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].

TODO....

ve shrnutí paperu říct, že jeho tělo je tvořeno dvěma částma, první se zabejvá generováním a jeho vztahem k redukcím, druhá řeší křížení s velkym důrazem na vyřešení problémů s lokálníma proměnejma.

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 **??**. Section 6 presents results of our method on three well-known tasks, and the paper is concluded by section 7.

1.1 Randombits

- Možná někam poznamenat že přínos toho článku je taky v tom že zpřístupňuje pojmy z teorie typovaného lc komunitě typovaného GP..

2. RELATED WORK

TODO zatim jen zkopčený z minula a dodaná diskuze zatim česky porovnávající nás a kombinatoři , opravit podle novýho kontextu

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

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

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

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.

POZNAMKY PRO DISKUSI POROVNAVAJICI NÁS A BRIGGSE:

The individual representation used in standard GP is S-expression. Basically there are two general approaches of generalizing S-expressions into trees of better expressive power suitable for typed GP with higher order functions: We can enrich the method with the polymorphic combinators [3] or we can enrich the method with the lambda abstractions and local variables [8].

Pokud chceme dělat GP nad stromama větší vyjadřovací síly než maj klasický S-výrazy, tak máme v záse dvě přirozený možnosti pro obecný řešení: bud' 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.

Zdá se že použití kombinátorů má řadu výhod (vymenovat nějaký co jmenuje briggs), ale konzistentní prozkoumání obecnýho řešení lambdovýho charakteru určitě stojí taky za zkoužku, což podporují nasledující argumenty:

(a) motivace pro práci s lambda termama je, že za nima košatá teorie, která například popisuje redukce lambda termů (ty zajišt'ujou jak zmenšení samotných stromu, tak to, že prohledávací prostor se zmenší, díky tomu, že se různé stromy redukují na ten samej). (ALT znění: pohovořit o tom, že normální formy zmenšujou prohledávaný prostor, protže stačí brát v potaz třídy ekvivalence a ne zamotný termy - reprezentanti těchto tříd ekvivalence jsou pak právě normální formy.) (ALT2: Dost podstatný ospravedlnění lambda reprezentace je fakt, že redukce (beta, eta) šlapou právě jen při tom, když máme lambdy.)

(b) úspornost - abstrakce eliminací produkuje termy až kvadraticky větší než je původní term. Dá se očekávat, že i při přímém generování je použití kombinátorů míň úsporný (proměná se může bez velký námahy objevit v hluboko vnořeném podtermu velmi jednoduše, zatímco pomocí kombinátorů se její hodnota musí do takto vnořeného podtermu složitě dopravit).

(c) použití kombů vyžaduje silnější typovej sytém - kombinátory musej bejt polymorfní a přidáváme je do stavební sady, lambdy v pohodě běžej na simply typed lc.

(d) použití proměnných je typické pro lidi, proto by se mohlo sát že i programy generovaný systémem kterej s proměnejma počítá budou o něco "lidštější"

Since abstr-elim a redukce neinvolvujou moc typově rilejtlý problémy, je ok celou věc demostrovat na jednoduchém typovém systému jakám je simply typed lc. Pro more soffisticated type systems would be situation similar.

Hlavním problémem lambda reprezentac proti kombinátorový je jak se vypořádat s problémy zpusobenými použitím localních proměných (aka jak křížit).

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

TODO: ještě určitě projít a vylepšit; někde zestručnit někde rozšířit

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.*

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

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

3.1 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* $\to_\beta$ of those two terms:

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

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

$$P \to_\beta Q \Rightarrow (R\ P) \to_\beta (R\ Q)$$
$$P \to_\beta Q \Rightarrow (P\ R) \to_\beta (Q\ R)$$
$$P \to_\beta Q \Rightarrow \lambda\, x\, .\, P \to_\beta \lambda\, x\, .\, Q$$

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) $M \twoheadrightarrow_\beta M$

[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.

(b) $M \to_\beta N \Rightarrow M \twoheadrightarrow_\beta N$

(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. $M \to_\beta N$ — "M β-reduces to N in one step."
3. $M =_\beta N$ — "M is β-convertible to N."

3.2 η-reduction

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

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

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

Relation $\to_{\beta\eta} = \to_\beta \cup \to_\eta$. (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 $=_{\beta\eta}$.

3.3 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:

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

3.4 Normal forms

asi řít tady v ty subsekci co že to je normalizace termu jednou větou

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*.

3.5 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.*

$$\alpha \in A \Rightarrow \alpha \in \mathbb{T}$$
$$\sigma, \tau \in \mathbb{T} \Rightarrow (\sigma \to \tau) \in \mathbb{T}$$

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.*

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

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 $\Gamma \vdash M : \tau$.

3.6 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 $L \twoheadrightarrow_\eta B$.*

3.7 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:

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

Let us describe transformation performing this process.

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

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 [7].

As is stated in [7], 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.

tady je to v původnim kontextu, odebrat slova co se tam nehodí

4. TERM 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 [1] is shown term generating grammar with following rules.

předělat ten zápis, víc to okecat, říct že to je jakoby gramatika, formálně 2-level-gramatika do footnoty, pač má nekonečně neterminálu i terminálů. že neterminaly jsou dvojice (typ, kontext) a že terminaly jsou symboly kterejma se zapisujou lambda termy

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

The second rule can be replaced by more effective one.
$(\tau_1 \to \ldots \to \tau_n \to \alpha, \Gamma) \longmapsto$
$(\; \lambda\; x_1 \; \ldots \; x_n \;.\; (\alpha; \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n) \;) \quad$ **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.

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 (ρ_i, Γ) will produce a direct subtree.

For context containing at least one higher order function we get special kind of root node containing lambda head with one direct subtree standing for body of the lambda function, v tomto tele se navíc mohou objevit proměnné definované v této hlavě (asi doplnit obrázkama).

Diskuze toho jakou přesnou strategií je nejvhodnější generovat termy v lnf je beyond the scope of this paper, mužeme například použít klasickou raped half-and half strategii. Footnotenout krat'oučkou zmínku o gemetrický.

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 $M =_{\beta\eta} N$. In other words, we avoid generating two programs with different source codes, but performing 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 from observing method for generating terms in β-*nf*. As is shown in [1], this method is obtained simply by replacing the first grammar rule by slightly modified rule, resulting in the following grammar.

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

The difference lies 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 by generating in β-*nf* we do not avoid it.

The disadvantage of the *lnf*, as the name suggests, is that it is long. Terms in *lnf* are said to be *fully η-expanded* [1]. Relevant property of η-reduction is that it always shortens the term that is being reduced be 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 leads to unique η-*nf* N.

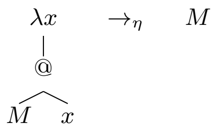
1. Every application of η-reduction shortens the term. Since every term has finite size, this process must end at some point. Thus every λ-term has η-*nf*.

2. Since η-reduction is *Church–Rosser*, η-*nf* is unique (see [**?**]).

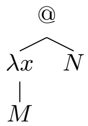
So we can take every generated λ-term M in *lnf* and transform it to shorter term in η-*nf*. The question is whether it remains in β-*nf*, thus being in $\beta\eta$-*nf*. The answer is yes; it can be proven by showing that no new β-redex is created by η-reduction.

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.
η-reduction:



And β-redex:



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. $\square$

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*.

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 3.6). Term M in *$\beta\eta$-nf* is in *β-nf*, thus it has desired *lnf* N which reduces to it.
Therefore it is surjection. $\square$

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

5. CROSSOVER OPERATOR

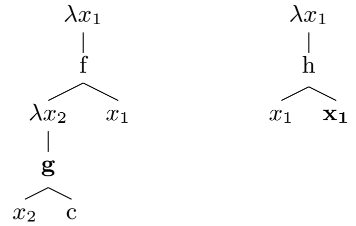
The design goal behind our approach to the crossover operation is to try to generalize the standard tree swapping crossover.

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.

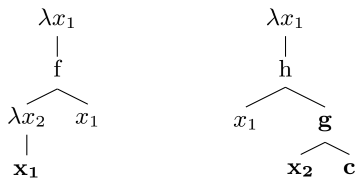
As in standard GP our crossover will be performed by swapping two subtrees. But now with constraint that both subtrees have the same type.

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 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 that the local variable is not defined in the new place or the variable is defined in the new place, but has some another 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 [8] 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 another variable defined in that place or, if there is no suitable candidate, to replace its occurrence with a newly generated term of 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 identified two approaches that utilize abstraction elimination:

The first one converts all the generated terms at the end of the population initialization phase. We call this approach to lambda term crossover the *hybrid*, because it can be understood as hybrid of the lambda calculus representation and the purely combinator representation used in [3]. 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 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 the 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 are probably 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 seems as a wasteful behavior.

Thus we propose following crossover operator, 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. (TODO: opravdu jí tam diskutovat) The difference between the *unpacking* and *hybrid* crossover is that in the case of the *unpacking* this increase is only 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 ones. TODO: Až bude známo jak moc to žere čas, tak to sem připsat. a taky někde zmínit asi výslovně, že pro closed problémy je to stejný jako kozovská pač vnich žadný abstrakce nejsou. Taky nějak vhodně zmínit analogii s přírodou.

5.1 Typed subtree swapping in greater detail

POZNAMKY:

Tuhle subsekci ještě pořádně projít, zatim je to tu skopčený z diplomky.

Na tuhle subsekci se odkazuje předchozí text o unpacking s tim že je tu rozebranej vliv počtu křižicích bodů na kvalitu křížení, takže to tu rozebrat, tzn. musi se odiskutovat rozdíl mezi @-tree a S-expresion.

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.

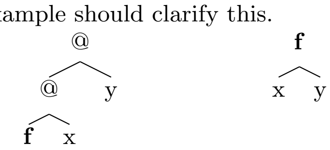
After selecting a node in the first parent we select node in the second parent such that type of that node must by the same as the type of the first node. Again, this may eliminate the "90-10" step of first deciding whether the selected node will be internal node or leaf node.

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 $D_{created} = 17 \times 3$.

6. EXPERIMENTS

TODO!!

7. CONCLUSIONS

TODO

mimojiné zmínit, že v budouznu bysme chtěli porovnat kombinátory s tou naší nějak extenzivně, ikdyž možná to tam vubec nepsat (což by bylo hezký už ted ale nestíhá se to..)

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] 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.

[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.