Utilization of Reductions and Abstraction Elimination in Typed Genetic Programming

Tomáš Křen
Faculty of Mathematics and Physics
Charles University in Prague
Malostranské náměstí 25, 11000,
Prague, Czech Republic
tomkren@gmail.com

Roman Neruda
Institute of Computer Science
Academy of Sciences of the Czech Republic
Pod Vodárenskou věží 2, 18207,
Prague, Czech Republic
roman@cs.cas.cz

ABSTRACT

Lambda calculus representation of programs offers a more expressive alternative to traditional S-expressions. In this paper we discuss advantages of this representation coming from use of reductions (beta and eta) and how to overcome disadvantages caused by variables occurring in the programs by use of the abstraction elimination algorithm. We discuss the role of those reductions in the process of generating initial population and compare several crossover approaches including novel approach to crossover operator based both on reductions and abstraction elimination. The design goal of this operator is to turn the disadvantage of abstraction elimination - possibly quadratic increase of program size - into a virtue; our approach leads to more crossover points. At the same time, utilization of reductions provides offspring of small sizes.

Categories and Subject Descriptors

H.4 [**TODO**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

TODO

Keywords

TODO, ToDo, todo

1. PLÁN BOJE

1.1 Jak to celý pojmout?

Pokud chceme dělat GP nad stromama větší vyjadřovací síly než maj klasický S-výrazy, tak máme v záse dvě přirozený možnosti pro obecný řešení: buď to dělat celý v (polymorfních) kombinátorech od začátku (a vyhnout se tak proměnejm a lambda abstrakcím), nebo to skusit v lambda kalkulu a nejak se vypořádat s proměnejma a lambdama. Motivace pro práci s lambda termama je, že za nima košatá teorie, která například popisuje redukce lambda termů (ty zajišťujou jak zmenšení samotných stromu, tak to, že prohledávací prostor se zmenší, díky tomu, že se různé stromy redukují na ten samej).

Jak se vypořádat s proměnejma a lambdama? (aka jak křížit)

- Po vy generování převod do kombinátorů pomocí eliminace abstrakcí (tak jak se to dělá v diplomce). To ale působí v něčem neohrabaně, když to porovnáme s generováním přímo v kombinatorech, když vezmeme v potaz to že eliminace abstrakcí má za důsledek až kvadratickej nárůst stromu - čili to co sme nahnali na redukcích stratíme eliminací.

- Jedince držíme jako redukovane malinké-kompaktní-a-elegantní lambda termy. Ve chvíly kdy křížíme provedeme na obou rodičích eliminaci abstrakcí (která ubere proměny a lambdy a namísto toho tam dá kombinátory S,K,I případně i další při fikanějších eliminacích), tím nám sice narostou, ale my z toho máme jedine radost, protože tím se nám zvýšil počet míst ke křížení (což se ukazuje jako dobráv věc, viz s-expr reprezentace vs @-tree reprezentace lambda termů). Po skřížení se vložený kombinátory nahradí odpovídajícim lambda termem (tzn např všude kde je K dám (x y . x) atd) výslednej term zredukuju a dostávam zase malinké-kompaktní-a-elegantní dítě. Nevýhoda toho zahrnout do článku i tohle je v tom, že k tomu nemám ještě žádný pokusy - ale k tomu zbytku mám upřímě v zato taky dost ubohý pokysy, takže toho bych se asi nebál. Většinu potřebnýho kodu bych k tomu ale měl už mít víceméně hotovou, takže pokud by se to na něčem nezaseklo, tak myslim že je realný udělat i pokus do toho dvacátýho. Zvlášť přitažlivý mi tohle křížení příde i kvůli tomu, že v přírodě se taky rozbalujou a zabaloujou chromozomi při meioze/mitoze.

1.2 Osnova

- Jak reprezentovat stromy programu pro GP?
  - Reprezentace v klasickym kozovy je S-expression.
  - Nebo mužem používat kombinátory jako Briggs a O'Neil (to si myslim je jakoby hlavní konkurence, vuči který by to chtělo obhájit)

GECCO'14, July 12-16, 2014, Vancouver, BC, Canada.

  - Lambda termy a jejich awesomeness
- **Povídaní o redukcích**
- Povídání o lnf
  - že lnf je přirozený rozšíření s-exprešnu do lambda kalkulu vlastnosti termu v lnf
    * proč je eta redukovat (eta redukcí lnf dostanem beta-eta-nf a že nemusíme beta redukovat pač se to tim nerozbyje)
- Generování
  - Generování gramatikou
  - Gramatika pro lnf
  - (?) Inhabitation trees jako intuitivní model takovýhodle lnf generování
- Problémy s proměnýma a jak je řešit.
  - Křížit lambda termy i s proměnejma a abstrakcema, problémy řešit když nastanou (pomluvit a odsoudit)
  - Zmenšit prostor termu (tim že někerý nejsme schopný vygenerovat) s kterým operujeme tak aby křížení už nebyl problém (to dělá Yu - v těle lambda fce dovoluje jen použití proměnných z její hlavy)
  - Převod hned po vygenerování
  - Převod až při křížení
    * eliminace abstrakcí
    * skřížim
    * vložený kombinátory nahradim odpovídajícím termem
    * celý to redukuju
- Pokusy (?!)
- Závěr

2. INTRODUCTION

Genetic programming (GP) represents an efficient method for automatic generating of programs by means of evolutionary techniques [4, 5]. 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 [8].

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 [7] 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 4 introduces necessary notions. Main original results about search strategies in individual generating are described in section 5. Section 6 presents results of our method on three well-known tasks, and the paper is concluded by section 7.

3. RELATED WORK

Yu presents a GP system utilizing polymorphic higher-order functions[1] and lambda abstractions [8]. 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.

Briggs and O'Neill present technique utilizing typed GP with combinators [3]. 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 than 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.

Binard and Felty use even stronger type system (*System F*) [2]. 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[4].

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 [1].

4. PRELIMINARIES

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

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.*

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

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*[2].

For better readability, $M_1\ M_2\ M_3\ \ldots\ M_n$ 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 $(\lambda\, x_1\, .\, (\lambda\, x_2\, .\, \ldots (\lambda\, x_n\, .\, M) \ldots))$.

4.1 β-reduction

In order to perform computation there must be some mechanism for term evaluation. In λ-calculus there is β-reduction for this reason.

A term of a form $(\lambda\, x\, .\, M)N$ is called β-*redex.* A β-redex can be β-reduced to term $M[x := N]$. This fact is written as *relation* $\rightarrow_\beta$ of those two terms:

$$(\lambda\, x\, .\, M)N \rightarrow_\beta M[x := N] \tag{1}$$

It is also possible to reduce *subterm β-redexes* which can be formally stated as:

$$\begin{aligned} P \rightarrow_\beta Q &\Rightarrow (R\ P) \rightarrow_\beta (R\ Q) \\ P \rightarrow_\beta Q &\Rightarrow (P\ R) \rightarrow_\beta (Q\ R) \\ P \rightarrow_\beta Q &\Rightarrow \lambda\, x\, .\, P \rightarrow_\beta \lambda\, x\, .\, Q \end{aligned}$$

In other words, β-reduction is the process of insertion of arguments supplied to a function into its body.

Another useful relations are $\twoheadrightarrow_\beta$ and $=_\beta$ defined as follows.

1. (a) $M \twoheadrightarrow_\beta M$
   (b) $M \rightarrow_\beta N \Rightarrow M \twoheadrightarrow_\beta N$
   (c) $M \twoheadrightarrow_\beta N, N \twoheadrightarrow_\beta L \Rightarrow M \twoheadrightarrow_\beta L$
2. (a) $M \twoheadrightarrow_\beta N \Rightarrow M =_\beta N$
   (b) $M =_\beta N \Rightarrow N =_\beta M$
   (c) $M =_\beta N, N =_\beta L \Rightarrow M =_\beta L$

We read those relations as follows.

1. $M \twoheadrightarrow_\beta N$ — "M β-reduces to N."
2. $M \rightarrow_\beta N$ — "M β-reduces to N in one step."
3. $M =_\beta N$ — "M is β-convertible to N."

[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.

4.2 η-reduction

Similarly as for β-reduction we can define η-reduction except that instead of 1 we use:

$$(\lambda\, x\, .\, (M\ x)) \rightarrow_\eta M \qquad \textbf{if } x \notin FV(M)$$

Analogically, a term of a form $(\lambda\, x\, .\, (M\ x))$ is called η-*redex.*

Relation $\rightarrow_{\beta\eta} = \rightarrow_\beta \cup \rightarrow_\eta$. (Relation $R = \{\ (a, b) \mid a\ R\ b\ \}$.)

Similarly as for $\twoheadrightarrow_\beta$ and $=_\beta$ we can define relations $\twoheadrightarrow_\eta$, $=_\eta$, $\twoheadrightarrow_{\beta\eta}$ and $=_{\beta\eta}$.

4.3 η^{-1}-reduction

η^{-1}-reduction (also called η-expansion) is the reduction converse to η-reduction. Again it may be obtained by replacing 1, now with:

$$M \rightarrow_{\eta^{-1}} (\lambda\, x\, .\, (M\ x)) \qquad \textbf{if } x \notin FV(M)$$

4.4 Normal forms

1. A λ-term is a β-*normal form* (β-*nf*) if it does not have a β-redex as subterm.
2. A λ-term M *has* a β-*nf* if $M =_\beta N$ and N is a β-*nf*.

A normal form may be thought of as a result of a term evaluation.

Similarly we can define η-*nf* and $\beta\eta$-*nf*.

4.5 Types etc

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.*

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

Type $\sigma \rightarrow \tau$ is type for functions taking as input something of a type σ and returning as output something of a type τ. $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n$ is an abbreviation for $\tau_1 \rightarrow (\tau_2 \rightarrow (\ldots \rightarrow (\tau_{n-1} \rightarrow \tau_n) \ldots))$. 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* $M : \sigma$ *is derivable from a context* Γ *(notation* $\Gamma \vdash M : \sigma$*) if it can be produced by the following rules.*

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

Přeformulovat aby se to hodilo do tohodle kontextu.... Our goal in term generation is to produce terms M for a given pair $\langle \tau; \Gamma \rangle$ such that for each M is $\Gamma \vdash M : \tau$.

4.6 Long normal form

Definition 5. *Let* $\Gamma \vdash M : \sigma$ *where* $\sigma = \tau_1 \to \ldots \to \tau_n \to \alpha, n \geq 0$.

1. *Then* M *is in long normal form (lnf) if following conditions are satisfied.*
   (a) M *is term of the form* $\lambda\, x_1 \ldots x_n \,.\, f\ M_1\ \ldots\ M_m$ *(specially for* $n = 0$, M *is term of the form* f*).*
   (b) *Each* M_i *is in lnf.*
2. M *has a lnf if* $M =_{\beta\eta} N$ *and* N *is in lnf.*

As is shown in [1], *lnf* has following nice properties.

Proposition 1. *If* M *has a* β*-nf, then it also has a unique lnf, which is also its unique* $\beta\eta^{-1}$*-nf.*

Proposition 2. *Every* B *in* β*-nf has a lnf* L *such that* $L \twoheadrightarrow_\eta B$.

4.7 Grammar producing λ-terms in *lnf*

In [1] is shown term generating grammar with following rules (our notation is used, but we will not highlighted terminals anymore).

$$
\begin{aligned}
(\alpha, \Gamma) &\longmapsto (\ f\ (\rho_1, \Gamma)\ \ldots\ (\rho_m, \Gamma)\ ) \quad \textbf{if } \alpha \in A, (f : \rho_1 \to \ldots \to \rho_m \to \alpha) \in \Gamma \\
(\sigma \to \tau, \Gamma) &\longmapsto (\ \lambda\ x\ .\ (\tau; \Gamma, x : \sigma)\ )
\end{aligned}
$$

The second rule can be replaced by more effective one.

$$
(\tau_1 \to \ldots \to \tau_n \to \alpha, \Gamma) \longmapsto (\ \lambda\, x_1\ \ldots\ x_n\ .\ (\alpha; \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n)\ ) \quad \textbf{if } n > 0
$$

This rule packs consecutive uses of the second rule into one use. This is valid since the use of the second rule is deterministic; it is used if and only if the non-terminal's type is not atomic.

5. OUR APPROACH

6. EXPERIMENTS

7. CONCLUSIONS

8. ACKNOWLEDGMENTS

This section is optional; it is a location for you to acknowledge grants, funding, editing assistance and what have you. In the present case, for example, the authors would like to thank Gerald Murray of ACM for his help in codifying this *Author's Guide* and the **.cls** and **.tex** files that it describes.

9. REFERENCES

[1] H. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus With Types*. Cambridge University Press, 2010.

[2] F. Binard and A. Felty. Genetic programming with polymorphic types and higher-order functions. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1187–1194. ACM, 2008.

[3] F. Briggs and M. O'Neill. Functional genetic programming and exhaustive program search with combinator expressions. *International Journal of Knowledge-Based and Intelligent Engineering Systems*, 12(1):47–68, 2008.

[4] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA, 1992.

[5] J. R. Koza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

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

[7] R. Poli, W. B. Langdon, and N. F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.

[8] T. Yu. Hierarchical processing for evolving recursive and modular programs using higher-order functions and lambda abstraction. *Genetic Programming and Evolvable Machines*, 2(4):345–380, 2001.