Utilization of Reductions and Abstraction Elimination in Typed Genetic Programming

Tomáš Křen*†
tomkren@gmail.com

*Faculty of Mathematics and Physics
Charles University in Prague
Malostranské náměstí 25, 11000,
Prague, Czech Republic

Roman Neruda†
roman@cs.cas.cz

†Institute of Computer Science
Academy of Sciences of the Czech Republic
Pod Vodárenskou věží 2, 18207,
Prague, Czech Republic

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 the use of reductions (beta and eta) and a way 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 propose two novel crossover operations based on abstraction elimination capable of handling general form of typed lambda term while being a straight generalization of the standard crossover operation. We compare their performances using the even parity benchmark problem.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program synthesis*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Lambda calculus and related systems*

General Terms

Experimentation, Languages, Theory

Keywords

genetic programming; lambda calculus

1. INTRODUCTION

Genetic programming (GP) represents an efficient method for automatic generating of programs by means of evolutionary techniques [5, 6]. Early attempts to enhance the GP approach with the concept of types include the seminal work [9] where the ideas from Ada programming language were used to define a so-called strongly typed GP. Usage of types naturally opens door to enriching S-expressions, the traditional GP representation of individuals, with concepts from lambda calculus, which is a simple yet powerful functional, mathematical and programming language extensively used in type theory [11].

One of the motivations for using lambda calculus is that it is backed by a considerable theoretical background. From this background we can utilize several useful concepts such as reductions, normalization or abstraction elimination, and use them to enhance the standard GP algorithm. We use normalization to reduce the size of the search space of program trees and the abstraction elimination to prevent the difficulties caused by the presence of variables in program trees.

In our system we use a simply typed lambda calculus. Since abstraction elimination and reductions do not involve many type related issues it seems to us as a sufficient choice of a type system. Techniques described here would be analogous for a more sophisticated type system and can be directly applied to such systems as the Hindley–Milner type system.

In this paper we present an approach to typed GP over lambda calculus capable of operating with the full array of notions given by a simply typed lambda calculus while being a direct generalization of standard GP [5]. We discuss the advantages of generating lambda terms in their long normal form and present two novel crossover operations for simply typed lambda terms. Compared to the approach to typed GP over lambda calculus proposed by Yu [11], it uses more general term representation which is not restricting the use of outer local variables in the body of an anonymous function. Compared to the purely combinator approach by Briggs and O'Neill [4], it has the advantage of reducing the size of the search space in the term generation phase.

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. The body of the paper is presented in sections 4 and 5; the former discusses the term generating procedure and its relations with reductions, the latter describes several approaches to lambda term crossover, and how they solve the variable problem. Section 6 examines those ideas on the even parity benchmark problem, while the paper is concluded by section 7.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2662-9/14/07 ...$15.00.
http://dx.doi.org/10.1145/2576768.2598361.

2. RELATED WORK

Yu presents a GP system utilizing polymorphic higher-order functions and lambda abstractions [11]. An important point of interest in the work is the 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 a significant difference from our approach since we permit all well-typed normalized λ-terms. This difference also causes the need for different crossover operators.

Briggs and O'Neill present a technique utilizing typed GP with combinators [4]. The difference between the approach presented in their work and our approach is that they generate terms in a straightforward way directly from the library of combinators, without any use of lambda abstractions. They use the Hindley–Milner type system. They also present an interesting concept of *Generalized genetic operator* based on term generation. The combinator based genetic algorithm has the advantage of avoiding the need of dealing with variables.

Binard and Felty use an even stronger type system, the (*System F*) [3]. But with the increasing power of the type system also comes an increasing difficulty of term generation. For this reason, the 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 a straightforward generalization of the standard GP[5].

In contrast to the above mentioned works, our approach uses a very simple type system (the 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[2].

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.*

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

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*[1].

[1]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.

To ensure better readability, $M_1\ M_2\ M_3\ \ldots\ M_n$ will be an abbreviation for $(\ldots((M_1\ M_2)\ M_3)\ \ldots\ M_n)$. And $\lambda\, x_1 x_2 \ldots x_n\,.\, M$ for $(\lambda\, x_1\,.\,(\lambda\, x_2\,.\,\ldots(\lambda\, x_n\,.\, M)\ldots))$.

3.1 Reductions

In order to perform computation, there must be some mechanism for term evaluation. In λ-calculus there is a β-reduction procedure 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."

Similarly as for β-reduction we can define η-reduction except that now it is defined as follows.

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

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

Relation[2] $\rightarrow_{\beta\eta} = \rightarrow_\beta \cup \rightarrow_\eta$. Similarly as for $\twoheadrightarrow_\beta$ and $=_\beta$ we can define relations $\twoheadrightarrow_\eta$, $=_\eta$,$\twoheadrightarrow_{\beta\eta}$ and $=_{\beta\eta}$.

The η^{-1}-reduction (also called η-expansion) is the reduction converse to η-reduction. It is defined as follows.

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

3.2 Normal forms

Definition 2.

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. Normalization of the term M is the process of finding normal form of M. Similarly we can define *η-nf* and *$\beta\eta$-nf*.

[2]Relation $R = \{\ (a, b) \mid a\ R\ b\ \}$.

3.3 Types

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 3. *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 4.

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

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

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

3.4 Long normal form

Definition 6. *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 [2], *lnf* has following nice properties.

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

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

3.5 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 **S**, **K** and **I**.

Those combinators are defined as:

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

Let us describe transformation AE performing this process.

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

As is stated in [10], 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 abstractions.

The algorithm presented here is a simple version of this process. More optimized version (used in our *hybrid* crossover described below), by means of the size of resulting term and its time performance is presented in [10].

4. GENERATING AND REDUCTIONS

This section is focused on the individual generating method producing terms in their long normal form, which can be understood as a straight generalization of S-expressions into lambda calculus. We discuss the relation between terms in *lnf* with beta and eta reductions, the advantages of such representation and we also show how to easily transform such terms into short *$\beta\eta$-nf*.

4.1 Grammar producing λ-terms in *lnf*

In [2], the following term generating 2-level grammar[3] for generating all possible terms in their *long normal form* is described.

If we want to generate a term M satisfying $\Gamma \vdash M\colon \tau$, then the initial non-terminal is (τ, Γ).

For every $\alpha \in A$ and context Γ and f such that $(f : \rho_1 \to \ldots \to \rho_m \to \alpha\ ) \in \Gamma$ we have a rule of the first form:

$$(\alpha, \Gamma) \longmapsto (\ f\ (\rho_1, \Gamma)\ \ldots\ (\rho_m, \Gamma)\ )$$

[3] 2-level grammar (unlike a standard grammar) can have infinitely many non-terminals and its rule is better described as *rule schema*. The original grammar described in [2] has $(\sigma \to \tau, \Gamma) \longmapsto (\ \lambda\ x\ .\ (\tau; \Gamma, x : \sigma)\ )$ as second rule, but our grammar is equivalent – it packs consecutive uses of this rule into our rule.

In other words, one possible way to construct a term of the atomic type α from the building symbols Γ is to take a function symbol f from Γ with the return type α (or if $m = 0$ a constant of the type α), put it into the root of the term and to generate a subtrees of appropriate types for each of f's arguments from the building symbols Γ.

For every atomic type α, context Γ and types $\tau_1, \ldots, \tau_n$ (where $n > 0$) we have a rule of the second form:

$$(\tau_1 \to \ldots \to \tau_n \to \alpha, \Gamma) \longmapsto (\ \lambda\ x_1\ \ldots\ x_n\ .\ (\alpha; \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n)\ )$$

In other words, the way to construct a term of the function type $\tau_1 \to \ldots \to \tau_n \to \alpha$ from the building symbols Γ is to create an anonymous function with fresh variables $x_1, \ldots, x_n$ in its head whose body is constructed as a term that has type α and is constructed using symbols from Γ enriched with those new variables $x_1, \ldots, x_n$, which are associated with appropriate types.

Long normal form is a generalization of S-expressions into lambda calculus: Let Γ be a context satisfying the closure requirement, then only the first rule will be applicable. (α, Γ) rewrites to $(\ f\ (\alpha, \Gamma)\ \ldots\ (\alpha, \Gamma)\ )$. One can see that f stands for the root node and that each (α, Γ) will produce a direct subtree.

For a context containing at least one higher order function (or if the desired type of generated terms is a function type) we get special kind of root node containing lambda head with one direct subtree standing for body of the lambda function. Local variables defined in the lambda head may also appear in this body.

Discussion of what is the best strategy for browsing the search space given by above described grammar in order to generate *lnf* terms is beyond the scope of this paper. Standard *ramped half-and-half* strategy can be used for this purpose, or some other, which we discuss in [7]. In our experiments we use the *geometric* strategy proposed in [7].

4.2 Benefits of generating λ-terms in *lnf*

By generating λ-terms in *lnf* we avoid generating λ-terms M,N such that $M \neq N$, but $M =_{\beta\eta} N$. This means that M,N correspond to two programs with different source codes, but both perform the same computation.

Every λ-term M such that $\Gamma \vdash M : \sigma$ has its unique *lnf* L, for which $L =_{\beta\eta} M$. Therefore, the computation performed by λ-term M is not omitted, because it is the same computation as the computation performed by λ-term L.

Generating λ-terms in *lnf* is even better than generating λ-terms in β-*nf*. Since *lnf* is the same thing as $\beta\eta^{-1}$-*nf*, every λ-term in *lnf* is also in β-*nf*.

This comes straight from the definition of $\beta\eta^{-1}$-*nf*, but one can also see it by observing the method for generating terms in β-*nf*. As shown in [2], this method is obtained by the following 2-level grammar.

$$\begin{aligned} (\pi, \Gamma) &\longmapsto (\ f\ (\rho_1, \Gamma)\ \ldots\ (\rho_m, \Gamma)\ ) \\ (\sigma \to \tau, \Gamma) &\longmapsto (\ \lambda\ x\ .\ (\tau; \Gamma, x : \sigma)\ ) \\ &\quad \textit{where} \quad (f : \rho_1 \to \ldots \to \rho_m \to \pi) \in \Gamma \end{aligned}$$

This grammar differs from the grammar for *lnf* only in the first rule[4]. Specifically, it differs in that f's type is no longer needed to be fully expanded ($\pi \in \mathbb{T}$ instead of $\alpha \in A$). This makes the grammar less deterministic, resulting in a bigger search space. The new rule is generalization of the old one, thus all terms in *lnf* will be generated, along with many new terms in β-*nf* that are not in *lnf*.

By generating λ-terms in *lnf* we avoid generating λ-terms M,N such that $M \neq N$ and $M =_\eta N$; but generating in β-*nf* does not have such a property.

The disadvantage of the *lnf*, as the name suggests, is that it is long. Terms in *lnf* are said to be *fully η-expanded* [2]. A relevant property of η-reduction is that it always shortens the term that is being reduced by it. And conversely, η-expansion prolongs.

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

Now we show that for every λ-term M, every sequence of η-reductions is finite and it leads to a unique η-*nf* N.

1. Every application of η-reduction shortens the term. Since every term has finite size, this process must end at some point. Thus every λ-term has η-*nf*.
2. Since η-reduction is *Church–Rosser*, η-*nf* is unique (see [1]).

So we can take every generated λ-term M in *lnf* and transform it to shorter term in η-*nf*. The question is whether it remains in β-*nf*, thus being in $\beta\eta$-*nf*. The answer is yes; it can be proven by showing that no new β-redex is created by η-reduction.

Proposition 3. *Let P be in β-nf and $P \to_\eta Q$. Then Q is in β-nf.*

Proof. For better clarity let us show the η-reduction and β-redex using @-trees (a λ-term representation derived from the definition of λ-terms where @ stands for *function application*).

η-reduction: $\lambda x \to_\eta M$ — λx → @ → M, x

β-redex: @ → λx, N; λx → M

Let us assume that $P \to_\eta Q$ creates a new β-redex B in Q. Since η-reduction only destroys and never creates *function applications* (i.e. @), the root @ of B must be present in P. But since P contains no β-redex, the left subterm L of this root @ is not λ-abstraction. Only possible way for L to be changed by $\to_\eta$ into a λ-abstraction is that L is the reduced subterm (so that L is changed for its subterm). But that is in contradiction with P not containing any β-redex, because it would cause L be a λ-abstraction. □

Notable property of *lnf* and $\beta\eta$-*nf* is that there is *bijection* (i.e. one-to-one correspondence) of the set of simply typed λ-terms in *lnf* and the set of simply typed λ-terms in $\beta\eta$-*nf*.

Proposition 4. *Reduction to η-nf is bijection between the set of simply typed λ-terms in lnf and the set of simply typed λ-terms in $\beta\eta$-nf.*

Proof. Since reduction to η-*nf* always leads to an unique term, it is a function. In previous proposition is shown that η-reduction of *lnf* leads to a term in $\beta\eta$-*nf*.

In order to show that a function is bijection it is sufficient to show that it is both *injection* and *surjection*.

[4]We use a different second rule for *lnf* grammar, but as is mentioned in footnote above, it also works with this second rule.

Suppose it is not injection.

So there must be M_1, M_2 in *lnf* such that $M_1 \neq M_2$ and N in $\beta\eta$-*nf* such that $M_1 \twoheadrightarrow_\eta N$, $M_2 \twoheadrightarrow_\eta N$. Therefore $M_1 =_\eta M_2$, so $M_1 =_{\beta\eta^{-1}} M_2$. This contradicts with M_1, M_2 being distinct *lnf*s.

Every M in β-*nf* has a *lnf* N such that $N \twoheadrightarrow_\eta M$ (proposition from 3.4). Term M in $\beta\eta$-*nf* is in β-*nf*, thus it has desired *lnf* N which reduces to it.

Therefore it is surjection. □

Suppose we have a systematic method (i.e. gradually generating all terms, but no term is generated twice) for generating terms in *lnf*, we may transform it to a systematic method for generating terms in $\beta\eta$-*nf* by simply reducing each generated term to its η-*nf*.

5. CROSSOVER OPERATOR

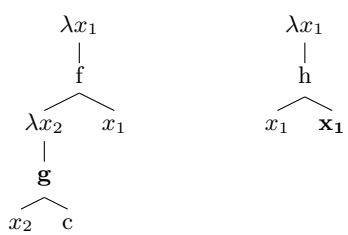
The design goal behind our approach to the crossover operation is to try to generalize the standard tree swapping crossover.

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.

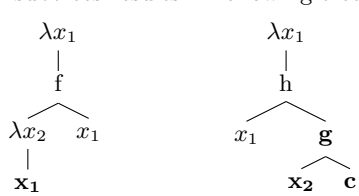
As in standard GP, our crossover will be performed by swapping two subtress of the same type.

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

The 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 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.

Generally speaking, the variable problem is caused by the fact that the local variable is not defined in the new place, or the variable is defined in the new place, but has a different type. Let us list some possible approaches to solving this problem.

One option is to choose a less general representation of lambda terms. Such approach is successfully used in [11] where the terminal set for constructing lambda abstraction subtrees is limited to use only constants and variables of that particular lambda abstraction, and an appropriate variable naming convention is used in order to prevent troubles with variables. We choose not to follow such approach because we do not want to lose the opportunity to represent every possible well-typed term in its normal form.

Another option is to overlook the variable problems and to correct the defects when they occur. Such correction can be performed by renaming the problematic variable to some other variable defined in the new local scope of swapped subtree. Or, if there is no suitable candidate, to replace its occurrence with a newly generated term of the required type. This approach seems to us as a very unwieldy one – there are clearly problems for which such collisions would occur very often and when there is no suitable rename candidate the generation of new term brings the unnecessary element of a mutation. Such solution seems to be useful only as a last resort.

There is an elegant way to overcome the problem with variables by getting rid of them. This is possible thanks to the abstraction elimination algorithm described above. It turns a λ-term into a λ-term that contains no lambda abstractions and no variables, instead, it contains additional function symbols standing for polymorphic combinators (i.e. **S**, **K** and **I** in the simple case of the algorithm). When there are no variables we can freely swap the subtrees of the same type.

We also use a little trick; if the type of the individual is a function type it is not necessary to abstract out the input variables (and to them associated lambda heads) of the function since all the individuals share them. (This is illustrated by the figure 2 bellow, where the root lambda head λx is still on the trees and so is the variable x in the leafs.)

We identified two approaches that utilize abstraction elimination.

5.1 Hybrid crossover

The first one converts all the generated terms at the end of the population initialization phase. We call this approach to lambda term crossover *hybrid*, because it can be understood as hybrid of the lambda calculus representation and the purely combinator representation used in [4]. The advantage of generating terms as *lnf* lambda terms instead of generating them directly as combinator terms is that it reduces the search space, because there is more than one way to represent a *lnf* lambda term as a combinator term (this can be seen considering that *lnf* term $(\lambda\, x\, .\, x)$ is equivalent to both **I** and **SKK**), but every combinator term has only one unique *lnf* form. Unfortunately, one disadvantage is also important; the abstraction elimination can cause up to a quadratic increase in the term size. It is reasonable to suspect that combinator terms generated directly may be smaller than those mechanically translated by the abstraction elimination. The act of producing a small normalized term and translating it immediately to a bulky combinator form can be seen as a wasteful behavior.

5.2 Unpacking crossover

Thus, we propose the following crossover operator in order to be able to analyze the above mentioned issues, which we call the *unpacking* crossover. Unlike the *hybrid* it operates over λ-terms and produces λ-term offspring in $\beta\eta$-*nf*. When two parent λ-terms are about to be crossed, they are both converted to combinator terms using the elimination abstraction algorithm. After the trees are swapped, the offspring terms are again normalized to the $\beta\eta$-*nf* λ-terms

containing no temporary combinators added during the abstraction elimination. This normalization is achieved by replacement of all occurrences of temporary combinators by their respective definitions (e.g. **K** is replaced by $(\lambda\, xy\,.\,x)$) followed by beta and eta normalization. We believe that the property of increasing the term size is advantageous for the *unpacking* crossover. This belief is based on the fact that larger terms have more crossover points and on the observation that higher number of crossover points is beneficial for crossover operation, this observation will be discussed in more detail in the subsequent subsection. The difference between the *unpacking* and *hybrid* crossover is that in the case of the *unpacking* this increase is only in the temporary stage, which is reversed by the normalization. The most notable disadvantage of this crossover approach is that it is more time consuming than the simple one.

5.3 Typed subtree swapping in greater detail

The first thing to do in a standard subtree swapping crossover is to select random node in the first parent.

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

The standard subtree swapping crossover 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, since 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 the type of that node must correspond to 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 than sexpr-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.

The following example should clarify this:

In the @-tree, **f** is leaf, thus it is a subtree, whereas in sexpr-tree it is an internal node and thus not a subtree. Those favorable properties of @-tree are also reported in [12].

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 to standard GP, 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 exceed this limit, then both are replaced by parents.

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

6. EXPERIMENTS

In this section we demonstrate our approach on the well-known even parity problem. More extensive experimental evaluation will be a subject of future work. We made two experiments, in the first we have used the *hybrid* crossover, while in the second we have used the *unpacking* crossover.

Each experiment consisted of 50 independent runs of GP algorithm. Each run had a limit of 51 generations, and the population size was 500 individuals. In the experiments we analyze the fitness of the best individual, the average size of an individual, the ability of the system to produce a correct solution, and computational cost estimates throughout generations. For the last two metrics we use the popular measurement methods within the GP field — the *performance curves* described in [5] — $P(M, i)$ (cumulative probability of success) and $I(M, i, z)$ (the total number of individuals that must be processed to yield a correct solution with probability $z = 99\%$). Difference of those two crossover methods is statistically analyzed by the Welch t-test by comparing the fitness values of the best individuals of the run. Graphs showing mean values of the fitness of the best individual and the average size of an individual throughout generations contain error bars representing the standard error of the mean (SEM).

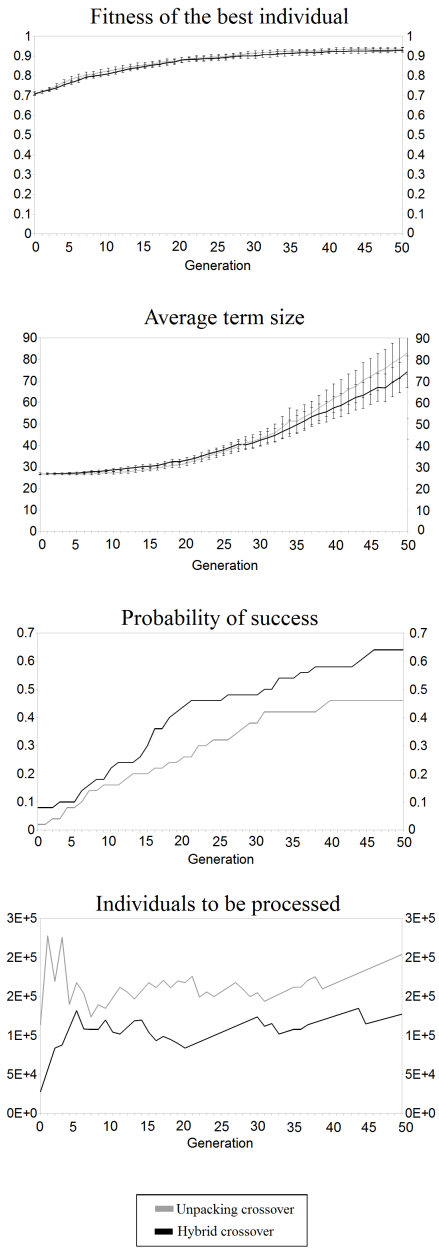
The even parity function is a boolean function taking as inputs N boolean values and returning *True* if an even number of inputs are *True*. For odd number it returns *False*. This problem has been used by many researchers as a benchmark for GP. We compare our results with that presented in [11, 4]. We use very similar set of *building symbols* as in [11].

$$
\begin{aligned}
\tau &= [Bool] \to Bool \\
\Gamma = \{ and &: Bool \to Bool \to Bool, \\
or &: Bool \to Bool \to Bool, \\
nand &: Bool \to Bool \to Bool, \\
nor &: Bool \to Bool \to Bool, \\
foldr &: (Bool \to Bool \to Bool) \\
&\quad \to Bool \to [Bool] \to Bool, \\
head' &: [Bool] \to Bool, \\
tail' &: [Bool] \to [Bool] \}
\end{aligned}
$$

The type $[Bool]$ stands for *list of Bools* and for purpose of this problem is considered atomic. Unlike in [11], we use specific instance of polymorphic function `foldr`. And modifications of functions `head` (returning the first element of the list) and `tail` (returning the list without the first element) are used; making them total by returning default value *False* and `[]`, respectively. We use the same fitness function as in [11]. The fitness function examines the individual by giving it all possible boolean lists of length 2 and 3.

Figure 1 shows results of this experiments. Table 1 summarizes statistical analysis of the fitness values of the best individuals of the run (p-*value* $= 0.7323$). According to

Figure 1: Graphs for Even parity problem.



	Unpacking	Hybrid
Mean	0.935000	0.930000
SD	0.066768	0.078490
SEM	0.009442	0.011100
N	50	50
t-value	0.3431	$\alpha = 0.05$
p-value	0.7323	**Not statistically significant**

Table 1: Statistical analysis.

GP approach	$I(M, i, z)$
PolyGP	14,000
Our approach (hybrid)	**28,000**
GP with Combinators	58,616
GP with Iteration	60,000
Our approach (unpacking)	**114,000**
Generic GP	220,000
OOGP	680,000
GP with ADFs	1,440,000

Table 2: Results comparison.

this analysis both crossovers performed about the same; by conventional criteria, this difference is considered to be not statistically significant.

The *unpacking* crossover scored 23/50 (46%) success rate. Minimal $I(M, i, z)$ was in generation 0 with 114,000 individuals to be processed. The average individual size for generation 50 was 83.1.

The *hybrid* crossover scored 32/50 (64%) success rate. Minimal $I(M, i, z)$ was in generation 0 with 28,000 individuals to be processed. The average individual size for generation 50 was 74.3.

Comparison with other results taken from literature is summarized in table 2. Concerning the time complexity, both experiments were run under same conditions on a fairly standard machine with 4 core 3.3 GHz processor and 4 GB RAM. The first experiment took 388 minutes while the second one took 33 minutes.

We have also analyzed ratio of failed crossovers due to exceeding the maximal permissible depth. Only around 0.5% of offspring created by the unpacking crossover fail due to exceeding the limit and 0.2% for the hybrid crossover. Thus we conclude that the case of crossover failure does not have a significant effect on the performance of the operators.

In order to illustrate how the individuals actually look like and how the crossover works, figure 2 depicts creation of the correct solution by the hybrid crossover in one particular run of the system.

The found correct solution can be rewritten as:

$$\begin{aligned}
&\lambda\, x\, .\, foldr\ (\mathbf{S}(\mathbf{S}(\mathbf{K}\ \mathbf{S})(\mathbf{S}(\mathbf{K}(\mathbf{S}(\mathbf{K}\ nor)))and))nor) \\
&\qquad (nor\ (head'\ x)\ (head'\ x))\ \ (tail'\ x) \\
&=_{\beta\eta} \\
&\lambda\, x\, .\, foldr\ (\lambda\, y\ z\, .\, nor\ (and\ y\ z)\ (nor\ y\ z)) \\
&\qquad (nor\ (head'\ x)\ (head'\ x))\ \ (tail'\ x)
\end{aligned}$$

Which is equivalent to:

$$\lambda\, x\, .\, foldr\ xor\ (not\ (head'\ x))\ \ (tail'\ x)$$

7. CONCLUSIONS

In this paper we presented a generalization of standard GP based on simply typed lambda calculus trees. We discussed the advantages of generating lambda terms in their long normal form and presented two novel crossover operations for this system.

The first one — the *hybrid* crossover — transforms all generated individuals into combinator terms at the end of generation phase. The second crossover approach — the *unpacking* crossover — uses the lambda term representation during the whole run, except for the temporary phases during the crossover. We have compared their relative performance on even parity problem; the first one is performing the same or better in all observed metrics. Since the use of

Figure 2: Hybrid crossover example.

the *unpacking* crossover brings a significant computational overhead while its performance is the same or worse than the performance of the *hybrid* crossover we conclude that the *hybrid* crossover is better suited for the job.

According to comparison of "individuals to be processed" metric our GP approach is performing better than the GP system presented in [4] and worse than the one presented in [11]. By observation of typical solution which is often some modification of `foldr xor True inputList` one sees that an important task for GP here is to create `xor` function. This is the advantage for the more specialized term representation used in [11], which restricts the use of outer variables.

It is also fair to say, that this popular metric (in which the comparison data are available) is somehow problematic, since it is prone to have large variation across various instances of the same experiment [8]. As is also visible from our results; both our approaches scored best in generation 0 thus the result shown in table 2 depended only on the generation method which those two approaches shared, yet the individuals to be processed estimates are quite different.

In the future work, we would like to support our theoretical results concerning relations of generating procedure and reductions described in this paper with substantially extended experimental results.

In this paper we touched varied aspects of the two lambda calculus concepts – reductions and abstraction elimination – in the hope that it will shed some more light on the promising intersection of lambda calculus and genetic programming.

8. ACKNOWLEDGMENTS

Tomáš Křen has been partially supported by the project GA ČR P202/10/1333 and by the SVV project number 260 104. Roman Neruda has been partially supported by The Ministry of Education of the Czech Republic project COST LD 13002.

9. REFERENCES

[1] H. Barendregt. *The Lambda Calculus: its Syntax and Semantics, revised ed.* North-Holland, 1984.

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

[3] 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.

[4] 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.

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

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

[7] T. Křen and R. Neruda. Generating lambda term individuals in typed genetic programming using forgetful A*. In *Proceedings of the Congress on Evolutionary Computation.* IEEE Computer Press, 2014. (in print).

[8] S. Luke and L. Panait. Is the perfect the enemy of the good? In *GECCO*, pages 820–828, 2002.

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

[10] S. L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[11] 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.

[12] T. Yu and C. Clack. PolyGP: A polymorphic genetic programming system in haskell. *Genetic Programming*, 98, 1998.