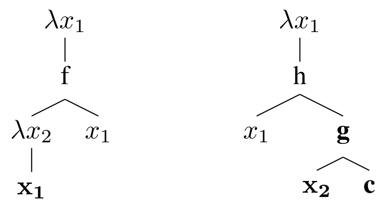
For typed lambda terms two difficulties arise: Types and variables. We will show how to crossover typed λ-term trees in both @-tree and sexpr-tree notation. As in standard GP our crossover will be performed by swapping two subtrees. But now with constraint that both subtrees have the same type.

Variables bring more difficulties then types do. This problem arises from variables that are free in subterms corresponding to swapped subtrees.

Following example illustrates the problem. Let us have these two parent trees with selected nodes in bold.



The swap of subtrees results in following trees:



The problem is that variable x_2 in the second tree is not bound by any λ-head and since it is not element of Γ, the second tree is not well-typed λ-term.

We can avoid dealing with this problem by avoiding use of variables. This can be achieved by process called abstraction elimination.

A. Abstraction elimination

Abstraction elimination is a process of transforming an arbitrary λ-term into λ-term that contains no lambda abstractions and no bound variables. The newly produced λ-term may contain function applications, free symbols from former λ-term and some new symbols standing for combinators $\mathbf{S}$, $\mathbf{K}$ and $\mathbf{I}$.

Those combinators are defined as:

$$\begin{aligned}\mathbf{S} &= \lambda\, f\, g\, x\, .\, f\, x\, (g\, x)\\ \mathbf{K} &= \lambda\, x\, y\, .\, x\\ \mathbf{I} &= \lambda\, x\, .\, x\end{aligned}$$

Let us describe transformation Æ performing this process.

$$\begin{aligned}
\text{Æ}[x] &= x \\
\text{Æ}[\,(M\,N)\,] &= (\text{Æ}[M] \quad \text{Æ}[N]) \\
\text{Æ}[\lambda\, x\, .\, x] &= \mathbf{I} \\
\text{Æ}[\lambda\, x\, .\, M] &= (\mathbf{K} \quad \text{Æ}[M]) && \textbf{if } x \notin \mathrm{FV}(M) \\
\text{Æ}[\lambda\, x\, .\, (\lambda\, y\, .\, M)] &= \text{Æ}[\lambda\, x\, .\ \text{Æ}[\lambda\, y\, .\, M]] && \textbf{if } x \in \mathrm{FV}(M) \\
\text{Æ}[\lambda\, x\, .\, (M\, N)] &= (\mathbf{S} \quad \text{Æ}[\lambda\, x\, .\, M] \quad \text{Æ}[\lambda\, x\, .\, N]) && \textbf{if } x \in \mathrm{FV}(M) \\
& && \vee x \in \mathrm{FV}(N)
\end{aligned}$$

This is simple version of this process. More optimized version, in the means of the size of resulting term and its performance is following one, presented in [12].

This version operates with more combinators:

$$\begin{aligned}
\mathbf{B} &= \lambda\, f\, g\, x\, .\, f\, (g\, x)\\
\mathbf{C} &= \lambda\, f\, g\, x\, .\, f\, x\, g\\
\mathbf{S}' &= \lambda\, c\, f\, g\, x\, .\, c\, (f\, x)\, (g\, x)\\
\mathbf{B}{*} &= \lambda\, c\, f\, g\, x\, .\, c\, (f\, (g\, x))\\
\mathbf{C}' &= \lambda\, c\, f\, g\, x\, .\, c\, (f\, x)\, g
\end{aligned}$$

And the transformation can be written as follows.

$$\begin{aligned}
\text{Æ}[x] &= x\\
\text{Æ}[\,(M\,N)\,] &= (\text{Æ}[M] \quad \text{Æ}[N])\\
\text{Æ}[\lambda\, x\, .\, M] &= A[x; \text{Æ}[M]]
\end{aligned}$$

$$\begin{aligned}
A[x;x] &= \mathbf{I}\\
A[x;y] &= (\mathbf{K}\ y)\\
A[x;\ (M\,N)\,] &= \mathit{Opt}[\ \mathbf{S}\ (A[x;M])\ (A[x;N])\ ]
\end{aligned}$$

$$\begin{aligned}
\mathit{Opt}[\ \mathbf{S}\ (\mathbf{K}\ M)\ (\mathbf{K}\ N)\ ] &= \mathbf{K}\ (M\ \ N)\\
\mathit{Opt}[\ \mathbf{S}\ (\mathbf{K}\ M)\ \mathbf{I}\ ] &= M\\
\mathit{Opt}[\ \mathbf{S}\ (\mathbf{K}\ M)\ (\mathbf{B}\ N\ L)\ ] &= \mathbf{B}{*}\ M\ \ N\ \ L\\
\mathit{Opt}[\ \mathbf{S}\ (\mathbf{K}\ M)\ N\ ] &= \mathbf{B}\ M\ \ N\\
\mathit{Opt}[\ \mathbf{S}\ (\mathbf{B}\ M\ \ N)\ (\mathbf{K}\ L)\ ] &= \mathbf{C}'\ M\ \ N\ \ L\\
\mathit{Opt}[\ \mathbf{S}\ M\ (\mathbf{K}\ N)\ ] &= \mathbf{C}\ M\ \ N\\
\mathit{Opt}[\ \mathbf{S}\ (\mathbf{B}\ M\ \ N)\ L\ ] &= \mathbf{S}'\ M\ \ N\ \ L
\end{aligned}$$

As is stated in [12], the biggest disadvantage of this technique is that the translated term is often much larger than in its lambda form — the size of the translated term can be proportional to the square of the size of the original term.

But the advantage is also tempting — no need to deal with variables and lambda heads.

B. Typed subtree swapping

First thing to do in standard subtree swapping is to select random node in the first parent.

We modify this procedure so that we allow selection only of those nodes with such a type that there exists a node in the second parent with the same type.

Standard subtree swapping crossover as a first thing selects whether the selected node will be inner node (usually with probability $p_{ip} = 90\%$) or leaf node (with probability 10%).

We are in a more complicated situation, because one of those sets may be empty, because of allowing only nodes with possible "partner" in the second parent. Thus we do this step only if both sets are nonempty.

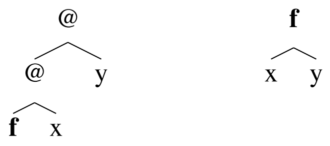
After selecting a node in the first parent we select node in the second parent such that type of that node must by the same as the type of the first node. Again, this may eliminate the "90-10" step of first deciding whether the selected node will be internal node or leaf node.

When both nodes are selected we may swap the trees.

If the abstraction elimination was performed, then since the trees are of the same type and there are no variables to be moved from their scope, the offspring trees are well typed.

Both sexpr-tree and @-tree are able to be crossed by this mechanism. But @-tree has more possibilities then @-tree. This comes from the fact that every subtree of the sexpr-tree corresponds to a subtree of @-tree, but there are subtrees of @-tree that do not correspond to a subtree of a sexpr-tree.

Following example should clarify this.



In @-tree, **f** is leaf thus subtree, whereas in sexpr-tree it is internal node thus not a subtree.

Another nice property of sexpr-trees with no lambdas is that they are the same representation as S-expressions used by standard GP.

Again, similarly as for standard version, a maximum permissible depth $D_{created}$ for offspring individuals is defined (e.g. $D_{created} = 17$). If one of the offspring has greater depth than this limit, then this offspring is replaced by the first parent in the result of the crossover operator. If both offspring exceeds this limit, than both are replaced by both parents.

For @-tree the $D_{created}$ must be larger since @-tree (without lambdas) is a binary tree. This enlargement is approximately proportionate to average number of function arguments. We use generous $D_{created} = 17 \times 3$.

IV. Experiments

A. Simple Symbolic Regression

Simple Symbolic Regression is a problem described in [1]. Objective of this problem is to find a function $f(x)$ that fits a sample of twenty given points. The target function is function $f_t(x) = x^4 + x^3 + x^2 + x$.

Desired type of generated programs σ and building blocks context Γ are following.

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

where

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

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

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.

B. Artificial Ant

C. Even-parity problem

V. Conclusions

Our goal was to design and implement a system performing GP over some typed functional programming language. As this typed functional programming language we have chosen the *simply typed lambda calculus*. We have also obligated ourselves to do it in a such way that generalizes the standard GP, rather than crates completely new system.

We may say that we have chosen the simplest possible option in both areas — the simply typed calculus and the standard GP. We see this decision as fortunate one — since it enables us add more complex features after sufficient exploration of those simplest cases.

We have started the journey toward this goal by study of subject that intersects both those areas: Method which for desired type and context generates lambda terms.

The way to obtain this method started with inference rules (the defining concepts of the typed lambda calculus), it proceeded through term generating grammars and finished with inhabitation trees.

A* algorithm in combination with inhabitation trees has been utilized to drive systematic enumeration of typed λ-terms in their *lnf*. This enumeration was further parameterized by simple search strategy in order to enable such different approaches as *systematic* generation and *ramped half-and-half* generation to be definable in the means of simple search strategy.

With this apparatus in hands we introduced novel approach to term generation by defining the *geometric* search strategy. This strategy is further parameterized by parameter q, but since we wanted to avoid suspicion that success of this generating method depends on fine-tuning of this parameter, we used its default value $q = 0.75$ in every experiment.

After being generated all terms undergo process of *abstraction elimination*, which enables our simple tree swapping crossover operation which is only genetic operator that we have used in all experiments.

We examined it in three different experiments, which all supported the idea that it has very desirable qualities.

First two experiments, *Simple Symbolic Regression* and *Artificial Ant*, were performed in order to compare the *geometric* strategy with the standard GP term generating method *ramped half-and-half*.

In both experiments were observed improvements in the following aspects.

1) Success rate was improved.
2) Minimal number of individuals needed to be processed in order to yield correct solution with probability 99% was lowered.
3) Run time was significantly reduced.
4) Average size of a term was decreased.

In the case of Artificial Ant problem, all improvements were significant.

These results make me believe that *geometric* strategy might be welcomed reinforcements in the fight against the bogey of the GP community — the *bloat*. Or at least, it seems to work for those two experiments in a such way.

In the third experiment, the *even-parity* problem, geometric strategy showed ability to yield correct solution in the initial generation 8 times out of 200 runs (throughout 5 experiments), which is interesting result since it is uninformed search.

Five consecutive experiments with even-parity problem supported hypothesis that η-normalization of generated terms enhances performance. We have also seen that use @-trees instead of more traditional sexpr-trees as tree representation of individuals is able to enhanced performance.

Use of optimized abstraction elimination instead of its basic variant showed significant improvements in time consumption with little or non effects on performance rates.

Implemented system was designed with importance of interactivity in mind, resulting in server/client architecture for core/GUI components.

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.

[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.