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. 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 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

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

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

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

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].

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.*

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

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].

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 $(\lambda x_1\,.\,(\lambda x_2\,.\,\ldots.\,(\lambda x_n\,.\,M)\ldots))$.

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

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

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

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.*

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

Unfinished leaf $\langle \tau; \Gamma \rangle$ stands for yet not specified λ-term of the type τ build from symbols of Γ.

4. OUR APPROACH

5. EXPERIMENTS

6. CONCLUSIONS

[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.

7. 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.

8. 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.