Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Tomáš Křen

Typed Functional Genetic Programming

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	RNDr. Petr Pudlák, Ph.D.
Study programme:	Theoretical Computer Science
Specialization:	Non-Procedural Programming and Artificial Intelligence

Prague 2013

Dedication.

Název práce: Název práce

Autor: Jméno a příjmení autora

Katedra: Název katedry či ústavu, kde byla práce oficiálně zadána

Vedoucí diplomové práce: Jméno a příjmení s tituly, pracoviště

Abstrakt:

Klíčová slova:

Title:

Author: Jméno a příjmení autora

Department: Název katedry či ústavu, kde byla práce oficiálně zadána

Supervisor: Jméno a příjmení s tituly, pracoviště

Abstract:

Keywords:

Contents

Introduction

1. Definitions

Let us first say some basic definitions.

1.1 Genetic Programming

Genetic programming (GP) is a technique inspired by biological evolution that for a given problem tries to find computer programs able to solve that problem. GP was developed by John Koza [1] in 1992.

A problem to be solved is given to GP in a form of *fitness function.* Fitness function is a function which takes computer program as its input and returns numerical value called *fitness* as output. The bigger fitness of a computer program is, the better solution of a problem.

GP maintains a collection of computer programs called *population.* A member of population is called *individual.* By running GP algorithm evolution of those individuals is performed.

Individuals are computer program *expressions* kept as *syntactic trees.* Basically those trees are rooted trees with a function symbol in each internal node and with constant symbol or variable symbol in each leaf node. Number of child nodes for each internal node corresponds to the number of arguments of a function whose symbol is in that node.

Another crucial input besides fitness function is a collection of *building blocks.* It is collection of symbols (accompanied with an information about number of arguments). Those symbols are used to construct trees representing individuals.

Let us describe GP algorithm briefly:

At the beginning initial population is generated from building blocks randomly.

A step of GP algorithm is stochastic transformation of the current population into the next population.

This step consists of two sub steps[1]:

- Selection of *parents* for individuals of the next population based on the fitness. The bigger fitness of an individual of the current population is, the better chance of success being selected as parent it has.

- Application of genetic operators (such as *crossover*, *reproduction* and *mutation*) on parent individuals producing new individuals of the next population.

This transformation is repeatedly applied for a predefined number of steps (which is called number of *generations*) or until some predefined criterion is met.

Let us now look on GP at more detail.

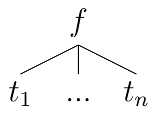
[1]TODO : Technically it is done in a little bit different fashion which is equivalent.

1.1.1 Program trees

In GP programs are represented as expressions. Let us define expression inductively:

- Constant[2] or variable symbol s is expression.
- Let there be a function with symbol f which has n arguments. And let there be expressions $e_1, ..., e_n$. Then (f e_1 ... e_n) is expression [3].

There is straightforward tree representation corresponding to these two cases:

- One node tree s.
- 

  Where $t_1, ..., t_n$ are trees corresponding to expressions $e_1, ..., e_n$.

Computer program with inputs $x_1, ..., x_n$ is realized as expression in which may occur variables $x_1, ..., x_n$.

1.1.2 Building blocks

Set of building blocks consists of two sets.

- Terminal set T : Set of symbols used as leaf nodes of program trees standing for constants and variables.
- Function set F : Set of symbols used as internal nodes of program trees standing for functions.

Therefore *building blocks* $= T \cup F$.

Beside symbols in $T \cup F$ there must also be an implementation for each symbol which is not variable. And for every function symbol from F there must be specified number of arguments.

There is one important constrain on function implementations for functions from F: There should be type A that for every function symbol $f \in F$ the corresponding function implementation standing behind this symbol should be of a type $A \times ... \times A \rightarrow A$ and should be total (defined on all combinations of inputs). And analogically for $t \in T$ being of a type A.
Satisfying this constrain ensures that every program tree build from $T \cup F$ will be total.

We can say that the motivation behind this work is to construct system where we eliminate this constraint.

[2]By constants we also mean procedures with zero arguments.

[3]This notation comes from Lisp programming language, classical notation would be $f(e_1, ..., e_n)$.

1.1.3 Generating initial population

We will describe tree generating method used by Koza in [1] called *ramped half-and-half.*

TODO!

1.1.4 Selection

In EA there is plenty of options for selection mechanisms for us to choose from. Again we will describe mechanism Koza used in his first book on GP. It is *Roulette selection.* It uses fitness value of each individual in straightforward way to determine probability of selecting this individual.

Let there be *popSize* individuals in the population.
And let f_i be fitness value for individual i where $i \in \{1 \ldots popSize\}$.
Then p_i probability of selection of individual i is computed as follows:

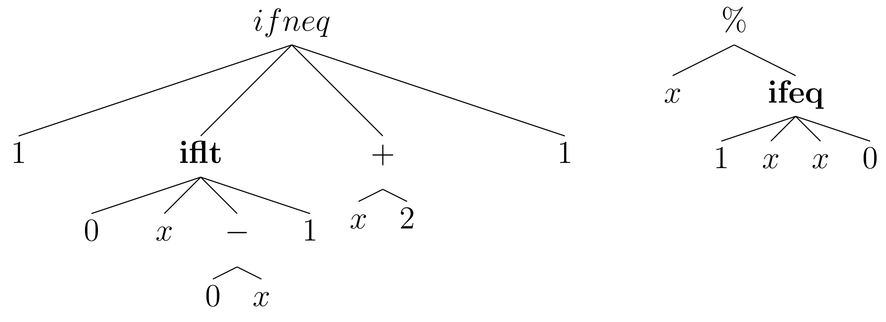
$$p_i = \frac{f_i}{\Sigma_{j=1}^{popSize} f_j}$$

1.1.5 Crossover

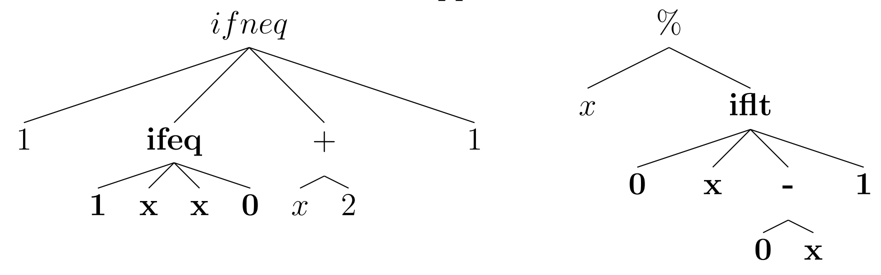
For John Koza the most important genetic operator in GP is crossover. It is operator inspired by sexual reproduction occurring in the nature. Generally speaking crossover takes two (parent) individuals and combines theirs genomes to produce two possibly new (child) individuals.

In GP the most common mutation is *Subtree swapping mutation.* This crossover randomly selects one node in each parent tree. Two new child individuals are constructed by swapping subtrees which have roots in those selected nodes.

Example should clarify this process. Here are two parent trees with selected nodes in bold:



And here are two child trees with swapped subtrees:



1.1.6 Reproduction

1.1.7 Mutation

1.1.8 GP algorithm in pseudocode

Here is pseudocode for GP algorithm:

```
Algorithm GP(fitness,buildingBlocks,numGens,popSize,probabs)
    $gen \leftarrow 0$
    $pop \leftarrow generateInitialPopuletion(buildingBlocks)$
    $(popWithF, terminate, best) \leftarrow$ evaluate($fitness$, $pop$)
    while $gen < numGens \wedge \neg terminate$ do
        $newPop \leftarrow$ empty population
        $newPop$.insert( $best$ )
        $i \leftarrow 1$
        while $i < popSize$ do
            $op \leftarrow probabilisticallySelectOperation(probabs)$
            switch $op$ do
                case $Crossover$
                    $parent1 \leftarrow$ selection( $popWithF$ )
                    $parent2 \leftarrow$ selection( $popWithF$ )
                    ($child1$,$child2$) = crossover( $parent1$ , $parent2$ )
                    $newPop$.insert( $child1$ )
                    $newPop$.insert( $child2$ )
                    $i \leftarrow i + 2$
                case $Reproduction$
                    $indiv \leftarrow$ selection( $popWithF$ )
                    $newPop$.insert( $indiv$ )
                    $i \leftarrow i + 1$
                case $Mutation$
                    $indiv \leftarrow$ selection( $popWithF$ )
                    $mutant \leftarrow$ mutate( $indiv$ )
                    $newPop$.insert( $mutant$ )
                    $i \leftarrow i + 1$
        $pop \leftarrow newPop$
        $(popWithF, terminate, best) \leftarrow$ evaluate($fitness$, $pop$)
        $gen \leftarrow gen + 1$
    return pop
```

In order to clarify the code let us describe in greater detail inputs and contained procedures generateInitialPopuletion(), evaluate(), probabilisticallySelectOperation(), selection(), crossover() and mutation().

1.1.9 Input

1.1.10 Output

...

1.1.11 TODO

- TALK ABOUT GP is part of EA etc. and maybe define the GP by defining EA and then specifying the differences or something like that...
- History, citations, etc ....
- ten algoritmus v kozovi dělá ”Designate Result” já tam vracim poslední populaci

1.2 Lambda term

Let V be set of *variable names.*
Let C be set of *constant names.*
Then Λ is set of λ*-terms* inductively defined as follows:

$$x \in V \cup C \Rightarrow x \in \Lambda$$
$$M, N \in \Lambda \Rightarrow (MN) \in \Lambda$$
$$x \in V, M \in \Lambda \Rightarrow (\lambda x.M) \in \Lambda$$

TODO

- TALK ABOUT ”parenthesis” conventions (and packing of lambda abstractions).
- BETTER SPECIFICATION V is infinite spočetná (?countable)

1.3 Type

Let A be set of *atomic type names.*
Then $\mathbb{T}$ is set of *types* inductively defined as follows:

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

TODO

- TALK ABOUT $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \alpha$
- TALK ABOUT arrow/parenthesis conventions.

1.4 Statement of a form $M : \sigma$

Let Λ be set of λ*-terms.*
Let $\mathbb{T}$ be set of *types.*
A *statement* $M : \sigma$ is a pair $(M, \sigma) \in \Lambda \times \mathbb{T}$.
$M : \sigma$ is vocalized as *”M has type σ”*.[4]

[4] $M : \sigma$ can be also imagined as $M \in \sigma$

The type σ is the *predicate* and the term M is the *subject* of the statement.

1.5 Context

Let $\Gamma \in \mathfrak{P}\,(\Lambda \times \mathbb{T})$. ($\Gamma$ is a set of *statements* of a form $M : \sigma$.)
Then Γ is *context* if it obeys following conditions[5]:

$$\forall (x,\sigma) \in \Gamma : x \in V \cup C$$
$$\forall s_1, s_2 \in \Gamma : s_1 \neq s_2 \Rightarrow \pi_1(s_1) \neq \pi_1(s_2)$$

In other words context is a set of statements with distinct variables or constants as subjects.

TODO: TALK ABOUT Context represents library/building blocks.

1.6 Statement of a form $\Gamma \vdash M : \sigma$

By writing $\Gamma \vdash M : \sigma$ we say *statement* $M : \sigma$ *is derivable from context* Γ .

We construct valid statements of form $\Gamma \vdash M : \sigma$ by using inference rules.

1.7 Inference rule

Basically speaking, inference rules are used for deriving statements of a form $\Gamma \vdash M : \sigma$ from yet derived statements of such a form. Those inference rules are written in the following form:

$$\frac{\Gamma_1 \vdash M_1 : \sigma_1 \qquad \Gamma_2 \vdash M_2 : \sigma_2 \quad \cdots \quad \Gamma_n \vdash M_n : \sigma_n}{\Gamma_{n+1} \vdash M_{n+1} : \sigma_{n+1}}$$

Suppose we have yet derived statements $\Gamma_1 \vdash M_1 : \sigma_1, \Gamma_2 \vdash M_2 : \sigma_2, \ldots, \Gamma_n \vdash M_n : \sigma_n$. It allows as to use the inference rule to derive statement $\Gamma_{n+1} \vdash M_{n+1} : \sigma_{n+1}$.

For deriving statements including types of a form $(\sigma \to \tau)$ are essential those two inference rules:

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}$$

$$\frac{\Gamma \cup \{(x,\sigma)\} \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : \sigma \to \tau}$$

This kind of inference rules allows us to derive new statements from yet derived statements, but what if we do not have any statement yet? For this purpose we have other kinds of inference rules such as *axiom* inference rule:

$$\frac{(x,\sigma) \in \Gamma}{\Gamma \vdash x : \sigma}$$

Let us consider an example statement of a form $\Gamma \vdash M : \sigma$:

[5] The π_1 corresponds to the projection of the first component of the Cartesian product.

$$\{\} \vdash (\lambda f.(\lambda x.(fx))) : (\sigma \to \tau) \to (\sigma \to \tau)$$

This statement is derived as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{(f, \sigma \to \tau) \in \{(f, \sigma \to \tau), (x, \sigma)\}}{\{(f, \sigma \to \tau), (x, \sigma)\} \vdash f : \sigma \to \tau} \qquad \dfrac{(x, \sigma) \in \{(f, \sigma \to \tau), (x, \sigma)\}}{\{(f, \sigma \to \tau), (x, \sigma)\} \vdash x : \sigma}}{\{(f, \sigma \to \tau), (x, \sigma)\} \vdash (fx) : \tau}}{\{(f, \sigma \to \tau)\} \vdash (\lambda x.(fx)) : \sigma \to \tau}}{\{\} \vdash (\lambda f.(\lambda x.(fx))) : (\sigma \to \tau) \to (\sigma \to \tau)}$$

1.8 Term generating grammar

Inference rules are good for deriving statements of a form $\Gamma \vdash M : \sigma$, but our goal is slightly different; we would like to generate many λ-terms M for a given type σ and context Γ.

Our approach will be to take each inference rule and transform it to a rule of term generating grammar. With this term generating grammar it will be much easier to reason about generating λ-terms.

It won't be a grammar in classical sense because we will be operating with infinite sets of nonterminal symbols and rules. [6]

Let $Non = Type \times Context$ be our *nonterminal* set. So for every $i \in Non$ is $i = (\sigma_i, \Gamma_i)$.

Let us consider each relevant inference rule and its corresponding grammar rule.

First inference rule is *implication elimination* also known as *modus ponens*:

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}$$

For every $\sigma, \tau \in \mathbb{T}$ and for every *context* $\Gamma \in \mathfrak{P}(\Lambda \times \mathbb{T})$ there is a grammar rule of a form[7]:

$$(\tau, \Gamma) \longmapsto \Bigg((\sigma \to \tau, \Gamma) \;\text{␣}\; (\sigma, \Gamma) \Bigg)$$

Second inference rule is *implication introduction*:

$$\frac{\Gamma \cup \{(x, \sigma)\} \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : \sigma \to \tau}$$

$\forall \sigma, \tau \in \mathbb{T}\ \forall \mathit{context}\ \Gamma \in \mathfrak{P}(\Lambda \times \mathbb{T})\ \forall x \in V$ such that there is no $(x, \rho) \in \Gamma$ there is a grammar rule:

$$(\sigma \to \tau, \Gamma) \longmapsto \Bigg(\;\boldsymbol{\lambda}\ \mathrm{X}\ .\ \ (\tau, \Gamma \cup \{(x, \sigma)\}) \quad \Bigg)$$

[6]TODO : mention terminal symbols - situation around variables and their construction with ' symbol.

[7] Terminal symbols for parenthesis and normally *space* now ␣ (for *function application* operator) are visually highlighted.

Third inference rule is *axiom*:

$$\frac{(x,\sigma)\in\Gamma}{\Gamma\vdash x:\sigma}$$

$\forall\sigma\in\mathbb{T}\ \forall context\ \Gamma\in\mathfrak{P}(\Lambda\times\mathbb{T})\ \forall x\in V\cup C$ such that $(x,\sigma)\in\Gamma$ there is a grammar rule:

$$(\sigma,\Gamma)\longmapsto\ \mathrm{X}$$

We will demonstrate λ-term generation on example. Again on $(\lambda f.(\lambda x.(fx)))$. We would like to generate λ-term of a type $(\sigma\to\tau)\to(\sigma\to\tau)$ with $\Gamma=\{\}$.

$$((\sigma\to\tau)\to(\sigma\to\tau),\{\})$$
$$\longmapsto\Big(\ \lambda\mathrm{f}.(\sigma\to\tau,\{(f,\sigma\to\tau)\})\ \Big)$$
$$\longmapsto\Big(\ \lambda\mathrm{f}.\ \Big(\ \lambda\mathrm{x}.\ (\tau,\{(f,\sigma\to\tau),(x,\sigma)\})\ \Big)\ \Big)$$
$$\longmapsto\Big(\ \lambda\mathrm{f}.\ \Big(\ \lambda\mathrm{x}.\ \Big((\sigma\to\tau,\{(f,\sigma\to\tau),(x,\sigma)\})\ \llcorner\!\lrcorner\ (\sigma,\{(f,\sigma\to\tau),(x,\sigma)\})\Big)\ \Big)\ \Big)$$
$$\longmapsto\Big(\ \lambda\mathrm{f}.\ \Big(\ \lambda\mathrm{x}.\ \Big(\ \mathrm{f}\ \llcorner\!\lrcorner\ (\sigma,\{(f,\sigma\to\tau),(x,\sigma)\})\Big)\ \Big)\ \Big)$$
$$\longmapsto\Big(\ \lambda\mathrm{f}.\ \Big(\ \lambda\mathrm{x}.\ \Big(\ \mathrm{f}\ \llcorner\!\lrcorner\,\mathrm{x}\Big)\ \Big)\ \Big)$$

1.8.1 "Barendregt-like" inference and grammar rules

Inference rule 1:

$$\frac{\Gamma\cup\{(x_1,\tau_1),\dots,(x_n,\tau_n)\}\vdash M:\alpha}{\Gamma\vdash(\lambda x_1\dots x_n.M):\tau_1\to\dots\to\tau_n\to\alpha}$$

Proof of correctness:

$$\cfrac{\cfrac{\cfrac{\Gamma\cup\{(x_1,\tau_1),\dots,(x_n,\tau_n)\}\vdash M:\alpha}{\Gamma\cup\{(x_1,\tau_1),\dots,(x_{n-1},\tau_{n-1})\}\cup\{(x_n,\tau_n)\}\vdash M:\alpha}}{\Gamma\cup\{(x_1,\tau_1),\dots,(x_{n-1},\tau_{n-1})\}\vdash(\lambda x_n.M):\tau_n\to\alpha}}{\cfrac{\vdots}{\Gamma\vdash(\lambda x_1\dots x_n.M):\tau_1\to\dots\to\tau_n\to\alpha}}$$

... there is a grammar rule:

$$(\tau_1\to\dots\to\tau_n\to\alpha,\Gamma)\longmapsto\Bigg(\ \lambda x_1\dots x_n.\ (\alpha,\Gamma\cup\{(x_1,\tau_1),\dots,(x_n,\tau_n)\})\ \ \Bigg)$$

Inference rule 2:

$$\frac{(f,\rho_1 \to \cdots \to \rho_m \to \alpha) \in \Gamma \qquad \Gamma \vdash M_1 : \rho_1 \quad \cdots \quad \Gamma \vdash M_m : \rho_m}{\Gamma \vdash (fM_1 \dots M_m) : \alpha}$$

Proof of correctness (**TODO REPAIR** Conceptually it is ok but there is sazbabug somewhere):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\boxed{(f,\rho_1 \to \cdots \to \rho_m \to \alpha) \in \Gamma}}{\Gamma \vdash f : \rho_1 \to \cdots \to \rho_m \to \alpha} \quad \boxed{\Gamma \vdash M_1 : \rho_1}}{\Gamma \vdash (fM_1) : \rho_2 \to \cdots \to \rho_m \to \alpha} \quad \ddots}{\vdots}}{\Gamma \vdash (fM_1 \dots M_{m-2}) : \rho_{m-1} \to \rho_m \to \alpha} \quad \boxed{\Gamma \vdash M_{m-1} : \rho_{m-1}}}{\Gamma \vdash (fM_1 \dots M_{m-1}) : \rho_m \to \alpha} \quad \boxed{\Gamma \vdash M_m : \rho_m}}{\Gamma \vdash (fM_1 \dots M_m) : \alpha}$$

... there is a grammar rule:

$$(\alpha,\Gamma) \longmapsto \Big(\text{f} \;\text{␣}\; (\rho_1,\Gamma) \;\text{␣}\; \dots \;\text{␣}\; (\rho_m,\Gamma) \Big)$$

TODO

- SHOW correctness of those inference rules by composing them of $E^{\to}$, $I^{\to}$ and *axiom*.
- SHOW more examples of inference rules transformed into grammar rules.
- DESCRIBE general algorithm for this transformation.
- TALK ABOUT $\tau_1 \to \cdots \to \tau_n \to \alpha$
- TALK ABOUT $\beta\eta$-normal form which is generated by this method.

1.9 Inhabitation tree

Now we will introduce *Inhabitation tree*, structure slightly different from *Inhabitation machine*, which was introduced in [2] by Henk Barendregt. We can think about Inhabitation tree as about unfolded Inhabitation machine. The motivation for using Inhabitation trees is belief that it will help us reason about generation of λ-terms of a given type σ and with a given context Γ.

1.9.1 Definition of Inhabitation tree

Inhabitation tree is a *rooted tree*, possibly infinite. It has two types of nodes:

- Type nodes - containing type $\sigma \in \mathbb{T}$ - aka "OR-node" , Nonterminal-node.
- Symbol nodes - containing "λ-head" (nonempty finite sequence of variable names) or constant name. - aka "AND-node" , Terminal-node.

We construct Inhabitation tree for given type σ and context Γ.
We will define Inhabitation tree by describing its construction for a given (σ,Γ).

Notice that it will closely follow the rules from 1.8.1:

- The root of Inhabitation tree for (σ, Γ) is *type node* with σ as type.
- All *type nodes* have as child nodes only *symbol nodes.*
- And all *symbol nodes* have as child nodes only *type nodes.*

Now we will resolve the child nodes of the root node.
There are two cases of σ (recall 1.3):

Atomic type $\sigma = \alpha$ where $\alpha \in A$.

Function type $\sigma = \tau_1 \to \cdots \to \tau_n \to \alpha$ where $n \geq 1, \alpha \in A$.

First case **Atomic type** — i.e., $\sigma = \alpha$ where $\alpha \in A$:
For every $(f, \rho_1 \to \cdots \to \rho_m \to \alpha) \in \Gamma$ where $\alpha \in A$ there is a child *symbol node* of the root containing constant name f. This symbol node containing f has m child subtrees corresponding to Inhabitation trees for $(\rho_1, \Gamma), \ldots, (\rho_n, \Gamma)$.

Compare this case with corresponding grammar rule:

$$(\alpha, \Gamma) \longmapsto \Big(\mathrm{f} \ \sqcup \ (\rho_1, \Gamma) \ \sqcup \ \ldots \ \sqcup \ (\rho_m, \Gamma) \ \Big)$$

Second case **Function type** — i.e., $\sigma = \tau_1 \to \cdots \to \tau_n \to \alpha$ where $n \geq 1, \alpha \in A$:
For every $i \in \{1, \ldots, n\}$ we create new *variable name* x_i which is not yet included in context Γ as variable or constant name.

There is one and only one child *symbol node* of the root containing "λ-head" $\lambda x_1 \ldots x_n$ which stands for sequence of variable names $(x_1, \ldots, x_n)$. This symbol node containing $\lambda x_1 \ldots x_n$ has one and only one child subtree corresponding to Inhabitation trees for $(\alpha, \Gamma \cup \{(x_1, \tau_1), \ldots, (x_n, \tau_n)\})$.

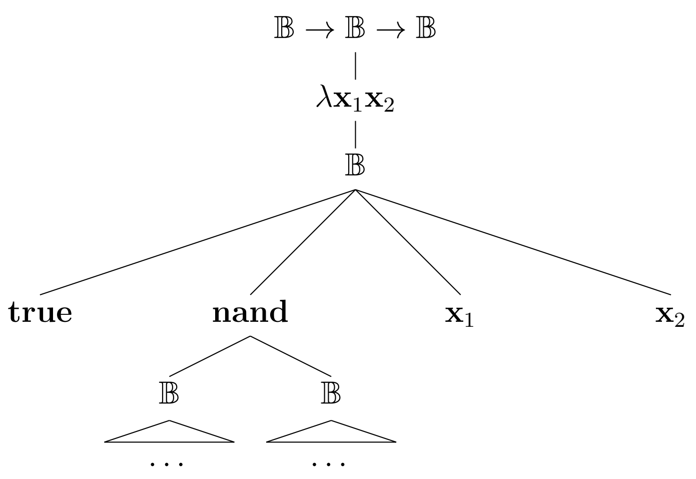
Compare this case with corresponding grammar rule:

$$(\tau_1 \to \cdots \to \tau_n \to \alpha, \Gamma) \longmapsto \Big(\lambda x_1 \ldots x_n . \ (\alpha, \Gamma \cup \{(x_1, \tau_1), \ldots, (x_n, \tau_n)\}) \ \Big)$$

Let us consider following (σ, Γ) as a simple example:

$$\begin{aligned} \sigma &= \mathbb{B} \to \mathbb{B} \to \mathbb{B} \\ \Gamma &= \{ \ \mathit{true} : \mathbb{B} \\ &\quad\ , \ \mathit{nand} : \mathbb{B} \to \mathbb{B} \to \mathbb{B} \ \} \end{aligned}$$

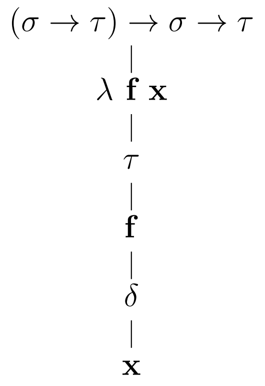
This particular (σ, Γ) results in the following tree:



Second example features our well known example:

$$\sigma = (\sigma \to \tau) \to \sigma \to \tau$$
$$\Gamma = \{\}$$

Which results in following tree:


$(\sigma \to \tau) \to \sigma \to \tau$
$\lambda\ \mathbf{f}\ \mathbf{x}$
τ
$\mathbf{f}$
δ
$\mathbf{x}$

1.9.2 And-or tree and searching in Inhabitation tree

Let us consider following definition of *And-or tree*[8]:

And-or tree is a rooted tree where each node is labeled as either *and-node* or[9] *or-node.*

By *solving* And-or tree T we mean finding T' subtree of T such that it follows these conditions:

- The root of T' is the root of T.
- Each *and-node* in T' has all the child nodes as in T.
- Each *or-node* in T' has precisely one child node.[10]

Let us now consider following labeling of Inhabitation tree:

[8] **TODO**: Mention that on WIKI there is more general definition, but for our purposes is this one sufficient.

[9] xor

[10] **TODO**: MENTION why precisely one and not at least one ..or CHANGE the def.

- **Type nodes** are labeled as **or-nodes**.
- **Symbol nodes** are labeled as **and-nodes**.

This labeling has following justification:

Selection of exactly one child node in *type node* corresponds to selection of exactly one grammar rule in order to rewrite nonterminal symbol.

Selection of all the child nodes in *symbol node* corresponds to rewriting all the nonterminal symbols in string that is being generated.

The motivation for defining *solving* of a And-or tree the way we did is that a found tree T' corresponds to generated λ-term. In order to understand this correspondence let's now talk about various tree representations of λ-terms.

1.9.3 Tree representations of λ-terms

From the definition of λ-term (1.2) we can straightforwardly derive the classical tree representation for λ-terms. Term M is translated into tree $T[M]$ by following rules:

- $x \in V \cup C$ translates into *leaf* x.
- (MN) translates into tree
$$\begin{tikzcd} & @ \arrow[dl, dash] \arrow[dr, dash] & \\ T[M] & & T[N] \end{tikzcd}$$
- $\lambda x.M$ translates into tree
$$\begin{tikzcd} \lambda x \arrow[d, dash] \\ T[M] \end{tikzcd}$$

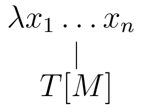
We can enhance this representation by compressing consecutive lambda abstractions into one tree node like this:

- $\lambda x_1 \dots x_n.M$ translates into tree
$$\begin{tikzcd} \lambda x_1 \dots x_n \arrow[d, dash] \\ T[M] \end{tikzcd}$$

As this representation comes directly from definition it is evident that it covers all possible λ-terms.

For representing expressions as trees it is however more common use a little different representation. It will also be the representation suitable for showing that *solving* Inhabitation tree generates wanted λ-term.

- $x \in V \cup C$ translates into *leaf* x.
- $(fM_1M_2 \dots M_n)$ where $f \in V \cup C, n \geq 1$ translates into tree
$$\begin{tikzcd} & & \mathrm{f} \arrow[dll, dash] \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\ T[M_1] & T[M_2] & \dots & T[M_n] \end{tikzcd}$$

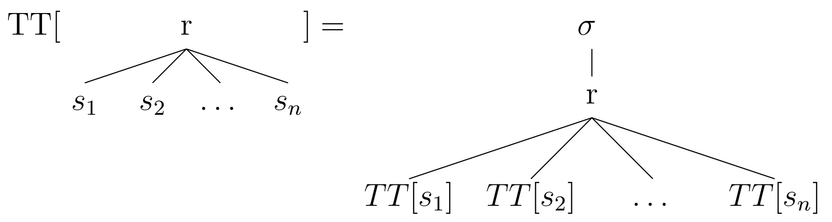
- $\lambda x_1 \dots x_n.M$ translates into tree
  

Notice that this representation does not cover all λ-terms, e.g. $(\lambda x.x)y$ is not expressible in it. But it does not bother us.

Let us now consider representation for *typed λ-terms*. Straightforward approach would be to add to each node a type entry which would be the type of the λ-term corresponding to subtree having this node as the root node.

Approach more suitable for our purpose is to add a special type node above each node. More specifically:

Let us consider tree t corresponding to a λ-term of a type σ with root r and subtrees $s_1, \dots, s_n$. Then corresponding tree $TT[t]$ for typed λ-term is obtained from the tree t as follows:



Now we can finally put the pieces together. Every solution to a Inhabitation tree has this just described tree form of a typed λ-term.

TODO

- EXAMPLES of tree representations of λ-terms
- TALK (more?) ABOUT "Barendregt-like" subsection 1.8.1
  Things about $\beta\eta$ normal form, etc.

1.9.4 Our approach to *solving* Inhabitation machine

A* algorithm

1.9.5 Inhabitation Machine

TODO

- DESCRIBE Inhabitation Machine...

1.10 Roadmap

1.11 Conversion to SKI combinators

2. Designed system

2.1 Top level view

2.1.1 Comments about main source files

Eva.hs

GP_Core.hs

2.2 Term generating

2.2.1 A* algorithm

2.3 Crossover

2.3.1 Finding same types

2.3.2 Two basic options

Resolve problems with free variables or avoid variables completely.

2.4 Mutation

2.4.1 Using term generation

3. Problems

In this section will be presented usage of the system in order to solve specific problems.

3.1 Even Parity Problem

3.2 Big Context

3.3 Fly

3.4 Simple Symbolic Regression

3.5 Artificial Ant

3.6 Boolean Alternate

Conclusion

Bibliography

[1] John R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[2] Henk Barendregt, Wil Dekkers, Richard Statman, *Lambda Calculus With Types*. Cambridge University Press, 2010.
`http://www.cs.ru.nl/~henk/book.pdf`